Michael Pinna

A Secure Card Game

Computer Science Tripos (Part II)

Gonville and Caius College

May 15, 2002

Proforma

Name:	**Michael Pinna**
College:	**Gonville & Caius College**
Project Title:	**A Secure Card Game**
Examination:	**Computer Science Tripos (Part II), June 2002**
Word Count:	**10054**
Project Originator:	**M. A. Pinna**
Supervisor:	**Dr I. W. Jackson**

Original Aims of the Project

To implement a secure distributed card game in C using a standard cryptographic API. This requires that, with no trusted intermediary, each player be able to get a set of cards and have confidence that they are a random selection from the shared deck, and that no other player is able to influence or observe the dealing process. No player should have to reveal his cards unless the rules of the game require it, yet if a player attempts to cheat by subverting the dealing process or lying about his cards, each other player should be able to detect this.

Work Completed

No implementations of cryptographic schemes with the correct properties were found, therefore it was decided to implement from scratch a protocol known as All-or-Nothing Disclosure Of Secrets, for which the Author was required to gain an understanding of new areas of mathematics. While the additional work took up a large portion of the time which had been allocated to the implementation of the card game itself, most of the code for the game was also completed and tested. All additional data required for players to prove they have behaved fairly is also distributed.

Special Difficulties

None.

Declaration

I, Michael Pinna of Gonville & Caius College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does no contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Card games have a long and distinguished history. The first recorded use of a deck of cards was in the mid-15th Century, by the wealthy Visconti family of Milan who played with a 78-card Tarot deck. It is thought that the deck commonly in use today, with its familiar four suits of thirteen cards each, was developed in France in approximately 1480. Since then, thousands of games have been developed using this apparently straightforward pile of pieces of paper. Some of these became very popular and are commonly played today; most of the others have eventually been forgotten. The rules of many popular games have not changed for several centuries, nor has the manner in which people get together to play them. Recently however, this has begun to change.

The advent of the computer age allowed people to play multi-player card games when there was no other person around to play with. As processing power and game-playing algorithms improved over the years, it became possible for individuals to play games with no need for human companions.

These games are far more enjoyable though if ones choices in a game have effects on another human being, not to mention the fact that people are still capable of playing in a far more unpredictable and innovative way than machines, so if there is any chance of playing against other people, we would much prefer to use our computers as a means of communication, rather than as an opponent.

With the recent explosive growth in availability of distributed networks such as the Internet, many people have become interested in the prospect of playing games with others who are far away enough to make such actions as the dealing of a physical deck inconvenient at best. Maybe they would like to be able to play their weekly game of Bridge even though one of the usual players is abroad on business, or perhaps they wish to gamble away their week's earnings playing Blackjack, but the nearest Casino is hundreds of miles away.

The Internet solves the problem of communication between the parties wishing to play in situations like these. Now however, the security of the game becomes an important issue, particularly if large amounts of money are at stake in an online Casino environment or similar. The simplest solution for people wanting to play friendly games would be to set up a system where a trusted server run by a third

party acts as an intermediary between the players, dealing cards to each of them and ensuring that nobody cheats during the game. However in the case of the online Casino, both the player and the Casino have a strong interest in the fairness of the system, so they might want to ensure that no single party has enough control over the system to be able to influence the outcome of the game. This means having a trusted server is not an option, as neither player can ever be entirely sure it is behaving in a fair manner. Thus we need to find a protocol which is truly 'distributed', which means that all secret information and trusted computation is shared among the players.

1.2 Possible Attacks

A useful description of the attacks to which our protocol might be subjected was given by Schindelhauer [4].

Sabotage Bad loser prevents game completion by refusing to cooperate in protocols, for example by disconnecting another player.

Protocol attack Player lies about properties of numbers generated (using a number that is not random where it ought to be, or a number with more prime factors than claimed).

Cheating Break rules of underlying game, eg playing the wrong card.

Secret coalitions Secret information exchange between players. This will be impossible to prevent, but it should be possible to prevent the coalition from gaining any additional information about other players' hands.

Public coalitions A group of players discriminating against another group without any secret information exchange. Purely a social issue, hence impossible to deal with.

Ghost players A player invents new virtual players to increase the probability of winning. Undetectable.

1.3 History of Secure Card Game Protocols

Initial attempts at making protocols for secure card games were all seriously limited in some way. In 1984 Shamir, Rivest and Adleman[1] proved it to be impossible in an information-theoretic sense, but that practical schemes could exist based on the computational difficulty of inverting certain cryptographic primitives. Some required a trusted third-party to ensure that all rules were followed during the game, and others only worked in a game involving just two players.

The first workable solution for the game of electronic Poker was provided by Crepeau in [2]. In the original paper, the solution proposed has one weakness: When checking at the end of the game that no player has cheated, each player is required to disclose details of exactly which cards were held at all stages of the

[1] http://citeseer.nj.nec.com/context/33713/0

game. This may reveal information about the player's bluffing behaviour, which is a problem that does not occur in the ordinary game using paper playing cards.

Briefly, Crepeau's solution involves each player generating a pseudorandom permutation of the numbers 1 to 52, and using the composition of all players' permutations to produce a permutation of an unshuffled deck which is effectively random (from the point of view of any one player) and can not be computed by any player without cooperation from all others. Various elaborate functions are then defined which are to be used by each player to control the reading of elements of his permutation and to check that this player does so honestly. At the end of the game, each player is required to reveal the secret information used in computing these functions so each other player may verify that all was done fairly. This is the step that has the side-effect of allowing every player to deduce the exact behaviour of each other player all the way through the game.

In [3] Crepeau adds mechanisms to his original solution which allow the honesty of each player to be checked without requiring all cards to be revealed. Each entry in every deck permutation which may be read using the secret-sharing scheme has additional information associated with it, which is read by a player at the same time as reading the entry itself. To detect cheating, it then suffices for each player to show that he has the correct extra information, without needing to say which elements of the permutations were actually read. Therefore no information about which cards are held by any player is derivable at any time.

The most recent advance in this field was by Schindelhauer in [4], where he puts together a generic toolbox for card games which might require more complex operations than Poker, allowing for example the use of multiple decks, players joining and leaving the game while it is running, cards being returned to the deck or players exchanging cards with one another. Unfortunately the complexity of this protocol made it infeasible for implementation as part of this project.

In this project it was decided to implement a portion of Crepeau's second solution, with the intention that the system to be produced be suitable for playing a real game of Poker with all the required security properties being satisfied.

Chapter 2

Preparation

2.1 Requirements Analysis

The first thing that needs to be done in putting together a framework for secure distributed play of card games is to decide what security properties the system ought to have. A good list of the constraints that should be satisfied by a protocol for playing a general game of cards was provided by Crepeau in [2]:

Uniqueness of cards. In the deck of cards, each member of the standard deck of 52 must appear exactly once.

Uniform random distribution of cards. When a card is dealt, each card which is not currently held by a player must have an equal probability of being chosen.

Absence of trusted third party. As explained above, all secret information and trusted computation must be shared among the players.

High probability of cheating detection. Each player should be able to discover if another player has violated the game rules or the playing protocol.

Complete confidentiality of cards. No player should be able to gain any information about the cards of another, except for that which he knows about his own hand and any cards which may have been publicly revealed.

Minimal effect of coalitions. Two players who choose to share information covertly should not be able to gain any information about the state of the game based on anything more than the cards each of them have seen as individuals.[1]

Complete confidentiality of strategy. The security protocol should not force a player to reveal his cards if the game rules do not require it. For example, in the game of Poker, a player may prefer not wish to show his hand unless it is necessary, as this will reveal information about his bluffing behaviour which is best kept secret for the sake of future rounds.

[1] In games such as Bridge, where players form partnerships while being unable to see each other's hands, the possibility of covert coalitions such as these introduces severe complications to any protocol for serious play in a distributed environment.

2.2 All-or-Nothing Disclosure Of Secrets

The protocol described in [3] relies on a cryptographic scheme with some interesting properties named 'All-or-Nothing Disclosure Of Secrets[2]' (shortened to 'ANDOS') for the data transmitted between players in the card-dealing process. This is a probabilistic encryption method which allows a player to ask the originator of some encrypted data to decrypt the data without the latter being aware of which data is being decrypted.[3] This is possible due to the fact that the reader of the data encodes the encrypted data as a question before asking the originator of the data to decrypt it, and then uses some secret data computed with the question to decode the decrypted data. We require the above property as, at the end of the process through which a card is dealt, one player will be asked to decrypt the item of data which says which card has been dealt, and if this player were in a position to view or modify this information he would always be able to see, and undetectably change, which card was being dealt.

