Master Project J2ME version of the Tuple Board
Travis Thomas
April 17 2009
committee:
Professor Alan Kaminsky
Professor Minseok Kwon
Professor Edith Hemaspaandra
# Table of Contents

1. Overview ........................................................................................................................................... 3

2 Tuple Board Background .......................................................................................................................... 3
   2.1 Tuple Board ..................................................................................................................................... 3
   2.2 Java 2 Micro Edition ......................................................................................................................... 4
   2.3 Sony Ericsson 990i ............................................................................................................................ 4

3 J2ME Tuple Board ................................................................................................................................... 5
   3.1 Design ............................................................................................................................................. 5
   3.2 Porting the J2SE Tuple Board ............................................................................................................. 11
   3.3 Using the J2ME Tuple Board ............................................................................................................. 12

4 Secure Card Deck .................................................................................................................................. 14
   4.1 Card Deck Encryption and Decryption ............................................................................................... 14
   4.2 Card Deck Tuples ............................................................................................................................... 15
   4.3 Security Threats ................................................................................................................................. 18

5 Game of War .......................................................................................................................................... 19
   5.1 Description of War Game .................................................................................................................. 19
   5.2 User’s Manual For Tuple Board Version ............................................................................................... 19
   5.3 Tuple interactions ............................................................................................................................... 21
   5.4 Software Design ................................................................................................................................. 26

6 Game of Texas hold'em .......................................................................................................................... 29
   6.1 Description of the game ..................................................................................................................... 29
   6.2 User’s Manual for Tuple Board Version .............................................................................................. 29
   6.3 Tuple interactions ............................................................................................................................... 32
   6.4 Software Design ................................................................................................................................. 41

7 Performance Testing ............................................................................................................................... 44
   7.1 War Game Performance ...................................................................................................................... 44
   7.2 Texas hold 'em game performance .................................................................................................... 44

8 Future Work .......................................................................................................................................... 45

9 Conclusion ............................................................................................................................................ 45

Appendix A. Developers Manual ............................................................................................................... 47
   A.1 Development Environment Setup .................................................................................................... 47
   A.2 Installing Programs on phone ............................................................................................................ 47
   A.3 Building the Card Game Suite ......................................................................................................... 49

Sources ..................................................................................................................................................... 53
1. Overview

The tuple board library was originally developed by Professor Alan Kaminsky to create programs that can be used in an ad-hoc networking environment. The library consists of code that allows a developer to easily communicate with other devices that are on the ad-hoc network.

However, it can only be run on devices that are capable of supporting Java 2 Standard Edition (J2SE). Currently, there is a growing number of devices that can run the Java 2 Micro Edition (J2ME). These devices have capabilities that will allow them to take part in an unstructured or ad hoc network.

This project will create a J2ME version of the tuple board library. This project will focus on three areas:

1. Whether or not the tuple board can be ported to Java 2 Micro Edition
2. How card games could be developed to run using the tuple board library
3. How someone might be able to securely deal a deck of cards in an ad-hoc environment

Card games were chosen as programs to build on the J2ME version of the tuple board in order to test its capabilities. One reason for this is many cell phones have already proven that card games can be run on them. All of these card games so far are missing the one element that could make them great: other real people. With the tuple board version of these card games, people won't have to carry around a deck of cards anymore. They will simply have to take their cell phone out of their pocket and start playing with their friends.

In order to do this two card games were developed for this project; the first is War and the other is Texas hold 'em. A secure card deck was also developed in an attempt to investigate how cards could be dealt securely in an ad-hoc environment. The idea is that it will be very similar to a real deck of cards, where no one will be able to know what a card is unless they are supposed to. The deck is able to be shuffled as well.

2 Tuple Board Background

Tuple space was an idea first developed by David Gelernter. A tuple is an object that contains information that needs to be shared with other processes. The concept of tuple space is the idea of posting “tuples” into a globally shared space in which all other participating computers are able to post and withdraw tuples. This is usually implemented using a client server architecture. A server will store all the tuples for a distributed application and clients will contact the server in order to perform tuple operations. In tuple space the following operations are used to make the distributed systems work: write, take, read, and notify. When a tuple is written, it is placed in the tuple space allowing all other devices to access it. All other devices are then able to perform operations on the tuple by using the tuple space to gain access to it. When a tuple is read; the device can see what values the tuple holds and make decisions based on what it sees. A tuple can also be taken. The device taking can read the contents of that tuple, and the tuple is removed from the tuple space. No other devices can use it anymore. Notify updates interested devices about tuples that have been added or removed from the tuple space.

2.1 Tuple Board

The tuple board library is a distributed computing paradigm for developing applications that run on mobile networking devices that are communicating using an Ad hoc network. The library was developed by Alan Kaminsky and extends the tuple space concept. It allows for the creation of programs for ad hoc networks that can use tuples to share information without the use of a central server required by tuple space. In order for the tuple board library to work in an ad hoc network, the functions that can be performed on tuples are limited. This makes the tuple board more forgiving than the tuple space if there is a network failure. The tuple board's basic operations are called post, withdraw, read and notify. The post function is very similar to write in that it writes a tuple and other
devices would then be able to see this new tuple. Withdraw is similar to take except only the device that posts a tuple is able to remove it from its tuple board. Read and notify perform the same function.

Tuple board unlike tuple space works like a display case. Only the person with the key to their devices display case (board) is able to make changes to what they are displaying. If another device gets near to the board they are able to read the board and make decisions based on what is posted, but since they don't own the display case they are not allowed to make changes directly to it.

Previous masters projects in ad hoc networking have led to an idea for a paradigm called “actual state/desired state.” Each device post its own state at all times. If a device would like another device to change state it simply publishes a new state that it wishes the state of application to go to. this paradigm was discovered while students were doing work using the Many to Many Invocation. The Many to Many Invocation was a library that preceded the tuple board for developing ad hoc programs. The tuple board was designed with the idea of taking advantage of this new paradigm for developing ad hoc applications.

2.2 Java 2 Micro Edition

Java 2 Micro Edition is similar to Java Standard Edition level 1.3. The Connected Device Configuration (CDC) contains many of the classes that are contained in Standard Edition excluding GUI support. CDC's foundation profile adds important functionality to CDC such as sockets required by the tuple board for communication. CDC's personal profile adds AWT GUI support. Some key elements that were changed for it to run on J2ME were generics. J2ME doesn't contain generics. Inorder to use the tuple board on J2ME one of the key elements that had to be changed was generics as J2ME does not contain generics. “For each” loops are also not present in J2ME and were replaced.

2.3 Sony Ericsson 990i

The Sony Ericsson 990i is a smart phone with features that allow for porting the tuple board to run on it. The 990i has the ability to run programs that have been developed using the Java language and includes the Java 2 Micro Edition(J2ME). The J2ME version installed on the phone includes packages that the tuple board requires to function, such as java.net and java.lang.reflect. The java.net package is required for communication to other devices. The java.lang.reflect package is required for the tuple board to work properly.

The phone has many interfaces for communication. Wi-Fi is available and can be used by Java programs that have been written for the phone. This feature allows the phone to connect to a wireless router and to the Internet. It also allow the phone to use the tuple board to communicate with other computers and phones. For phone communication, the phone is compatible with the Global System for Mobile Communication (GSM) type of cell phone networks.