See Figure 2.1 for a diagram showing the data that goes between originator and reader, and the operations that are performed at various stages of the process. This is rather complex and therefore warrants careful explanation. The bare protocol works on single bits at a time.

1. The originator of the bit to be shared encrypts the bit using a probabilistic scheme. The encrypted bit is sent to the reader, along with the encrypted versions of all the other bits he may wish to read.

2. When the reader decides that he wants to find out what one of these bits is, he encodes it as a 'secret question'. This consists of a question which will be asked of the originator, and some additional data which the reader keeps secret for his own later use.

3. The reader sends the question without its additional data to the originator.

4. The originator answers the question using the private key that he originally used to encrypt the secret. As it has been encoded by the reader, he is not able to link the question being answered to the original secret, and therefore does not know which of his secret data he is giving out. To prove that his answer to the question is not a lie, he is required to accompany it with a proof of its correctness.

5. Receiving the answer to his question and its proof, the reader now checks the proof to ensure he is not being cheated. If it is correct he combines the secret data he removed from his question with the answer to this question, and therefore decodes the originator's answer, getting the secret data.

In order to understand the inner workings of ANDOS it was necessary to become familiar with some elements of number theory, as the essence of its decryption function is the computation of the Jacobi symbol $J(y, n)$ in an efficient way using the knowledge that n factorises to pq.

[2]This was originally described by the same author in [1].
[3]See Section 2.3 for explanation of why this is required.

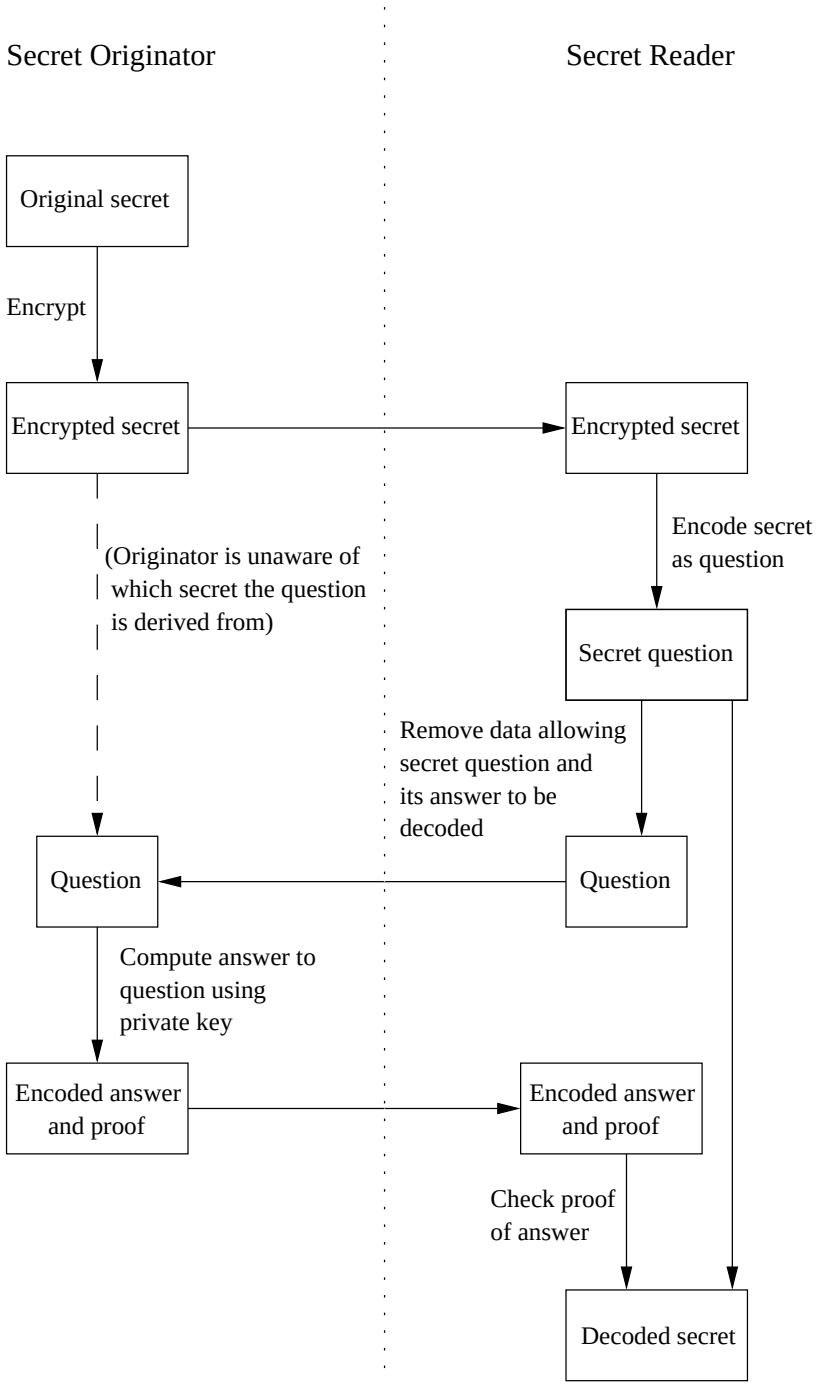

Figure 2.1: Flow of data in ANDOS

2.3 Shuffling the Deck

A deck of 52 cards will be maintained centrally, and when a player requires a new card he publicly chooses from here at random a card that has not already been selected by another player, and marks the fact that he is using it. The central deck is actually a random permutation of a deck of 52 in the standard order, which is produced by composing random permutations produced by each player. This is best explained by means of a simplified example with a diagram.

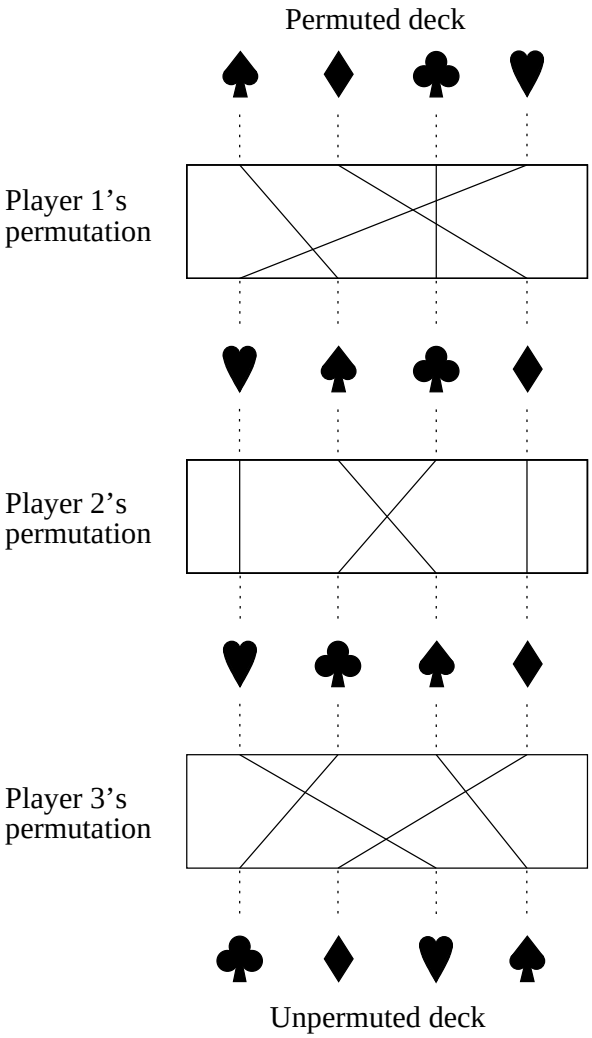

Figure 2.2: Shuffling the deck

In Figure 2.2 is an example of a four-card deck with three players involved in the shuffling process. If a player wants a card, he chooses a free one at random from the permuted deck, say the fourth. We know this is the ♡, but he needs to get some information from each other player to discover this fact. So he asks player 1 which position his permutation moves card 4 to, and that player says it goes to card 1. Player 2 then says that card 1 remains card 1 according to his permutation,

and player 3 says that this goes to card 3. As player 3's permutation maps on to the unpermuted deck which is in a known order, the information from him allows our player to determine that the real value of the card he has chosen is ♡. We can now see why it is sometimes necessary to use a secret-sharing scheme where the originator of a secret is unable to determine which data is being decrypted or replace it with falsified data, as if this were possible player 3 would be aware of exactly which cards each other player was being dealt and would be able to change them undetectably at any time.

Though this way of shuffling the deck appears to be elegant, it has one property that makes it unsuitable for most games apart from poker: once a player has discovered through legitimate means that a certain card in the permuted deck corresponds to a particular card in the known deck, he will be able to recognise this card if he ever discards it and another player happens to pick it up. This means that a great deal of care would be required if the game to be implemented here were to allow players to exchange cards from their hands with fresh cards from the deck. One way of achieving this would be, for example, to deal eight cards to each player rather than the usual five, but initially only allow each player to obtain enough secrets to discover the identity of five of them. Then if a player chose to discard any of his original five cards, he could publicly announce which ones from the permuted deck were being exchanged, at which point the other players would let him read further secrets from them to decode more of the cards that had been dealt to him. He would never be able to reuse the cards he had discarded, as all players would know which they were, even though they would not know their real identities.

This might work for the game of poker, if there are few enough players for the dealing of additional cards to each person not to be a problem, but now we come to the problem most other games would have with this system. If a different player should ever happen to pick up a card from the deck which has previously been in another player's hand, that other player will know which card is held. This is clearly unacceptable for practically any other game.

2.4 Selection of Tools

Once the ANDOS secret-sharing scheme had been chosen, it became clear that the project would be very computationally intensive (with modular-exponentiation operations to be performed by more than one player per bit of data that was to be transmitted). It was therefore decided that Java would be an unsuitable choice of programming language, and an alternative was sought. The most sensible way of structuring the code seemed to be as sets of functions in layers, as described in Section 2.6, so there was no need to put effort into mastering the complexities of C++, and therefore C was chosen.

The choice of development platform was fairly unimportant on a technical level, so Linux was chosen for its stability and the ease of working remotely. The GMP (GNU multiple precision arithmetic) library was chosen for the big integer and modular arithmetic computations.

It was necessary to get a good understanding of how to use the tools that had been chosen before beginning to write code for the project. The Author had very

limited prior experience of programming in C, so some preliminary example code was written to ensure a certain level of familiarity with the constructs which differ from Java: primarily the use of structures, pointers and pointer arithmetic, memory allocation and deallocation, and header files.

At the same time a Makefile was written to automate the process of compilation of the project. This required familiarisation with basic use of the `make` tool.

2.5 Changes to the Proposal

The original proposal for this project stated that a complete working card game would be built on top of standard encryption techniques. However as ANDOS is not a widely available secret-sharing method, it has been decided to implement this as part of the project, and as such it has become clear that it the project is unlikely to be able to go as far into the implementation of the card game as was originally intended. It has been decided to try and get as far as dealing a deck using all the right cryptographic techniques in such a manner that all data required to prove the security of the game is distributed correctly among the players. The proofs that players are playing fairly are to be omitted unless there is time at the end of the project.

2.6 Design of the Code

The code of the project has been divided up into four layers, each of which will use functionality provided by the layer below. These are as follows:

1. Non-ANDOS specific utility functions, such as general random number and random prime generators, a random permutation generator, and a set of functions for computing modular square roots.

2. Functions which perform the encryption, decryption, and related operations of ANDOS on single bits.

3. Functions which perform ANDOS operations on multiple bits of data.

4. Functions for generating and manipulating card-game data structures which get cards from the deck and communicate securely with other players to discover what they are.

2.6.1 Utility Functions

Various functions will required to perform 'standard' (ie not specific to ANDOS or the card game) operations through the implementation of the project.

Generation of Random C `ints`

It will be necessary to write a function which generates cryptographically secure random C `unsigned ints` i, where $0 \leq i < n$ for a given bound n. This should be just as efficient when generating integers in a large range as when generating single random bits.

Generation of Random Large Integers

This function should allow integers of arbitrary length l bits to be generated. It will most likely rely on a function provided by the GMP library.

Generation of Random Large Primes

This function will return integers of length exactly l bits to be generated which have a very high probability of being prime. It will be important to ensure it is not unnecessarily inefficient, as it is already likely to be very computationally intensive.

Computation of Modular Square Roots

A function will be required which can compute the square root of a number q modulo a number n whose two factors i and j are known. It will probably rely on a function able to compute square roots modulo a prime, to generate numbers s and t such that $s^2 = q \pmod{i}$ and $t^2 = q \pmod{j}$. These will then be combined to give the final result.

2.6.2 Encryption and Decryption of Single Bits

The basic operations of ANDOS will be performed on units of data corresponding to single bits of plaintext. It has been decided that a function should be written to implement each of these, in each case returning a data structure either to be kept for future use or to be sent to the other player. These functions will be written separately from those which perform the ANDOS operations on multiple bits of data, so that the correct functioning of the mathematical elements may be tested separately from the correctness of the parts which combine them. Their prototypes may be seen in Appendix A.1.

Key Generation

A private key in ANDOS consists of two large random primes, i and j, their product n, and a random number y satisfying $0 \leq y < n$, where y must be a quadratic non-residue (mod n) and the Jacobi symbol $J(y, n) = +1$. The public key contains just n and y; the computation of $J(v, n)$ for arbitrary v is considered a hard problem equivalent in difficulty to factoring n.

Encryption

An encrypted bit is simply a number e, where $0 \leq e < n$, which is a quadratic residue mod n if the original bit was 0, or a nonresidue if the bit was 1.

Making the Secret Question

Here the player wishing to read the bit generates some secret data and uses it to convert the encrypted value e into a value q which the secret originator will not recognise. He will generate a secret bit m. If this is 0, q will be a quadratic residue (mod n) if and only if e is also a residue. If m is 1, q will be a residue if and only if e is not. Without knowing the value of m, the originator will have no way of knowing

which bit he is decrypting or what its value is, but if the reader holds on to it he will be able to use the decrypted value to compute the real value of the original bit.

Removing Secret Data from the Question

This step just involves taking the value of q from the computed question. It is important to recognise it as a separate step, as it will involve manipulation of data structures when we get on to talking about encrypted data structures rather than isolated bits.

Answering the Question

The answer b to the question consists of a single bit indicating whether the value q was a quadratic residue (mod n). To prevent cheating by the player answering the question, he will be required to prove his answer correct by providing r, which satisfies $r^2 = q$ if it is a residue or $r^2 = q \times y$ if it is not.[4]

Decrypting the Bit

Using the value of m which was generated at the same time as the question, it will be straightforward to compute the value of the plaintext bit, as it is just $b \oplus m$.

2.6.3 Encryption and Decryption of Multiple Bits

The basic operations in ANDOS only work on single bits of plaintext at a time. It is therefore necessary to write data structures and functions which allow data structures containing an arbitrary number of bits to be put through the secret-sharing process. The definitions of these may be seen in `andos.h` which is included in Appendix A.1. Most of them contain an array of the corresponding structure for single bits along with an integer indicating how long this array is. The basic type for plaintext is clearly a single bit, so these are packed into an array of unsigned integers to avoid memory wastage. Functions need to be written to initialise each of these data structures, taking as arguments a pointer to the structure to be initialised and the length of the data to be placed in it. Similarly, functions will be written to clear out and unallocate the memory by these data structures.

2.6.4 Dealing of Cards

See Section 2.3 for a general explanation of the deck shuffling process. One practical issue that is not discussed there is that a player p getting a new card will at some stage need to perform a lookup in his own permutation, as he is involved in the shuffling process. As the value returned by this lookup is entirely secret, as there is no way another player may legitimately discover what it is, its input need not be hidden from the other players. It follows that all the lookups in the permutations of players before p may be done publicly, so no encryption is required.

If we consider the last player n, we can see that n may trivially work out which card p is getting if n is allowed to see which element in his permutation is looked up

[4] While $q \times y$ is not always a quadratic residue for general nonresidues q, it will be one in practice due to how q is constructed. See Section 3.2.5 for further explanation.

by p, as n's permutation maps directly onto the unpermuted deck. Similarly, if n and $n-1$ join forces, they can determine which card p is getting merely by knowing which element he looks up in the permutation belonging to $n-1$. Arbitrary length chains of cheats may form, until we get to p himself, who we assume would prefer not to share the identity of his card with the other players (and in any case he would have far easier ways of doing this if he chose to). Therefore all lookups on players from $p+1$ to n need to be done in such a manner that the player whose permutation is being consulted is unable to determine which part of the permutation he is giving away, and it is for this that we crucially need ANDOS.