The phone comes with sixty megabytes of RAM that can be used by various user applications. It also comes with a built in hard drive that can store sixty megabytes of user data. The CPU is an Advanced Rish Machine (ARM) type and is two hundred and eight megahertz.

The phone uses the Symbian 9.1 operating system. Which is designed for mobile phones. This phone also uses UIQ 3.0. UIQ allows developers to create graphically feature rich programs.
3 J2ME Tuple Board

Figure 1 shows the overall system architecture of the card games suite that has been developed for this project. The secure deck relies upon the tuple board to communicate and provides a secure deck of cards to a card game. The card games communicate with other instances of the same card game using the tuple board. Card games create and use a secure deck to shuffle and deal the cards and later to decrypt the cards. All of the individual pictures in the above figure run on a device with the J2ME version of Java. More details about the individual software units can be found in other sections of this document.

3.1 Design

The design of the J2ME version of the tuple board is very similar to the J2SE version. One of the main differences is the TupleBoardProxy class uses the Socket class instead of a Channel class from the std library. Another change is that a custom ByteBuffer class had to be written for the J2ME version instead of using the java.nio ByteBuffer. This is because that package is not available on the J2ME platform.

The main class of the J2ME tuple board is the TupleBoard class shown in Figure 2. This class creates the tuple board so that the creating program can use it to communicate to other programs that have also created a tuple board.
The TupleBoard class creates many other classes to help it perform the necessary communication functions. Most of the classes that have significant roles are detailed in this section. For classes that are not detailed here information can be found by looking at source 4 (Tuple Board Documentation) since the J2SE version is very similar to the J2ME version. When a program wants to get information from the TupleBoard class the TupleBoard class will create a RequestHandler. One type of request handler is shown in Figure 3. All request handlers contain the same functions.

**Figure 2: TupleBoard class.**

**Figure 3: PostWithdrawRequestHandler class.**
The RequestHandler is used by a class called the TupleBoardImpl (Figure 4) which gets created and used by the TupleBoard class. This class keeps track of all the tuples that have been posted to the tuple board and checks them against requests to return the proper tuples to programs.

The TupleBoardImpl class relies on a class called the TupleBoardMap (Figure 5). The TupleBoardMap keeps track of all the other tuple boards and a TupleBoardProxy (discussed later in this section) for communication.

**Figure 4: TupleBoardImpl class.**

The RequestHandler is used by a class called the TupleBoardImpl (Figure 4) which gets created and used by the TupleBoard class. This class keeps track of all the tuples that have been posted to the tuple board and checks them against requests to return the proper tuples to programs.

The TupleBoardImpl class relies on a class called the TupleBoardMap (Figure 5). The TupleBoardMap keeps track of all the other tuple boards and a TupleBoardProxy (discussed later in this section) for communication.

**Figure 5: TupleBoardMap class.**
The TupleBoardImpl also uses the classes TupleMap (Figure 6) and RequestMap (Figure 7). The TupleMap class keeps track of all the posted tuples for the TupleBoardImpl class. The RequestMap class keeps track of the active RequestHandlers.

<table>
<thead>
<tr>
<th>TupleMap</th>
</tr>
</thead>
<tbody>
<tr>
<td>+close(theCaller:TupleBoardRef)</td>
</tr>
<tr>
<td>+getPostedTupleNumbers(): IntSet</td>
</tr>
<tr>
<td>+getTuple(theTupleNumber:int): Info</td>
</tr>
<tr>
<td>+iterator(): Iterator</td>
</tr>
<tr>
<td>+post(theTuple:Tuple, theSecurityContext:SecurityContext): Info</td>
</tr>
<tr>
<td>+reportWithdrawn(theCaller:TupleBoardRef, theSecurityContext:SecurityContext, theRequestNumber:int, theTupleNumber:int)</td>
</tr>
<tr>
<td>+withdraw(theTuple:Tuple)</td>
</tr>
</tbody>
</table>

*Figure 6: TupleMap class.*

<table>
<thead>
<tr>
<th>RequestMap</th>
</tr>
</thead>
<tbody>
<tr>
<td>+addRequestHandler(theRequestHandler:RequestHandler)</td>
</tr>
<tr>
<td>+close(theCaller:TupleBoardRef)</td>
</tr>
<tr>
<td>+getNextRequestNumber(): int</td>
</tr>
<tr>
<td>+getRequestHandler(theTupleBoard:TupleBoardRef, theRequestNumber:int): RequestHandler</td>
</tr>
<tr>
<td>+iterator(): Iterator</td>
</tr>
<tr>
<td>+iterator(theTupleBoard:TupleBoardRef): Iterator</td>
</tr>
<tr>
<td>+removeRequestHandler(theRequestHandler:RequestHandler)</td>
</tr>
<tr>
<td>+removeRequestHandler(theTupleBoard:TupleBoardRef, theRequestNumber:int)</td>
</tr>
</tbody>
</table>

*Figure 7: RequestMap class.*

The TupleBoardImpl uses TupleBoardProxy (Figure 8) objects to communicate with other TupleBoards running in separate processes. These TupleBoardProxies are stored in the TupleBoardMap. The TupleBoardProxy in the J2ME version of the tuple board uses a Transfer Control Protocol (TCP) Socket to maintain its connection to the other tuple boards. The J2SE version used java.nio.Channels instead.
There are two classes that allow the tuple boards to find the other tuple boards. These classes are the Announcer class (Figure 9) and the Acceptor (Figure 10) class. The Annoucer class uses a User Datagram Protocol (UDP) packets to announce its BoardId, IP address and port number to other tuple boards. These UDP packets are sent to a multicast address. When a tuple board receives one of these announcements, it will do one of two things. Either it will see it already knows about this tuple board and do nothing or it will see that it does not know about this tuple board. If it does not know about this tuple board and the tuple board has a lower id it will create a TupleBoardProxy with a TCP Socket setup with the information sent in the UDP packet. If the board has a higher board id it will wait for the lower board id tuple board to make the TupleBoardProxy. All non announcement communication will take place using the TupleBoardProxy.
The Acceptor class waits for an Announcer class to setup a TCP connection to it. When a TCP connection is established it will create a TupleBoardProxy for all further communication between the two tuple boards.

The TupleBoardProxy uses input and output streams that have been developed for the tuple board. If a tuple board application is using the DefaultSecurityContext it will use a ChunkedInputStream class and a ChunkedOutputStream class. The Chuck streams allow data to be written in chunks. If a tuple board is using the FixedHelixSecurityContext, it will use HelixInputStream class and HelixOutputStream class. The Helix streams allow the data to be written in a series of chunks using the Helix stream cipher (see source Helix Fast Encryption and Authentication in a Single Cryptographic Primitive.). Both the DefaultSecurityContext class and the FixedHelixSecurityContext class extend the SecurityContext (Figure 11) class and override the getInputStream and getOutputStream functions.