Now, however, we need to find some way of checking that p is not looking up the wrong elements of permutations, as doing this would allow him to discover which cards another player holds.[5] One might have thought that this is not a real danger, as a player doing this would forsake his chance of discovering what his own card was, but there are possible circumstances in which finding out another player's card rather than one's own would give a net advantage. To solve this problem, we attach some additional data τ to each element of every player's permutation, which must be read along with the permutation element. At the end of the game, all players will be able to check that p did not read any π values corresponding to their cards, by checking that he has valid τ values corresponding to different elements of the permutation.

The definitions of the data structures from `player.h` may be seen in Appendix A.2, but their contents warrant some explanation. Each player in the game has a pointer to a structure named `player` which has pointers to the player's public and private data, along with the public data of all other players.

The private data contains this player's permutation π, all the unencrypted values of τ, the real identities of the cards in this player's hand, and his private key. For each player whose id is before that of the current player, 52 values of τ must be generated, each of which consists of an array of `SECURITY_PARAMETER` unsigned integers (where `SECURITY_PARAMETER` is defined in `player.h` to be 2, giving on a 32-bit architecture a probability of 2^{-64} that a player claiming to have read it can get away without having actually done so). Therefore τ must be an 3 dimensional array of unsigned integers of dimension `[id][52][SECURITY_PARAMETER]`.

The public data consists of numeric player id, public key, encrypted permutation, encrypted τ values, the number of cards in the hand, and the publicly-visible identity of the cards held from the permuted deck. The unencrypted π and τ arrays contain integers, so the encrypted versions of these are arrays of the same dimensions of encrypted integers.

[5] Recall from Section 2.1 that one of the requirements of this implementation is that players should not be required to show their cards at the end, unless the rules of the specific card game require it, so we are unable to use the obvious solution of making player p declare all secret data used in order that others may check it was done correctly.

Chapter 3

Implementation

Through the implementation of the project, various techniques were used to assist the debugging of functions as they were written, rather than leaving the testing of the project to be done after the main body of the code was written. The C `assert` library function was used to ensure that decrypted values, modular square roots and array indices had reasonable values before they were returned, and each layer of functions was tested carefully before writing the functions in the layer above. See Section 4.1.2 for a discussion of how this was done.

3.1 Assorted Utility Functions

3.1.1 Random Number Generation

Two functions were written for generating random numbers. The first was for generating random C integers between 0 and a limit n, and the second for generating random arbitrary precision integers with a given length of l bits.

Bounded Random Primitive C `ints`

The behaviour of the function `bounded_random` is somewhat counterintuitive, and its specification needed to be changed when the time came to implement it. Initially one might think that a function like this should generate non-negative results which were strictly less than the given bound, so that a bound of 10 would yield one of the ten numbers between 0 and 9. This was the behaviour described in the initial specification. However during the implementation, it became clear that this behaviour would make the function unsuitable for generating a 'random non-negative integer', as giving it a bound of `UINT_MAX`[1] would mean that `UINT_MAX` was not in the set of possible return values. Instead it was decided that a bound of n should yield one of the $n + 1$ numbers between 0 and n inclusive.

The function also required some care in its implementation. It is tempting to write a uniform random number generator by getting random numbers from the C library function `rand` until one of them is in the required range. The first problem with this is that the numbers from here have notoriously bad randomness

[1] `UINT_MAX` was `#defined` to be analogous to `INT_MAX` for `unsigned ints` as `~(unsigned int)0`.

properties. Also, in the case where a random bit is required (which comes up often in this project) this will be unacceptably inefficient, as the expected number of integer random numbers before getting a suitable return value is 2^{31}. So we need to try and use as much of the range of the random numbers coming in as possible, though we will not always be able to use all of it when this would lead to non-uniformity in our random numbers, as lower numbers will be favoured if $n + 1$ is not a power of 2. This function therefore works as follows:

1. If the bound n is `UINT_MAX`, get a number from `/dev/urandom` and return it directly.
2. Otherwise, continue by computing the largest number less than `UINT_MAX` which is an integer multiple of $n + 1$.
3. Repeatedly get a random number from `/dev/urandom` until the result is less than our computed maximum.
4. Return the random value mod $(n + 1)$.

Arbitrary Precision Integers

The function `get_random_number` returns a multiple precision integer of length l bits. The GMP library provides a function[2] that returns random integers containing a given number of 'limbs' (where a limb is the same size as an `int`), so this function does the following:

1. Compute the number of limbs required in the return value and call `mpz_random` to get a random integer of that size.
2. If necessary, clear high bits in the return value to make it the correct number of bits long.

3.1.2 Random Prime Generation

This function returns a random number a given number of bits long which has a high probability of being prime. It works as follows:

1. Get a random number of appropriate length and set its top and bottom bits so it will be of the correct length and will be odd.
2. Perform trial division by all primes less than 2000 (which are computed once and stored for future use). If any are found to be factors, start again with a new random number.
3. Use the Miller-Rabin primality testing algorithm (an implementation of which is provided by the GMP library) 25 times, going back to the start to try a new number if it ever fails.

[2]The documentation for this function says that it does not have very good randomness properties. If this project were to be used in a real-world system, a replacement would need to be written using `/dev/urandom` as for primitive `int`s.

According to the Prime Number Theorem, a random odd number of the length we are considering (512 bits) has a probability of around 1 in 177 of being prime.[3] The process of trial division will reject the vast majority of non-primes, so the probability of a number being prime if it has passed this test is far higher than 1 in 177. A number that is not prime but happens to have passed the trial division stage has a probability of 2^{-50} of also passing the Miller-Rabin tests. Given that the expected number of runs before our random number generator picks a real prime is less than 177, we are safe to assume that any number passing this final test is actually prime. Therefore a number coming out of this function will, as required, have a high probability of primality.

3.1.3 Generation of Random Permutations

In the implementation of `player_init` (see Section 3.4.1), a function is needed which generates a random permutation of the numbers 0 to 51, representing a single player's shuffling of the deck. The function `generate_permutation` achieves this by effectively picking a random number from 1 to 52!, and applying a bijective mapping from the set of the possible values of this number to the set of all possible permutations of the numbers 1 to 52. The returned permutation is in the form of an array of 52 integers, where the permutation maps the integer i to the ith integer in the array. It works as follows:

1. Allocate two arrays of 52 integers, called `permutation` and `result`.
2. Fill `result` with values of -1 to indicate that its elements are initially unset.
3. In the first element of `permutation` place a random number from 0 to 51, in the second a number from 0 to 50, and so on, up to the last element which will just be 0. There are 52! ways this step can be done, so these numbers may be interpreted as representing one of the possible random permutations of the numbers 0 to 51.
4. For each n from 0 to 51, count along the `result` array until unset element number `permutation[`n`]` is reached. Set the value of this to n. At each stage this has an equal probability of selecting any of the unset values in the permutation.

3.1.4 Computation of Modular Square Roots

The decryption function described in Section 3.2.5 requires the computation of the square root of a number q modulo another number n whose two prime factors i and j are known. This is achieved by this by computing the square root of q modulo i and j in the function `andos_modular_sqrt`, then combining them in `andos_do_sqrt` using the Chinese Remainder Theorem to get the root modulo n. There is an algorithm for computing modular square roots modulo any prime, but it is rather complex, and it was decided that it would be easier to ensure that the primes were

[3] The Prime Number Theorem states that the number of primes not exceeding x is asymptotically equal to $\frac{x}{\ln(x)}$. Therefore the probability of a large number n being prime is approximately $\frac{1}{\ln(n)}$. For odd numbers, the probability is clearly twice this.

chosen to equal 3 (mod 4), so that a far simpler algorithm could be used. To compute a square root r modulo a prime of this form, it is sufficient to use the formula $r = q^m \pmod{n}$, where $m = \frac{n+1}{4}$.

3.2 Encryption and Decryption of Single Bits

Each of the functions described in this section performs a single stage of the ANDOS protocol on units of data corresponding to single bits of plaintext. The names of the functions and the data structures passed to them as arguments are all of the form `andos_bit_*`. The definitions of the data structures and prototypes of the functions may be seen in Appendix A.1.

3.2.1 Key Generation

In this implementation of the protocol i and j, the primes making up the private half of the key, are chosen to be equal to 3 (mod 4). This makes the computation of square roots easier in `andos_modular_sqrt` (see Section 3.1.4).