Both input streams rely on the java.nio ByteBuffer class. This class had to be created for the J2ME version. The implementation of the ByteBuffer class used by the J2ME version of the tuple board is shown in Figure 12. Of the original ByteBuffer that can be found in the java.nio package of J2SE only the functions that were used by the tuple board in the original ByteBuffer were implemented.
3.2 Porting the J2SE Tuple Board

The original tuple board that was developed by Professor Kaminsky and his students was written using Java 1.5 constructs. The J2ME version that is available on the Sony Ericsson 990 is similar to Java Standard Edition 1.3. The standard library is much smaller then the one used to make the J2SE(1.5) version of the tuple board. The J2ME version of Java also does not support generics.

The general process for porting the tuple board was to start by porting the parts of the code that didn't require the network. The first step was to add files to the project one at a time and remove the code that wasn't compatible with the J2ME version of the tuple board. After the non networked parts were working, the classes that detect other tuple boards were added. Finally, the actual network portion of the tuple board was ported.

In order to port generics it was necessary to take any collection that was just one type, converting it to a collection that could hold any type of object, and then put in the correct cast in the code when it was needed. The lack of generics makes it more difficult to take advantage of the Java Collections classes.

Another area that had to changed were “for each” loops. These also did not exist in the J2ME version of Java. The below code is an example of changing a “for each” loop. The original code:

```java
for (Field field : myTupleClass.getFields()) {
```

was replaced by something like this:

```java
for (Field field : myTupleClass.getFields()) {
```
for(int i = 0; i < myTupleClass.getFields().length; i++){
    Field field = myTupleClass.getFields()[i];
}

It was important to make sure the loop was able to get at the same data and move on from there.

Other code changes involved changing a particular class that was not available to use for developing applications on the phone. For example, an IdentityHashMap was used in one of the classes. In this case it was determined that a linked list and a loop could replace its functionality.

The original tuple board relied on the java.nio package for its networking communication. This package was not available for development using the J2ME phone. All of the Channel calls had to be replaced by Sockets and input and output streams.

Another class that was in the nio package that was used heavily was the ByteBuffer class. Since no class in the J2ME library could replace the functionality of this class one was written, with only the specific functions that were required to get the tuple board working. The real nio ByteBuffer class had the ability to read in both endianess orders by flipping a flag. The class that written requires the programmer to call a specific function to get and put the data in little endian order.

3.3 Using the J2ME Tuple Board

Developing applications for the J2ME version of the tuple board is very similar to developing applications with the J2SE version of the tuple board. In this section of the document examples will be shown of how to perform common tasks in the J2ME version of the tuple board and how they differ from how things are done in the J2SE version of the tuple board. More details about tuple board usage can be found at "http://www.cs.rit.edu/~ark/tb/doc/index.html" (Tuple Board Documentation).

The post operation is done in exactly the same way in both versions of the tuple board. The tuple board post operation can be done by doing the following:

```
PlayerTuple plyrTuple = new PlayerTuple("name", cardTable.getBoardId());
this.cardTable.post(plyrTuple);
```

PlayerTuple is an object that inherits from the base Tuple class. The cardTable object is a TupleBoard object. This operation is done exactly the same in the J2ME and J2SE versions of the tuple board.

The read operation is an area where slight differences exist between the J2SE version and the J2ME versions. In the J2ME version the following code is required to complete a read operation:

```
PlayerTuple template = new PlayerTuple();
Tuple tuple = this.cardTable.read(template);
PlayerTuple read = (PlayerTuple) tuple;
```

In the J2ME version of the tuple board a read returns a tuple type object. This is because this version of the tuple board is not running on a version of Java that supports generics. In the J2SE version of the tuple board the same process could be accomplished without the need to cast the value of the returned tuple. An example of how a read would be done in J2SE is below:

```
PlayerTuple template = new PlayerTuple();
PlayerTuple read = this.cardTable.read(template);
```
Tuple iterators must be changed in a very similar way to the read operation of the tuple board. In the J2ME version, just like in the read, the returned tuple object will have to be cast to be used in the program. The J2SE version of the iterator object is able to return the proper type of the tuple simplifying things slightly for the developer.

Tuple board listeners also have to be handled differently on the J2ME version of the tuple board because the J2ME version is lacking generics support. The following code will illustrate how to setup a tuple listener in the J2ME environment:

```java
public class HoldEmGame implements PostListener{

    public HoldEmGame(){
        TupleBoard board = new TupleBoard(); //create a tuple board
        PlayerTuple temp = new PlayerTuple(); //make a template
        board.addListener(this,temp); //add a listener
    }

    posted(Tuple tuple){ //required because PostListener was implemented
        //will be called whenever a postlistener has detected something matching
        //its template
        //Tuple type object will be passed in
        if(tuple instanceof PlayerTuple){ //need to figure out what kind of tuple was
            //posted
            //    do Something;
        }
    }
}
```

Since the J2ME version of the tuple board doesn't support generics every listener will call the same posted method. In the J2SE version of the tuple board the listeners can be customized more. For example, to create a player tuple listener in J2SE:

```java
public class HoldEmGame implements PostListener<PlayerTuple>{

    public HoldEmGame(){
        TupleBoard board = new TupleBoard();
        PlayerTuple temp = new PlayerTuple();
        board.addListener(this,temp); //add a listener
    }

    posted(PlayerTuple tuple){ //required because PostListener was implemented
        //PlayerTuple type of object will be passed in
        //    do Something;
    }
}
```
In the J2SE version of the tuple board the posted method can be setup to deal with one specific type of tuple. The J2ME posted method will be called for all tuple types that are being listened for and it is up to the programmer to add logic to handle each type. In the J2SE version each posted method will be called for only the type specified making it easier to handle posted tuples.

During the implementation of the J2ME version of the tuple board, a known problem with the J2ME InetAddress class was encountered. When this class is used in an attempt to get the devices IP address the device returns its loop back address. The work around for this problem is to use a socket to get the J2ME devices local address. More details about this bug can be found in FAQ-0762 (Source 5).

4 Secure Card Deck

4.1 Card Deck Encryption and Decryption

When two or more people sit down to play cards one person typically shuffles the cards and then deals them out. Ideally after the cards are dealt no participant will know what cards their opponents have. This is the idea behind creating the secure deck of cards. In most multi player computer games there is a trusted server that would deal the cards to the participants. Since the tuple board is for an ad-hoc network a trusted server is not an option. Another way of securing the deck is required. The secure deck of cards is an attempt to securely shuffle and deal cards without the need of a trusted central server.

The secure deck of cards is designed to allow two or more devices to shuffle a deck of cards. After the cards are dealt no single device playing the game should know what cards its opponents have. Secure deck allows players to received their hands without their opponents being able to tell what cards they have.

The secure deck of cards uses the following technique to encrypt the cards in the deck. One player makes a large prime modulus $P$. Using $P$ they make their encryption key $E$ and decryption key $D$ as follows:

$$ED = 1 \pmod{\phi(p)}, \text{where } \phi(p) = p - 1 \text{ since } p \text{ is prime}$$

$P$ is shared with all the devices that are going to be using the secure deck of cards. Every device generates their own $E$ and $D$ based on $P$. Now that all devices have their encryption key each device will encrypt every card in the deck, shuffle the cards and pass them to the next player. To actually encrypt a card $X$ each user must raise the cards BigInteger representation to their encryption key as shown below:

$$X^E \pmod{p} = XE$$

The first player passes their list of encrypted cards to the second player who encrypts and shuffles as follows:

$$XE^{E2} \pmod{p} = XEE$$

After all the cards have been encrypted and shuffled by each player the cards need to be distributed to each player. Every player will get a copy of the entire deck. Players are given their cards
by being given the indexes into deck. This way players know what encrypted cards they have but not what they contain.