Recall from Section 2.6.2 that $n = i \times j$, and that we need to compute a value y, such that y is not a quadratic residue (mod n) but that $J(y, n) = +1$. Now

$$J(y, n) = L(y, i) \times L(y, j)$$

(where $L(a, b)$ is the Legendre symbol of a and b), but $L(y, i)$ and $L(y, j)$ cannot both equal $+1$ (if they did, y would be a residue), so they must both be -1.

Therefore we generate random numbers of the same length as n until we find one that satisfies all the properties we require (that it is less than n and that $L(y, i) = L(y, j) = -1$), and save this as y.

3.2.2 Encryption

The specification says that we require a value e which is a quadratic residue if the bit to be encrypted is 0, and is a non-residue if the bit is 1. In the case where we require a non-residue, the decryption function will in fact rely on e being y multiplied by some quadratic residue. There is therefore an obvious way of generating it:

1. Generate a random number r, where $0 \leq e < n$.
2. Compute r^2.
3. If the bit to be encrypted is 1, multiply by y from the public key.
4. Set e to the remainder (mod n).

The encrypted value is just e, so this is returned in an `andos_bit_encrypted` structure.

3.2.3 Making the Secret Question

Here we need a value q which either has the same quadratic character as e or the opposite character, depending on a secret bit m.

1. Generate a random number r where $0 \leq r < n$, and a random bit m.
2. Set q to $e \times r^2 \times y^m \pmod{n}$.

Note that multiplying by r^2 or y^2 does not change the quadratic character of e, so if $m = 0$ we know that q will have the same character as e, otherwise $q \times y$ will instead, in which case q will have opposite character to e. Therefore if the reader holds on to the value of m until the question is answered, he will be able to use it to decode the answer to the question and determine the original plaintext of the bit.

We return `andos_bit_secret_question`, containing r, m, and q.

3.2.4 Removing Secret Data from the Question

This simply involves taking the value of q out of the `andos_bit_secret_question` and copying it into the `andos_bit_question` which will be sent to the secret originator.

3.2.5 Answering the Question

The answer to the question is b, which is 0 if q was a residue and 1 if it is not. It is accompanied by r, where $r^2 = q$ if $b = 0$, or $r^2 = q \times y$ if $b = 1$.

If q is not a residue, we know that it is the product of an unknown residue d^2 with y, which is not a residue. Therefore $q \times y$ will equal $d^2 \times y^2$, and hence a residue.

1. Set b to the value of $J(q, n)$, which is $+1$ if $L(q, i) = +1$ and $L(q, j) = +1$, and -1 otherwise.
2. Use `andos_do_square_root` (described in Section 3.1.4) to compute the square root of $q \pmod{n}$ if $b = 0$, or the square root of $q \times y \pmod{n}$ otherwise, and set r to the result.

3.2.6 Decrypting the Bit

1. Check the proof of the correctness of the decrypted bit by squaring $r \pmod{n}$ and checking that it equals q if $b = 0$, or $q \times y$ if $b = 1$.
2. The decrypted bit is the exclusive-or of b and m, as an m of 0 left the quadratic character of the question equal to that of the encrypted it, whereas an m of 1 toggled its quadratic character.

3.3 Encryption and Decryption of Multiple Bits

Most of the data structures used through the ANDOS protocol for holding data corresponding to a group of plaintext bits just contain an array of the corresponding structure for single bits and an integer indicating how long this array is. In `andos_data_plaintext`, the single bits making up the plaintext are packed into unsigned integers. The code computes the number of bits that may be stored in an integer on the target architecture using the value of `INT_MAX`. In these, the most significant bit of the binary representation of the `unsigned int` holds the first bit,

the next most significant holds the second, and so on up to the the least significant bit which contains the last.

Each of the data structure initialisation functions allocates space for its array of single-bit data structures, initialising all the multiple-precision integers they contain. Functions were also written that clear and unallocate the memory used by the multiple-bit structures and everything they contain.

Most of the functions that perform the basic operations in ANDOS simply iterate over the elements in the array contained by their input data structure, and write the results into the data structure which is returned. The more interesting ones are those which need to deal with single bits of data which are packed into `unsigned ints`, namely `andos_data_encrypt` and `andos_data_decrypt`. In order to deal with individual bits in these, these functions also iterate over the powers of 2 between 1 and `(unsigned int)INT_MAX+1`, using a bitwise 'or' operation to set bits and a bitwise 'and' operation to read and clear individual bits.

3.4 Getting a Card

The process by which a player gets a card is rather complex, and requires communication with each other player. This is because we require that no single player or third party be able to control which cards go to each player, or see which cards a player has once they have been dealt. Therefore each player must have some secret data that influences the dealing of every card.

3.4.1 Player Initialisation

The function `make_player` sets up the data structures for a new player to be able to play in the card game. It does the following:

1. Allocate memory for the player's public and private data, and set the pointer to all other players' public data.
2. Generate the player's secret permutation, set the initial size of the hand to 0, and initialise the arrays which will hold the public and private data about cards held.
3. For each player whose numeric id is less than that of the current player, generate the τ array of 52 random bit strings, where each bit string is represented as an array of `unsigned ints`.
4. Generate the player's private key.
5. Get the player's public key from his private key.
6. Compute the encrypted π and τ arrays.

3.4.2 Getting a Card

The function `get_a_card` gets a new card from the deck for player n.

1. Get a free card from the permuted deck and add this to the publicly visible hand.

2. For every player whose id is less n, ask each in turn (in the clear) for the element of his permutation which is required to get the real identity of the card (as explained in Section 2.3 and demonstrated in Figure 2.2).

3. Look up the result obtained from player $n-1$ in our own permutation.

4. Use ANDOS to get the required elements of π (and corresponding τ elements) from all players with id $> n$.

Chapter 4

Evaluation

4.1 Testing the Code

A number of different approaches were taken to ensure that each function was properly working before it was used by a higher-layer function which depended on this.

4.1.1 Assertions

It is important to try and detect runtime errors as soon as they occur, so the code which needs to be corrected may be found easily. The C library function `assert` was used for this purpose. For each piece of code that performs a non-trivial operation whose results may be validated or verified in some way, the results are checked within the same function. If they are incorrect, `assert` is used to halt execution of the program immediately, making it easier to work out what problem has occurred, rather than allowing the inevitable 'Segmentation fault' message to be the only initial clue as to what has gone wrong. For example, in `generate_permutation` (see Section 3.1.3) an index into the `result` array is computed, but before this value is used to write to the array it is checked to ensure it has not gone off the end; in `andos_modular_sqrt` (see Section 3.1.4), the computed square root of a number is squared again to check that it really is a root.

Functions also `assert` that their arguments are reasonable, in order to avoid performing calculations on them which are clearly incorrect or impossible. For instance, `andos_bit_encrypt` takes an integer argument for the bit to be encrypted (as C does not have specific boolean type) so it is important to check that the value being encrypted is actually 0 or 1.

4.1.2 Testing Functions

Every time a function was completed, it was tested thoroughly on its own to ensure it worked correctly before code was written that depended on it. Otherwise a bug in a low-level function could cause odd behaviour when a higher-level function that relied on it was called, leading to a far more difficult debugging process. In the case of those like `andos_modular_sqrt` which use `assert` to check their own results, it

was sufficient to run them many times with random data, as running to completion indicates that they are working correctly.

The functions that populated data structures were tested by writing functions to print out the contents of these structures, then checking by eye that each one contained the data that was expected.

Functions involved in the ANDOS encryption and decryption process, both on the level of single bits and on the level of larger chunks of data, were tested by running repeatedly through the ANDOS process of encryption of random data, making the question, converting the question, answering the question, and decryption; the correct working of these functions as a group was easily checked by ensuring the decrypted data was the same as that which went in.

When errors were found that did not have an immediately obvious solution, GDB, the GNU Debugger, was used to find out which line of code had caused the observed effects, to give a clue where the bug might be.

4.2 Work Completed

A lot of effort went into this project, but at first sight the fact that a complete working secure card game was not produced may make it appear that it was not an unmitigated success. As explained in Chapter 2 however, the protocol chosen for implementation in this project is a great deal more demanding than was originally expected, as it required the coding of a nonstandard cryptographic algorithm effectively from scratch, whereas it was expected that libraries would be used for operations on this level. Therefore it seems justifiable that the implementation of some of the higher-level game mechanics should be sacrificed for the more technically challenging work of gaining complete understanding of the workings of a complex cryptographic algorithm. Working code was produced which performed the following operations:

- All steps in the ANDOS probabilistic secret-sharing scheme for single bits.
- Extension of ANDOS to allow sharing of secret data in bitstrings of arbitrary length.
- Initialisation of data structures with all data required to run a secure card game.
- Use of these data structures and communication between players using the ANDOS scheme to deal cards to a player in such a way that no other player may control or work out which card is dealt, and with all auxiliary data that will allow a player to prove that he really has the card held, and has not cheated during this process.