For a card to be shown to only the player it belongs to. They have to get all the players to use their decryption key on the cards. For a player to remove their encryption they must do the following:

\[(X^E)^D (mod \ p) \text{becomes } X\]

After all the players opponents have removed their encryption keys the player who wants to see the card will use their decryption key. The card value will now represent an actual card in the deck of cards.

To reveal a card to all players, all players would have to go through the decryption process individually. An alternative would be to have one player go through the decryption process and then share the value of the card with the other players. For this project, it was chosen to have all players do this individually so that no one player would have this kind of control.

### 4.2 Card Deck Tuples

This section will explain the tuple interactions use to create a secure deck of cards. The secure deck of cards relies on three tuples to get setup. The first tuple that it uses is the DeckInfoTuple (Tuple 1). The DeckInfoTuple allows the large prime modulus to be shared with all SecureDeck objects making the secure deck of cards. This tuple also allows every device to know what the unencrypted card representations are by sending an array of their original representations.

| private BigInteger cardNumbers[]
| private BigInteger primeModulus
| private final Integer deckNum |

*Tuple 1: DeckInfoTuple.*

The next tuple used by the secure deck is the DeckTuple (Tuple 2). This tuple is used for passing the encrypted and shuffled cards around to each player so that they can encrypt and shuffle the cards. The encrypted cards are sent in the cards array. The list of BoardIds that still have to encrypt the deck are stored in the stillToEncrypt linked list.

| public BigInteger[] cards
| public BoardId[] encryptedBy
| public LinkedList stillToEncrypt
| public final Integer deckNum |

*Tuple 2: DeckTuple.*

The final tuple that signals to all the devices that the secure deck is setup is the FinalDeckTuple (Tuple 3). This tuple allows all devices to know that the secure deck is ready for use and also contains the final encrypted deck that has been encrypted by all devices.

| public BigInteger[] cards
| public BoardId postedBy
| public final Integer deckNum |

*Tuple 3: FinalDeckTuple.*

15
The secure deck of cards also post HandTuples, which occurs when a card game asks the secure deck to deal hands to all the players in the game.

The sequence diagram in Figure 13 shows the first steps that the secure deck of cards goes through when at least two devices make one. There always needs to be exactly one main secure deck. There can be any number of regular secure deck instances. A device will have the main secure deck when it is constructed with the make deck flag set to true. As the sequence diagram shows the main device has to create the prime modulus that all the devices will use to calculate the encryption and decryption keys.

![Secure Deck Sequence Diagram](image)

*Figure 13: Secure Deck Sequence Diagram 1.*

After the main device creates the prime modulus it also immediately makes its encryption and decryption keys. The main device also has the responsibility of making up all of the original card numbers. The next step is that the main deck of cards will post a DeckInfoTuple. This tuple sends the original cards and the prime modulus to all of the other participating secure decks. When the regular decks receive this DeckInfoTuple they are able to use the prime modulus to calculate their encryption and decryption keys.

The prime modulus is calculated in the secure deck of cards by using the BigInteger class' constructor:

```java
BigInteger prime = new BigInteger(bitLength,certainty, new SecureRandom());
```

The bit length currently being used is sixty four bits. The game only needs to keep the cards secret for a short time. The encryption key is calculated just like the prime modulus. The decryption key is
calculated using the extended Euclidean algorithm.

Now that the main device has shuffled and encrypted the cards it will post a DeckTuple as shown in Figure 14. All secure decks will iterate over the DeckTuples. Since they are iterating they will only see each of the tuples once. Each device will check every DeckTuple to see if their ID is the next that should encrypt and shuffle. They will also make sure that there are still devices on the list to encrypt and shuffle. If the device is not supposed to encrypt this tuple and the list of devices to encrypt is not empty the device will read the next tuple. If this is the device that is supposed to encrypt the cards next it will take the cards from the tuple, apply its encryption key to all of the cards and then it will shuffle all of them. Then it will create a new DeckTuple, removing itself from the list of devices that should still encrypt. After it has encrypted, shuffled and removed itself from the list of devices still to encrypt it will post a DeckTuple with these encrypted and shuffled cards. After it posts the DeckTuple it will go back to reading in all the tuples. This loop is shown in Figure 14 and is the same for all devices. When a device reads in a DeckTuple and the still to encrypt linked list is empty the device will stop looping and will post a FinalDeckTuple.

![Secure Deck Sequence Diagram 2](image)

*Figure 14: Secure Deck Sequence Diagram 2.*

As shown in Figure 15 once devices stop iterating over the DeckTuples they will read in a
FinalDeck Tuple and every device will withdraw all tuples except the FinalDeck tuples and HandTuples.

4.3 Security Threats

Ad hoc networks present unique security considerations for the applications that are written to use them. The secure deck of cards that was developed for this project and the fixed helix security context are two tools that developers can use to combat some security threats. Many applications that are developed to run on the tuple board or on an ad-hoc network in general will require the ability to securely transfer data from one device to another. Data sent over any network has some risk of being intercepted. This risk is increased when the network is ad-hoc and does not have any trusted infrastructure to rely on. The helix streams that were developed for the J2SE version of the tuple board and ported to work on the J2ME version of the tuple board offers a method to encrypt data while it is being transmitted. When a tuple board application is using the default security context any device would be able to intercept and understand the data being transferred. The Fixed Helix security context encrypts the data that is being sent making it much more difficult for an attacker to gain access to any confidential information that may be being sent. A malicious user would simply have to intercept the confidential data if the application only used a default security context. If the application used the fixed helix security context the malicious user would have to intercept the data and be able to decrypt it to gain access to this information.

Card games present additional security threats. The secure deck of cards was developed to help solve some of these problems. The first problem is that without a trusted server how can cards be distributed in secret? Posting tuples that say which cards a user has could easily be read by the other users at the table allowing them to gain an advantage. The secure deck of cards mitigates this problem. All cards are encrypted by all participants and then distributed no device is able to see device 1 cards unless device 1 removes its encryption.

Why can't one device use a FixHelixSecurityContext to tell individual devices their cards? This has a couple of problems the first is that the dealing device probably wants to play in the game and if they are telling other players what card values they have the dealing device will have an advantage. Even if the dealing device decides to sit out a round this still isn't very secure since it could tell another device 18
device about the cards it is distributing without the knowledge of the other devices. Devices that may be communicating in secret when the secure deck of cards is in use would only be able to tell each other what cards they had and would not be able to remove the encryption from their opponents cards without tricking their opponent into removing it for them.

5 Game of War

5.1 Description of War Game

War is a game that is traditionally played between two human players with a deck of cards. The deck is shuffled and the cards are split in half with each player receiving twenty six cards. Game play begins by each player taking the top card off of their deck and placing it face up on the table. The values of the cards are then compared. Usually ace is high. The player who played the higher value card gets to put both cards on the bottom of their deck. In some cases both players will play a card with the same value. When both players have played cards with the same value a “war” happens. Each player will place three more cards face down on the table. Next they will get another card from their pile and flip it over. At this point the values of these cards are compared and the winner takes all of the cards both face up and face down. There is a chance that a war could result when both players obtain the same value on their second flip, subsequently another war takes place. Play continues in this fashion until one player is able to win all of their opponents cards.

5.2 User's Manual For Tuple Board Version

The war game program that has been developed uses the rules described in section 5.1. The war game that was developed for this project will allow two players to play the game of war using an ad-hoc network.

To start the game double click on the cards.jar file that was produced by Netbeans. It should be located in the dist (For example C:\Documents and Settings\Travis\My Documents\NetBeansProjects\War\dist) folder under the Java project.

If you are going to be playing on the phone you will first have to install the game (see the installing game on the phone section in the appendix of this document). After the game is installed on the phone it will be located under the tools menu and can be launched from there.

After the application has been open you will have to:

- Click on the “War” button to play a game of war.
- To create a session you need to type the session name in the text box next to the Create Session button (Figure 16).
- To create a secure session a password should be entered in the text box next to the “Join Session” button (Figure 16).

![Session Select Gui](image)

Figure 16: Session Select Gui
Any other devices that are on the same Local Area Network should be capable of seeing the created session.

To join a session click on the session in the list and click the “Join Session” button. If the session has the word locked after the session name you will need to enter the proper password in order to join the session. The password should be entered in the text box next to the join session button.

After a user has successfully joined or created a session the War Game Gui (Figure 17) will be presented to the user. The play card button will be disabled until two people have joined the session. Note that there will be some delay between the second user joining and the game beginning. This is caused by the two devices needing to setup the Secure Deck of cards that will be used.

After two players have joined the game the “Play Card” button will be enabled. After it is enabled users should click the Play Card button to flip over a card. Next to where the card is displayed is the count of how many cards each user has in play and in their pile. The winner of each round will have the choice of which of the two cards won to take first, as shown in Figure 17. If the round ends with a war the game will display that a war has occurred and the “Play Card” button will be enabled. The game will put three cards from each player face down on the table and then turn over each players fourth card. The winner of a round with a single war will get all five of their opponents cards.

The game will end when one player runs out of cards. Both devices will display the game is over and both players should quit and start a new session if they desire another game of war.

If one of the devices loses its connection a message will be displayed on both devices saying that someone disconnected. Devices losing their connection could be a common occurrence depending on the ad-hoc network being used. If the device is able to reconnect to the game the session will recover and will continue from where play left off. If the disconnected user can't reconnect both users will have to quit the game and start over.

![Figure 17: Winner of Round shown on War Game Gui.](image)
5.3 Tuple interactions

Tuples Used

The War game posts many types of tuples to the “tuple board” to allow the game to function. Each tuple allows different information to be shared with the other devices that are using the tuple board. Since the war game is designed to run on an ad-hoc network the devices on the network share these tuples directly instead of relying on a server. The figures below show the data each tuple can hold and gives a short description of how the war game uses this tuple.

**Tuple 4: HandTuple**-allows devices to get their hand. The linked list of card numbers contains indexes into the final deck.

```java
public final BoardId whoseCards
public LinkedList cardNumbers
```

**Tuple 5: OrderTuple**-used to exchange who is playing the game.

```java
public LinkedList order
public BoardId postedBy
```

**Tuple 6: WarGameStateTuple**-used to update the main status: who has played what card and what cards they have in their hand.

```java
public final Integer turn
public final LinkedList cardsInPlay
public final LinkedList pile
public final BoardId postedBy
```

**Tuple 7: SessionTuple**-used to announce that a new session has been created.

```java
private String sessionName
```

**Tuple 8: DecryptedTuple**-used to share cards that one device has used decryption on.

```java
public final BigInteger player1decrypted
public final BigInteger player2decrypted
public final BoardId decryptedBy
public final Integer turn
```

**Tuple 9: PlayerTuple**-used to announce the presence of another device. The playersBoard field is used when making list of BoardId that are playing.

```java
private String name(not used)
private BoardId playersBoard
```
The War game is designed to first go through some initialization before the game can begin. These steps are outlined in Figure 18. The first initialization step is for the devices to create session tuple listeners. The listeners allow the WarGame object to tell the tuple board object they would like to be notified when a session tuple is posted.

The next step is for one of the devices to post a session tuple. The session tuple is posted twice once in the clear, with no encryption so anyone can read it, and once using a security context. Only devices that are able to create the proper security context will be able to read the second tuple. If the device does not care who it plays it will post the second session tuple using the default security context. Any device can make a default security context. If the user wants to create a private game their password will be used to make a Fixed Helix security context. The password is used as the encryption key. **This is for demonstration purposes only, this is not secure (see future work section).** If it is a secure session the joining device will have to make sure it can read the session tuple that was posted using the security context. If it can then it has the correct key/password and will be able to play the game.

Figure 18: Sequence Digram shows the first steps to initialize the WarGame.
After the session tuple has been posted a WarGame object will be created and its thread will be launched. This tuple announces to other war games a session exists. The tuple board will notify all of the devices that created listeners for this event. After a device is notified it can join the session by posting a PlayerTuple. The device that posted the SessionTuple reads in all player tuples until it finds one that isn't its own. After it finds a player tuple that is not its own initialization can continue. At this time the SecureDeck object is created. Both devices will create a SecureDeck object. The SecureDeck objects will interact with each other and initialize the secure deck of cards. More details about this process can be found in the secure card deck implementation section of this document.

After both devices have created their SecureDeck objects the war games will read in a final deck tuple. The read function will allow the two games to block until the final deck has been successfully created by the SecureDeck objects. The FinalDeckTuple is the signal that the secure deck of cards has been initialized and is ready to be used. The final step of initialization is that the device that created the session will post an OrderTuple which allows the device that joined to know who it is playing against. The remainder of the steps for the War game initialization are shown in Figure 19.

Figure 19: Completion of the War game initialization.
Once initialization is complete the actual game begins. WarGame (1) (The session creator) will request that the SecureDeck object deal twenty six cards to both players. The SecureDeck object will deal the cards by posting a HandTuple (Tuple 4) for both players. The HandTuple will contain a list of indexes of the cards that the user has been given in the final deck. Using this index they can get the encrypted representation of their card which can be decrypted using both devices decryption keys. Each device will read until their HandTuple is returned by the tuple board. Once a device has its hand tuple it will enable the play cards button on the war game GUI (Figure 17). The first steps of the War game are diagrammed in Figure 20.

Figure 20: First steps in War game round.
After the device has enabled the play card button the game continues as shown in Figure 21. After a user presses the play card button the device will post a WarGameStateTuple (Tuple 6) to let the other device know that this device has played a card. This tuple allows a device to post the indexes of the cards it is placing on the table for the war and also what cards are in its pile (the pile of cards that are not being played yet). After a device has posted its state it will attempt to read the opponents state until it is returned by the tuple board.