4.3 Sample Output

4.3.1 Encryption of Single Bits

There follows a sample of the output of the test code for the single-bit operation of ANDOS.

```
Performing encryption test.

Generated private key:
pri->i:
0x9777cbfd48036d0d0b461de872aa45c2dc4e4afd3c3bf0cb894247a4e6ffc5f8
  93696b6faddf7c049de6f7ecae5ee6b3ee422592462265afa85a088f0f7d02eb
pri->j:
0xf4defa25874f8a5b379955911794d1dc30f239460a05f05c576332d4540c276c
  7eedc0a7108fc4a4b5b940634bd19d1f2cc3434fc7ba98416d9a5c0609df96fb
pri->n:
0x90e21c517fdcc73ed1a0a4ac859087725da7dc8658c61d43357c213ee4b587cc
  96b09df52d369b8a534ee394391317ca23cf262b7c6c95870134eea8cc57b1b1
  fd3b5c1e81f5cc7ae64306f4139cbbd1c0ff347f9d8a96eba9c34b227c92b666
  80acd8f5e95d98c118179f64274a0a97f0b8ea9bfd4bdea167cc9fd93cfc8e69
pri->y:
0x1d2fa43a785ae560a41aebdd2b76339c630b5ef4633e2f5c6f78f3911581d7b7
  7f802809cbed53f6e8382f15e0ffe8e283931e29e47f323d1fa86c0ae0586568
  9a97094bad364c54866670114e8382f41c2af184014c9e144ab2102771907273
  6e6fca789cd2db5dc2b6f0b9215101a34fe22ab13c404108c39efc3a2bca586c

Entering function andos_bit_encrypt.
secret = 0
r:
0x099ed6705059d4937287447676c6c8c9bcfbfbac872e5c0a326a02161dbd432f
  9f634033585f7c324e4557d744bb1f807928b4f88b5641668cc714824108125b
  a50305780030e4735c7affae475a2d6362d24a243b2fe22563b3f7d059a8942c
  dc27a673e11ad25bc36596fdcfd86fddafc953d0012b9013683cb5dd9e27c95a
result->e:
0x057aec3d51ffebaddab8e750b2416b5b9226e6e4ee67a0efbe8d98641cf2bfed
  6605e395b03c1e9eab8207f8932c45ae15973e7083e05cf1062cd00157dfd1c4
  8fd82d496057aca20ce9efca470c0397737535f9f4c863931196e0689c679694
  2596bd4ceca58a24af78b6f2b09b0b514aa4635e88f53a945aabd6de1f1fd602
Leaving function andos_bit_encrypt.

Entering function andos_bit_make_question.
result->r:
0x6a15fac32ce68ce15894b6ae2d83b750a2d4e2e4da9708cb31f19e6ab721f9cc
  b57719ac4c7a4bd20d65156681bbde519b6e7e06bbeb687a84d4ed2c34069365
  c90d2695892569406105 6d33da2d3d02bc0f576f1f2b0deffd42293f406e24a8
  322b92ee1832ee5c60f283629f1fd3b6657b81db50aa5924641d02f243d52a33
result->m: 0
result->q:
0x2c1dba425e95a69ac43d4864eeaeeced941a4d5f52d75a4bedaf6aee5ace1392
  a373a3d8e8e4fcceedc04e3c55f8ee3747e8f27c8d8046ff9b4d15e86b0b54dd
  4df3d290c2787f10f926b54a67d6e19562e7b8d9d5c4149d908d30a7ed482e2f
  188fd73087fc3546c348b7c63018081c4485dbbe53074389704f4424a89715b4
Leaving function andos_bit_make_question.

Entering function andos_bit_answer_question.
result->b = 0
result->r:
0x054c5a1c740a0275bbd7806bd1d73afb6e5cd07c98421615 05e476f409514058
  9719769f33262615d295d636d26fdf485e1c7e2e1ba8e3200c4cbd5e660d59e0
  7cc0f4ac9e53eb6dba14a598929353785fd3191f02311ef3542cb492293ebffb
  7cd3246b4eb3a8c057c95de5a1f42af5b458e3aa721a7c3616909089592e41fe
Leaving function andos_bit_answer_question.

Entering function andos_bit_decrypt.
Returning decrypted bit of 0
Decoded answer:
a=0
```

4.3.2 Encryption of Data

Clearly the uncut output of this, consisting mainly of multiple copies of that shown in the single bit case, contains huge amounts of data. It has therefore been decided to omit the reams of debug data for this entirely.

4.3.3 Data Structure Initialisation

The following is a trimmed version of the debug output of the initialisation of the data for Player 2, showing the generated private key, the π values in four groups of 13, and part of the two τ values for this player.

```
Displaying info for player 2:
Private key:
p->priv->key->i:
0x88f5c0c5fb6516c5988c44092850035a44ab9abeab2ba2dca690ab25450e6c5c
  42646be4465cfd5b92d854715ed382b816b6cf0a3369198d50d3f855f75cab4f
p->priv->key->j:
0xca04dcb781cee6dd8407c34df6f330d3975d9dc7cbb1c0136d22ad444d82e0d6
  926da434dc9a9cdf9a3661a7ce0068e1d393077967e7e92ad06f96a6d5e486bb
p->priv->key->n:
0x6c1484089a3f2a860b448b6ce133f00f9190d89fad41b468eaed7ccfe6a3a0d4
  900d32cca9d4abe8d249e4c0deb1c7fe0d0dcadb2c96faa8bbe8da348261d5ea
  4fca92ebe1423dbe58325f59730d13ddb1b95e0222a69e1d380a468ed67c1bb5
  4b84fa985b348f8b87d5e82b8e9fedaec5950f0754e0f1f4dbec2e917fb87cb5
p->priv->key->y:
0x54617804e4b0905f08e6bfeeb9cbe5be005849cee117dbe59d415da6d35aaccf
  47b1fa08fd4906e5203dfd5bce7461cfc2479e7709d84635e20d1c611fea4f0a
  207c4d415d8c884ae8411f2eefb6b2f52ce5082c09017e0940ab8c5e2a0b52e4
  404ac5c1de6bbb6d70cd03beffec960ba73dc9d24822dade2c58a2aa07a33cce
pi:
[ 18 36 25 13 44 37 40 09 21 42 12 49 20 ]
[ 17 11 35 26 10 47 32 39 04 19 38 30 41 ]
[ 22 07 14 01 06 31 15 27 34 05 33 16 46 ]
[ 02 24 28 51 43 08 45 50 48 23 29 03 00 ]
There are 2 values for tau:
  tau 2,0 is as follows:
    Entry  0: 10100011001111110110110000000100101000011110111010011000000111110
    Entry  1: 01000001111111001001111010110111010111111110011001001001111111110
<snip>
    Entry 50: 01100100101001100110100110001111110110100101010000011001101111011
    Entry 51: 11000011110110001110111110000011100110001001000100010001100000010
  tau 2,1 is as follows:
    Entry  0: 11010111011110100110000011010101110010111000010001100000001001001
    Entry  1: 11000001110111111101110100101000111000010011100010010000010011010
<snip>
    Entry 50: 11010010000100110111001010010010010110010011111110010101001101000
    Entry 51: 00001100001011000100100000000001111111111001010100100100000100011
Hand contains 0 cards.
```

4.3.4 Card Dealing

The culmination of the project, the following shows the dealing of cards to players utilising all of the above mechanisms. For brevity, a single card is dealt to each of four players.

```
Getting a card for player 0
Chose element from the permuted deck: 40
Got my own pi element directly: 38
Got encrypted pi element from player 1: 22
Got encrypted pi element from player 2: 46
Got encrypted pi element from player 3: 9

Displaying info for player 0:
Hand contains 1 card.
Card 0 has public value 40 and private value 9.
  Card is Ten of Clubs.


Getting a card for player 1
Chose element from the permuted deck: 46
```

```
Got unencrypted pi element from player 0: 25
Got my own pi element directly: 0
Got encrypted pi element from player 2: 24
Got encrypted pi element from player 3: 1

Displaying info for player 1:
Hand contains 1 card.
Card 0 has public value 46 and private value 1.
  Card is Two of Clubs.


Getting a card for player 2
Chose element from the permuted deck: 16
Got unencrypted pi element from player 0: 7
Got unencrypted pi element from player 1: 44
Got my own pi element directly: 25
Got encrypted pi element from player 3: 11

Displaying info for player 2:
Hand contains 1 card.
Card 0 has public value 16 and private value 11.
  Card is Queen of Clubs.


Getting a card for player 3
Chose element from the permuted deck: 17
Got unencrypted pi element from player 0: 5
Got unencrypted pi element from player 1: 29
Got unencrypted pi element from player 2: 31
Got my own pi element directly: 51

Displaying info for player 3:
Hand contains 1 card.
Card 0 has public value 17 and private value 51.
  Card is King of Spades.
```

Chapter 5

Conclusion

5.1 Work Completed

As has been made clear earlier on in this dissertation, the aims of the project grew significantly from those which were originally stated in the proposal, making it into a far more interesting and challenging project.