After the device has been able to read in its opponent’s state it will ask its SecureDeck of cards object to tell it what encrypted BigIntegers are associated with each of the played card indexes. The device will ask its secure deck of cards to use its decryption key on the cards played by the user and the opponent. This leaves just the opponents key encrypting each of the cards.

Figure 22 shows the remainder of the steps in the War game. Each device will post a DecryptedTuple (Tuple 8) containing the cards with their encryption removed. Both devices will read in the DecryptedTuple that was posted by their opponent. When the tuple board returns the DecryptedTuple from their opponent the devices can use their decryption key to see the values of the cards. These decrypted cards are displayed to the user.

Using the decrypted cards each device will calculate whether they won, lost, or need to do a war. If the user:

<table>
<thead>
<tr>
<th>WarGame (1)</th>
<th>SecureDeck</th>
<th>TupleBoard</th>
<th>WarGame (2)</th>
</tr>
</thead>
</table>

**Figure 21: War game round middle**
• Won, the device will enable the take card buttons. After the user has picked up the cards it won it will cycle back to where the play card button is enabled (Figure 21).
• Lost, the game will cycle back to where the play card button is enabled (Figure 21).
• If war (two cards with the same face value) is detected the game will cycle back to where the play card button is enabled (Figure 21). Additionally the count of how many cards will be moved from the pile to the played pile will increase from one to four and the process will repeat until someone wins the round.

The game will continue looping until one device has run out of cards in its pile and can no longer play.

5.4 Software Design

During the War game program's initialization it uses the classes that are shown in Figure 23. The GameSel class is the launcher class that contains the programs main function. The GameSel class creates the GameSelFrame object.

The GameSelFrame object extends Frame and displays a GUI for the user to pick whether or not they wish to play War or Texas hold'em. If the user selects the War game the GameSelFrame will create a WarStart object.
The WarStart object will create a SessionSelectUI and a TupleBoard object. The WarStart class implements two interfaces. The first interface that the WarStart class implements is the PostWithdrawListener. The PostWithdrawListener requires the WarStart class to implement a posted(Tuple tuple) function and a withdrawn(Tuple tuple) function. The WarStart class will register itself as a PostWithdrawListener to the TupleBoard object that it creates. The tuple board will then be able to alert the WarStart object when tuples are posted and withdrawn from the tuple board. For more information on TupleBoard listeners see the source entitled “edu.rit.tb (Tuple Board Documentation)”. The second interface that the WarStart object implements is the SessionSelectUIListener. This interface requires the WarStart object to implement two functions createSessionClicked and joinSession. These two functions allow the WarStart object to tell the SessionSelectUI that it is a listener and the SessionSelectUI will call the WarStart object appropriately.

SessionSelectUI is a GUI object that allows a user to create or join sessions of the War game. The SessionSelectUI will call the SessionSelectUIListener functions on the WarStart object to communicate the user's wishes.
After the user interacts with the SessionSelectUI the WarStart object will create a WarGame object. The WarGame object implements two interfaces. The first interface that the WarGame implements is the WarSessionListener. The WarSessionListener interface requires the WarGame to implement the following functions playCard, quitSessionClicked, and takeCards. These methods allow the WarSession GUI to interact with the WarGame object. The second interface that is implemented by the WarGame object is the PostWithdrawListener. The WarGame object creates the following objects WarSession (GUI), SecureDeck, and a WarThread. It also uses the TupleBoard to communicate with other WarGame objects that are playing against it.

The WarSession object is created by the WarGame object and is the GUI used to play the WarGame. This class allows the user to interact with the WarGame. It allows the user to play cards, quit, and take cards after the user has won a round.

The SecureDeck is used by the WarGame to encrypt all the cards, deal the cards, and allow all the games to decrypt the cards at the proper time.

The WarThread object is another thread of execution that the WarGame launches after it has played its card. The WarThread is responsible for reading in the opponent's cards, getting the cards decrypted for a round and displaying the cards to the user.

Figure 24: War game classes for actual game play.
6 Game of Texas hold'em

6.1 Description of the game

Texas hold 'em is a variant of poker in which players are dealt two hole cards and all players share five community cards. Each player creates their best hand by using the five of the seven cards that give them the best poker hand.

Play begins with all participants receiving two cards face down. Like most poker games the person sitting to the left of dealer receives their cards first and the cards are dealt clockwise with the dealer getting their cards last. These cards will only be shown to the other players at the end of the game.

Everyone will look at their two cards at this point. The person to the dealers immediate left is required to put a certain amount of money into the pot known as the small blind. The person to the small blinds left is known as the blind and is required to put the blind bet into the pot. The blind is usually a larger sum then the small blind. Betting proceeds in a clockwise manner until all players have either put in the same amount of money or have folded.

After the initial round of betting is over the dealer will deal three community cards face-up this is known as the flop. Players will bet based on their hole cards and the flop starting with the person to the dealers left. After the second round of betting is complete the dealer will flip over one more community card called the turn card. Betting will commence again after which the dealer will turn over the final community card called the river. Betting will happen one last time. After the final round of betting completes, assuming there is still at least two people still active (everyone hasn't folded), the showdown will occur. In the showdown each player will create their best five card poker hand. The hands are compared and a winner will be awarded the pot. If both have identical value hands then they will split the pot.

6.2 User's Manual for Tuple Board Version

The Texas hold 'em game can be started exactly the same way as the War game except the Texas hold'em option should be picked instead of the War game option.

Session creation

Session creation is similar to the War game as well. The steps are to:

1) Put the name of the session in the box next to the “Create Session” button.
2) If an encrypted game is desired then the box next the “Join Session” button should be filled in with the password for the session.
3) Click the “Create Session” button.
4) Set the default conditions for the game.
a) The number of players is the number of participants the game will contain. The number of players should be from two to four players.
b) The default number of credits is the number of credits users start the game with. Each user can actually decide how much they want to start with.
c) The small blind value is the amount the small blind will be. The big blind will be double the small blind.
5) Click “OK” and other devices on the local area network will be able to see this newly created session. The sessions will appear as shown on the Session Select GUI Figure 16. The one difference from what is displayed is that for the Texas hold 'em game the small blind information will also be displayed.

**Joining a Session**
A user wishing to join an already created session will:
1. Click on the session to join in the list of sessions. All the sessions that are currently posted on the ad-hoc network should be visible.
2. If the session requires a password it needs to be entered in the text box next to the “Join Session” button.
3. Enter how many credits to start the game with as shown in Figure 26.

The Green Section of Figure 26 will display the community cards. Play button will bring up a window for betting. The Bet Window (Figure 27) allows user to call (match bet) other users so far. Or raise force other players to bet more. Fold give up on this hand.

**Playing The Game**
When the users have all joined the session, the game will take some time to initialize the SecureDeck object. After the SecureDeck object is setup the blinds will automatically place their bets. Players that need to bet or fold will be prompted to do so by the play button being enabled as shown in Figure 27. After the user has hit the play button the bet window (Figure 28) will be displayed. This window will allow players to raise, call, or fold. Players will get to bet after the flop, the turn, and the
After all the rounds of betting are complete all players except the dealer will have the ability to decide whether or not they want to stop playing as shown in Figure 29. The dealer will be able to quit after the next round.

![Image](figure29.png)

**Figure 29:** End of turn Gui. Stop playing will allow user to gracefully leave the game. Keep playing they will keep playing. Amount won is how much they won. It also shows the cards that each player had.

### Devices Leaving Game

![Image](figure30.png)

**Figure 30:** Lets users know someone has left the game in the middle of a round.

If any devices leave the game in the middle of the round the game will not be able to continue unless that device can reconnect. If a device leaves during the middle of a game all users will see the window shown in Figure 30. Since the game is being played on an ad-hoc network and no server is used in the game it can be hard to figure out who has disconnected. As a rule of thumb if the game has more then two players a user who sees more then one of Figure 30 may have lost connection and should try to reconnect to the ad-hoc network. A user that only sees one of these should wait for a little while to see if the person can reconnect. If the other user reconnects the game will pick up from where it left off. If the other device cannot reconnect then the other users will have to quit the game and begin a new session to continue playing.
6.3 Tuple interactions

Similar to the War program the Texas hold 'em game posts many different types of tuples to the tuple board in order for it to function. Each tuple allows different information to be shared with the other devices that are using the tuple board. Since this game is designed to work in an ad-hoc networking environment the client devices share this information directly without using a server. The figures below summarize the data each tuple can hold and gives a short description of how the game uses this tuple.

<table>
<thead>
<tr>
<th>Tuple 11: PokerPlayerStatusTuple</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PokerPlayerStatusTuple</strong> - allows hold em games to update other devices about their current status.</td>
</tr>
<tr>
<td>public LinkedList toDecrypt</td>
</tr>
<tr>
<td>public final BoardId statusOf</td>
</tr>
<tr>
<td>public Integer money</td>
</tr>
<tr>
<td>public String status</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tuple 10: GetDecryptedTuple</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GetDecryptedTuple</strong> - is used by the Texas hold 'em game to allow the card to be decrypted by all the devices currently playing the game.</td>
</tr>
<tr>
<td>public LinkedList toDecrypt</td>
</tr>
<tr>
<td>public final BoardId nextToDecrypt</td>
</tr>
<tr>
<td>public final BoardId wantedDecrypted</td>
</tr>
<tr>
<td>public final Integer turn</td>
</tr>
<tr>
<td>public LinkedList orderToDecrypt</td>
</tr>
<tr>
<td>public final Integer round</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tuple 12: TellPlayerTurnTuple</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TellPlayerTurnTuple</strong> - allows devices to know what BoardId is having a turn. The community int lets devices know if a flop river or turn card is coming. The smallBlind and bigBlind fields let a device know whether it is the big blind or not. The toRaise and toCall fields represent how much a device needs to play to call or to raise the other devices. The turn and round fields help keep all the devices on the same page.</td>
</tr>
<tr>
<td>public final BoardId yourTurn</td>
</tr>
<tr>
<td>public boolean smallBlind</td>
</tr>
<tr>
<td>public boolean bigBlind</td>
</tr>
<tr>
<td>public Integer toRaise</td>
</tr>
<tr>
<td>public Integer toCall</td>
</tr>
<tr>
<td>public final Integer number</td>
</tr>
<tr>
<td>public int community</td>
</tr>
<tr>
<td>public final Integer round</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tuple 13: TotalBetTuple</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TotalBetTuple</strong> - is used by devices to let other devices know how much they are betting. The totalBet field is how much the device is betting. The moneyLeft field is used to keep track of how much money the player betting still has. The id is the player who is betting. Increment is a number of actions this round. Folded indicates that this device has folded. The round field is number of the rounds the game has been through.</td>
</tr>
<tr>
<td>public final Integer totalBet</td>
</tr>
<tr>
<td>public final Integer moneyLeft</td>
</tr>
<tr>
<td>public final BoardId id</td>
</tr>
<tr>
<td>public final Integer increment</td>
</tr>
<tr>
<td>public final Boolean folded</td>
</tr>
<tr>
<td>public final Integer round</td>
</tr>
</tbody>
</table>
Initialization

The poker game is designed to start almost exactly like the War game; the only difference is that it asks the user for more information and there may be more than one opponent. The game is setup to work with two to four total players. Once the HoldEmGame object thread is started, the first thing it will do is create the SecureDeck of cards object. Similar to how the war game program works the creator of the session will wait for all the players to join the game. Then it will create the SecureDeck object with the makedeck flag set to true.

**Tuple 14:** `HandResultTuple`- is a tuple that allows all devices to see what another device has for a hand.

```
public final Integer turn
public HandResult myHand
public final BoardId postedBy
public final Integer round
```

**Tuple 15:** `KeepPlayingTuple`- allows devices to let other device know whether they are going to play the next round.

```
public final Integer turn
public final BoardId player
public final Boolean playOn
public final Integer round
```

**Tuple 16:** `DealerTuple`- allows a device to assign another device to be the dealer in the next round.

```
public final BoardId newDealer
public final Integer turn
public final BoardId oldDealer
public final Integer numPlayers
public final Integer smallBlind
public final Integer round
```

---

Figure 31: Texas hold 'em just as the game starts.
All of the devices will have to wait for the SecureDeck to go around and secure the cards for the deal. The game will wait for the SecureDeck object to post a FinalDeckTuple (Tuple 3). As shown in Figure 31 the creator of the session (Poker (1)/Dealing device) will be the dealer in this first hand. After the final deck is detected the dealing device will give all the players two cards. Devices will figure out the BoardIds of their opponents by using the order tuple that the dealing device will post. Poker (1) in the sequence diagrams represents the device that is the dealer for the round. The figures only show a game with two participants. If the game has more players the other devices would behave exactly like Poker (2).

Figure 32: Texas hold 'em Game decrypting hand.

All the devices will read in their HandTuple (Tuple 4) and get the cards in their hand decrypted in order to be displayed to the users. The hand is read in by reading in a HandTuple (Tuple 4). This is done by finding the HandTuple for the BoardId that the device is using to identify itself. The device will then figure out what encrypted card it has been assigned by using the index that was stored in the HandTuple for its hand.
The Game

Whenever a device needs to display a card it has to ask all of its opponents to decrypt the card. Figures 31, 32, and 33 show the interactions needed in order to decrypt a card. Each device keeps track of which device is sitting on its left, right and across from them. Each device will ask an opponent to use its decryption key on the card, get it back and then ask another opponent to use their decryption key on the card. This is performed until all devices have removed their decryption key. After all the opponents have removed their key, the device will use its own key and the card will be decrypted.

After each device has had their two cards decrypted it will post information about its state so that all the devices can update their screens. This is done by every device creating and posting a PokerPlayerStatusTuple (Tuple 11). Using the statusOf boardId (field in PokerPlayerStatusTuple) all
the devices are able to update their screens with the proper amount of money for all the opponents at the start of a round.

The creator of the session/dealer/Poker (1) for this round will direct the other devices when it is their turn. The dealer will be a different device each round so that only one device does not have to do most of the work for the entire game; the other devices will do as they are told in turns.