The additional work which was done, the implementation of the All-or-Nothing-Disclosure-Of-Secrets probabilistic encryption scheme, required the understanding of new areas of mathematics. Its code needed required a great deal of care, as in encryption code a trivial error usually leads to completely incorrect output.

On top of this, code was written to allow cards to be dealt among a group of players in a secure manner, with all additional data required for these players to prove that they have not been cheating.

5.2 Possible Extensions

The most obvious addition to the project would be code that allows players to prove they really hold the cards they claim to have, and to prove they have not cheated by reading permutation elements corresponding to other players' cards. Specifically, one problem is that at the moment there are no mechanisms in place to prevent a player cheating by asking for one permutation element and a τ value corresponding to a different element of the permutation. The original ANDOS paper[1] describes a way of proving that a group of functions is 'fair', meaning that each question corresponds to the same secret; in fact it is this proof that gives the secret-sharing scheme its 'All-or-Nothing' property. This basically involves the reader of a secret generating many groups of questions for each multiple-bit secret, each of which has had a different random permutation applied to it, which is kept secret. The originator of the secret then chooses some set of these groups of questions, and asks the reader to reveal all secret data which was used to prepare them. Due to details of their construction which will not be described at length here, this allows the originator to have a high probability of detecting attempted cheating by the reader, without enabling him to discover which secret the reader is attempting to read. A similar process would allow players to prove the validity of their permutations.

No changes to the current code would be required if one decided to add proofs like this, as the data structures already contain all the required data. However the proofs involve a significant amount of mathematics which would take a lot more effort to understand and implement correctly.

As explained in the Introduction, the whole motivation for this project was to allow games of cards to be played over the Internet. However the code as currently written does not actually allow this, as the data which is supposedly moving between players at opposite ends of the planet is just going through functions which simulate these players' behaviour, in a single process running on a single machine. An obvious extension to the project is therefore to add networking code which would allow players to send this data between different machines. As each secret-reading from one player to another requires three messages to be sent over the network, each of which containing a 1024-bit number for each bit that is being read, and in some circumstances a player needs to perform this protocol with each other player in the game in order to get a single card, we can see that a large amount of data will pass over the network in the course of a complete game. This is a consequence of the fact that we needed a probabilistic scheme with the properties discussed above, so really is inevitable.

Another piece of work that would be interesting to add to the project a formal proof that it satisfies all the required security properties. If the project were to be used in the real world, this would allow the players to have a certain level of trust in the resistance of the game to attack, which is necessary in an environment such as a Casino where money is changing hands based on the outcome of the game.

5.3 Looking Back to the Introduction

In Section 1.2, a list of possible attacks on a supposedly secure card game was presented.

If one used the framework which has been implemented in this project for playing a game such as poker, we can see that most of the attacks suggested by Schindelhauer[4] are dealt with relatively well:

Sabotage After the initial deal of cards, no single player may do anything to get in the way of the remainder of the game protocol. If a player decides to stop cooperating with the remaining players, the only thing that might happen is that he will be unable to blow the whistle on another player who has cheated by reading incorrect permutation elements from him.

Protocol attack If a player's permutation is not random this can only be his loss, as if all the other players decide to join forces against him they will be able to work out the identity of all the cards in the deck, as his randomising influence is no longer present. The one risk is that a player will decide to change his permutation so that some cards from the deck are 'edited out' and replaced with duplicates of others. Crepeau's paper describing the protocol[3] deals with this with some additional proofs that must be done by each player, and this is one of the things that would need to be added in a real-world system.

Cheating We must assume that a Poker player is not able to fool the other players into thinking his hand has some value it does not. Therefore this is not really a risk.

Secret coalitions Our scheme ensures that no single player ever has any secret information about another player, therefore no amount of sharing of secret data can help a group of players determine the cards held by another player.

Public coalitions and ghost players As noted in the introduction, these are both social issues that cannot be solved by a game-playing protocol.

At the end of the project, the question that needs to be asked is, what scope is there for people in the real world to use systems like this one? Is the lack of a trusted third-party actually a good thing, or even achievable? In the example of the Casino discussed in the Introduction, it is likely that in practice the Casino would not be prepared to release its control of the game In any case, the player needs to have a certain level of trust in the Casino, as most betting systems rely on the player's credit card being debited or credited by the Casino at the end of the game. One is therefore led to wonder if there is any point in deliberately removing trust from the Casino, as ultimately it is in control of the better's money. Coupled with the fact that Casinos are unlikely to agree to surrender control over the game, we are led to the conclusion that this kind of distributed game is unlikely to be used in this situation.

The other use proposed for a game like this was among a group of equals who want to play an informal game over a network. Currently people usually use a trusted server such as Yahoo! Games to play this kind of game, but a system like that implemented in this project might well be preferable to this, for example if the parties are not connected to the global Internet, or if they want more control over the user interface for their game than is provided by such systems.

Therefore I conclude that this project has been a useful demonstration of the facilities that probabilistic cryptography provides for use in the everyday lives of normal people.

Bibliography

[1] Gilles Brassard, Claude Crepeau, and Jean-Marc Robert. All-or-nothing disclosure of secrets. In *Advances in Cryptology: Proceedings of CRYPTO*, pages 234–238, http://citeseer.nj.nec.com/brassard87allor.html, 1986.

[2] Claude Crepeau. A secure poker protocol that minimizes the effect of player coalitions. In *Advances in Cryptology: Proceedings of CRYPTO*, pages 73–86, http://citeseer.nj.nec.com/crepeau86secure.html, 1985.

[3] Claude Crepeau. A zero-knowledge poker protocol that achieves confidentiality of the players' strategy or how to achieve an electronic poker face. In *Advances in Cryptology: Proceedings of CRYPTO*, pages 239–247, http://citeseer.nj.nec.com/crepeau86zeroknowledge.html, 1986.

[4] Christian Schindelhauer. A toolbox for mental card games.

Appendix A

Selected header files

A.1 andos.h

```
/* Header file for andos.c */

#ifndef ANDOS_H
#define ANDOS_H

#include <gmp.h>
#include "utils.h"

/* Length of each prime in bits will be half this */
#define ANDOS_KEY_LENGTH 1024

typedef struct {
    mpz_t i, j;
    mpz_t n;
    mpz_t y;
} andos_privkey;

typedef struct {
    mpz_t n;
    mpz_t y;
} andos_pubkey;

typedef struct {
    int m;
    mpz_t q;
} andos_bit_secret_question;

typedef struct {
    mpz_t q;
} andos_bit_question;

typedef struct {
    mpz_t e;
} andos_bit_encrypted;

typedef struct {
    mpz_t r;
    int b;
} andos_bit_answer;

/* In all the following data structures, length refers to the number of elements
 * in the array, whether they correspond to bits of data or unsigned-ints-full
 */

typedef struct {
    andos_bit_encrypted *data;
```

```
    int length;
} andos_data_encrypted;

typedef struct {
    andos_bit_secret_question *data;
    int length;
} andos_data_secret_question;

typedef struct {
    andos_bit_question *data;
    int length;
} andos_data_question;

typedef struct {
    andos_bit_answer *data;
    int length;
} andos_data_answer;

typedef struct {
    unsigned int *data;
    int length;
} andos_data_plaintext;

/* Initialisation functions for andos */
andos_privkey *andos_prepare();

andos_pubkey *andos_get_pubkey(andos_privkey *);

/*
 * In all of these, the int argument stores the length of the data being
 * stored in *UNSIGNED INTS*.
 */
void andos_init_data_encrypted(andos_data_encrypted *, int);
void andos_init_data_secret_question(andos_data_secret_question *, int);
void andos_init_data_question(andos_data_question *, int);
void andos_init_data_answer(andos_data_answer *, int);
void andos_init_data_plaintext(andos_data_plaintext *, int);

/* Functions to clear out data structures */
void andos_clear_data_encrypted(andos_data_encrypted *);
void andos_clear_data_secret_question(andos_data_secret_question *);
void andos_clear_data_question(andos_data_question *);
void andos_clear_data_answer(andos_data_answer *);
void andos_clear_data_plaintext(andos_data_plaintext *);

/* Functions to print out the contents of andos data structures */
void output_data_plaintext(const andos_data_plaintext *);
void output_data_encrypted(const andos_data_encrypted *);
void output_data_secret_question(const andos_data_secret_question *);
void output_data_question(const andos_data_question *);
void output_data_answer(const andos_data_answer *);

/* Functions that perform andos on single bits */
void andos_bit_encrypt(andos_bit_encrypted *, andos_privkey *, int);

void andos_bit_make_secret_question(andos_bit_secret_question *,
                                    andos_pubkey *,
                                    andos_bit_encrypted *);

void andos_bit_convert_question(andos_bit_question *,
                                andos_bit_secret_question *);

void andos_bit_answer_question(andos_bit_answer *,
                               andos_privkey *,
                               andos_bit_question *);

int andos_bit_decrypt(andos_bit_secret_question *,
                      andos_bit_answer *);

/* Functions that perform andos on larger bodies of data */
```

```
void andos_data_encrypt(andos_data_encrypted *,
                        andos_privkey *,
                        andos_data_plaintext *);

void andos_data_make_secret_question(andos_data_secret_question *,
                                     andos_pubkey *,
                                     andos_data_encrypted *);

void andos_data_convert_question(andos_data_question *,
                                 andos_data_secret_question *);

void andos_data_answer_question(andos_data_answer *,
                                andos_privkey *,
                                andos_data_question *);

void andos_data_decrypt(andos_data_plaintext *,
                        andos_data_secret_question *,
                        andos_data_answer *);

/* Modular square root functions */
void andos_do_sqrt(mpz_t, mpz_t, mpz_t, mpz_t);

void andos_modular_sqrt(mpz_t, mpz_t, mpz_t);
#endif
```

A.2 player.h

```
/*
 * player.h :  Contains the definition of the player data structure.
 */

#ifndef PLAYER_H
#define PLAYER_H

#include <gmp.h>
#include "andos.h"

/* The number of unsigned ints used for storing each value of tau. */
#define SECURITY_PARAMETER 2

typedef struct {
    int *pi; /* This player's permutation */

    /* A 2D array of bitstrings of dimension [id][52], where a bitstring is
     * represented as an array of ints, with the MSW being earliest. */
    unsigned int ***tau;

    /* Private data about hand (ie which cards from the real deck are held) */
    int *hand;

    andos_privkey *key;
} player_private;

typedef struct {
    int id; /* Player ID, starting at 0 */
    andos_pubkey *key;

    /* Array of 52 encrypted permutation elements */
    andos_data_encrypted *pi_enc;

    /* A 2D array (dimension [id][52]) of encrypted tau bitstrings */
    andos_data_encrypted **tau_enc;

    /* Number of cards in the hand */
    int hand_size;

    /* Public data about hand (ie which cards from DECK are held) */
    int *hand;
} player_public;

typedef struct {
    player_private *priv;       // This player's private data
    player_public *pub;         // This player's public data
    player_public *all_players; // Array of all players' public data
} player;

void make_player(int);

player *init_players(int);

void get_a_card(player *p);

/* The next few methods are to be called by the routines which get a new card
 * for a player.  The intention is that they could be replaced by pairs of
 * functions which communicate over a network, the callee's end of which
 * checking that the correct number of requests for encrypted data and data
 * decryption occur. */
int get_pi_element(player_public *, player_public*, int);
andos_data_encrypted *get_encrypted_pi_element(player_public *,
                                               player_public *,
                                               int);
andos_data_answer *get_decrypted_pi_element(player_public *,
                                            andos_data_question *);
void display_player_info(player *);
#endif
```

CST Part II Project Proposal

A Secure Distributed Card Game

M. A. Pinna, Gonville and Caius College

Originator: M. Pinna

19 October 2001

Special Resources Required

The use of my own IBM PC (1.4GHz Athlon, 512Mb RAM and 40Gb Disk).

Project Supervisors: Dr G. Titmus & Dr I. W. Jackson

Directors of Studies: Dr G. Titmus & Dr P. Robinson

Project Overseers: Dr F. H. King & Dr A. M. Pitts

Introduction

The proposed project is an implementation of a variation of the 'mental poker' protocol described in section 4.11 of Schneier's Applied Cryptography. A public-key cryptographic protocol will be designed that allows a card game to be played over a network in a secure manner, with no need for a central trusted server. The protocol should ensure, for example, that no player may unfairly influence or discover which cards any player is dealt, and that no player may play a card that he has not been dealt legitimately.

Some types of cheating will only be detectable by making each player (or perhaps just the winner) disclose his private key at the end of the game, but the system should prevent or detect as much attempted cheating as possible immediately.

The intention is that the system will be flexible enough to allow a variety of card games to be run on top of an underlying framework that deals with the cryptographic protocols. Adding a new game to the system should simply be a matter of specifying its rules and allowing the generic lower layer to do the hard work.

As an optional extension, formal proofs of the security of the system may be attempted. These will involve showing that no player is able to control any part of the dealing process or find out what cards other players have been dealt, and that each player is independently able to verify that a card played by another player was actually held by that player.

Work that has to be done

The project breaks down into the following main sections:

1. Design of the cryptographic protocols to be used in the system.
2. Implementation and testing of the layer which will allow general card game operations to be performed (deal card, play card etc), using a freely available cryptographic API.
3. Implementation and testing of a layer working on top of the above, which allows one or more particular games to be played.

Difficulties to Overcome

The following main learning tasks will have to be undertaken before the project can be started:

- To gain familiarity with the tools to be used for the project (C under Linux, and whichever cryptographic API is chosen).

Starting Point

The basic cryptographic operations of encryption, decryption, signing and verification will be performed by a publicly available cryptographic library, and will therefore not need to be re-implemented as part of this project.

Resources

I intend to use my own PC for coding and testing of the project. If this breaks down I will use PWF linux, where my quota of 25Mb ought to be sufficient.

Backups will be performed automatically to the University's backup service, Pelican, and to at least one other machine I have access to.

Work Plan

The work will be split up into ten work units, as follows:

Work Unit 1

Deadline: 9 November 2001

- Background reading - relevant research papers, previous work that has been done in the field.

Deliverables:

- Written summary of relevant and interesting ideas that have been found.

Work Unit 2

Deadline: 23 November 2001

- Preparation of a more detailed timetable for the coding phase of the project.
- Design of the protocols to be used in the system.

Deliverables:

- Detailed timetable.
- Formal description of the protocols.

Work Unit 3

Deadline: 7 December 2001

- Familiarisation with the tools to be used in the project (C under Linux).
- Design of the protocols to be used in the system.

Deliverables:

- A firm decision on which libraries are to be used in the project.

Work Unit 4

Deadline: 21 December 2001

- Architecture of the system.

Deliverables:

- A description of the modules making up the system, their purposes and functions, and the relationships between them.

Work Unit 5

Deadline: 11 January 2002

- Implementation of the layers providing basic card-game operations (shuffle deck, deal card etc).

Deliverables:

- A functional but unpolished implementation of the lower layers.

Work Unit 6

Deadline: 25 January 2002

- Testing of above layer and bug fixes.
- Progress report.
- (slack time)

Deliverables:

- A fully debugged implementation of the lower layers.
- The progress report.

Work Unit 7

Deadline: 15 February 2002

- Implementation of one or more card game(s) on top of the above layer, including a simple user interface (a more sophisticated interface may be added, time permitting).

Deliverables:

- A complete but undebugged implementation of at least one card game.

Work Unit 8

Deadline: 8 March 2002

- Testing of above layer and bug fixes.
- Attempt formal proofs of the security of the system (time permitting).

Deliverables:

- A fully working implementation of a secure card game.

Work Unit 9

Deadline: 12 April 2002

- Evaluation of the project.
- Initial draft of dissertation.

Deliverables:

- A full-length draft of the dissertation.

Work Unit 10

Deadline: 3 May 2002

- Polishing of dissertation.
- (slack time)

Deliverables:

- The finished dissertation.