The dealing device will go through the following processes:
1. Get everyones initial bets.
2. Turn over the first three community cards.
3. Read in the flop.
4. Get bets now that the flop has been read by all devices.
5. Flip over the Turn card.
6. Read in the community card.
7. Get bets again.
8. Flip over the River card.
9. Read in the community card.
10. Get bets again.
11. Find winner of Round.
12. Figure out who will keep playing and transfer control to next dealer.

The other devices wait until the dealing device posts a tuple that lets them know what is expected of them. During each of the above steps the dealer will post many TellPlayerTurnTuples.

Figure 35: Hold'em game in the get bets portion.

These tuples tell the devices that are not dealing when it is required for them to do something.
Each device counts the “turn” (not the turn card just a count of interactions) as each round of the game progresses. Many tuples use this number to ensure devices are reading tuples in the proper order. Since card games are order dependent it is important that tuples be read in the proper order. This way the devices will always be able to make sure they are reading the most up to date tuples from a particular device. The program is setup so every device counts independently, but should always be on the same number so that the game can continue. The round is also associated with this counting. The round counter makes sure that devices only read in tuples from the current round of play.

**Get Bets**

![Sequence Diagram](image)

*Figure 36: Hold'em game end of get bets.*

The get bets functionality is setup so that all the devices can place their bets. Figures 35 and 36 show sequence diagrams of the get bets functionality. The game is setup so two devices will have to post blinds unless only two devices are playing. When there is only two players the one thats not dealing will post a blind. Each player every round will have a chance to bet. If a device bets early it will have a chance to stay in the hand by matching the amount of money the others have bet.

The dealing device will use the list of players that joined to go through each players turn to bet. The dealers ID will be the last one in the list so it will always be the last to bet. Using the joined list a TellPlayerTurnTuple (Tuple 12) will be created and posted to tell the device thats it is their turn to bet. The dealer will let the small and big blinds know who they are. A device that is told it is a blind will automatically post their bet otherwise the dealer will be keeping track of the highest bet so far. For example in a game with three players the dealer will tell ID 1 it is the small blind. The dealer will tell ID 2 that it is the big blind and then it would tell itself that to call it must match the big blind. The dealer will then tell the small blind that they must match the big blind as well. After cards have been turned over this process works the same except nobody has to put in a blind; the first player could call with a bet of zero. If another device raises, the dealer will tell the first device to match or fold.

When a device finds out that it is time for it to bet it will do one of two actions: if it is a blind it will automatically have to bet and will do so or it will enable the “Play” button and wait for the user to place a bet. In either case the device will be creating and posting a TotalBetTuple (Tuple 13). All other devices will wait for the betting device to post this tuple and then update their displays.

**Community Cards**

The poker (1) device will post the flop after the first round of betting is complete. The Poker (1) device will tell the other devices that it is time for a particular community card by setting the
community field of the TellPlayerTurnTuple (Tuple 12) to predefined values that all the devices understand. It will ask the SecureDeck of cards for the next three cards from the deck. The next three cards at this time will be the indexes of the next three cards in everyones final deck. A CommunityCardTuple will be posted for every index. To keep the order the same for all the devices the number of turns will be incremented so all the devices will read them in the same order.

After the dealer has posted the flop each device will read in the three cards that were posted by the dealer. They will then use the index that was in the tuple to find the BigInteger representation that was in the FinalDeck that will have been encrypted by all participants' encryption key. At this time for each card each device will do the decryption steps shown in figures 31, 32, 33. The turn and the river community cards work the same way as the flop except instead of three cards being revealed only one card will go through the above process.

Figure 37: Texas hold 'em flop (turn over first three community cards). Same as rest of community cards except flop has three and river and turn are only one card.
Find Winner

After all the rounds of betting and flipping cards have been finished it is time to figure out what device should take the pot. Figure 38 shows the sequence of interactions that are required to figure out who the winner is. Poker (1) will post a TellPlayerTurnTuple (Tuple 12) that lets the devices know it is time to calculate the winner. Each device will then figure out what its best hand is.

The principles for determining the best hand came from the "Best Poker Hand" forum topic. There was code posted but I wrote it myself and modified the idea some. The basic idea on the forum was to assign three numbers to each hand. The major rank tells you what kind of hand that the device has. For example this could be something like a flush, pair, etc. The minor rank represents the value of the major rank. For example if a device has a pair of kings the major rank would be pair and the minor rank would be king. The third part is to keep track of kickers. Kickers are the rest of the five card poker hand. In the example of having a pair since a hand is five cards you would have three kickers. The kickers are sent along to the other devices with the major and minor rank so that each device can figure out who won.

After all the devices have figured out what their best hand is, each device will post a HandResultTuple (Tuple 14) so that all the other devices can see their result. Each device will read in all the other HandResultTuples (Tuple 14) and figure out who they think the winner is. There are three possible outcomes to this comparison: the device beat all other devices, split or lost. If a device beats all it can take the whole pot. Split this means that two or more devices had the same value hand. When this is the result they will figure out how many people need to split the pot. Each device will take an equal share. Finally, the device could have lost and it will not take anything from the pot.

Figure 38: Hold'em tuple interactions to find the winner.
At the end of every round the dealer will check and see who is still playing in the next round. It will assign the next dealer and tell the new dealer how many people are going to keep playing. Figure 39 shows the tuple interactions that take place in order for the Poker (1) device to pick the next dealer.

The Poker (1) device will iterate over all the KeepPlayingTuples (Tuple 15) that all the devices should be posting. All devices that are in the Poker (2) role will determine the winner of the round and then prompt their user to see if they wish to keep playing. After the user has responded each Poker (2) device will post a KeepPlayingTuple (Tuple 15). The Poker (1) device, after it has read in whether or not all the other devices will keep playing, will then ask its user whether or not to keep playing. Poker (1), knowing who will be playing the next round, will assign the job of being the dealer (Poker (1)). It will assign this responsibility to the device that will still be playing closest to itself going around the table clockwise.

After the dealer device figures what device it wants to be the dealer next, it will post a DealerTuple (Tuple 16) allowing the devices to know who the new dealer is. This tuple will also allow the new dealer to know how many devices are going to keep playing and what the small blind for the game is.

All the devices will then wait until they can read in the DealerTuple (Tuple 16). Once they read in the DealerTuple (Tuple 16) another round of the game will begin.
6.4 Software Design

The classes involved in getting the Texas hold 'em game to start are similar to the classes that are used to start the War game. The first set of classes created when a user wishes to play Poker are shown in Figure 40. The first difference between the two games is that the PokerStart class is substituted for the WarStart class. The PokerStart class implements a couple of interfaces that the WarStart class does not implement. The first new interface is PokerSetupUI listener. The PokerSetupUI listener requires the PokerStart class to implement the following function: setupOkButtonClicked. The setupOkButtonClicked allows the CreateSetupUI to interact with the PokerStart class. The second new interface implemented is the JoinSetupUIListener. The JoinSetupUIListener interface requires the PokerStart class to implement the following function: joinSetupOkButton. The JoinSetupUIListener allows the JoinSetupUI to interact with the PokerStart class. The JoinSetupUI allows a user who is joining a session to set how much money they wish to start with.

Figure 40: First classes created when the Texas hold 'em game begins.
The HoldEmGame object shown in Figure 41 is the main class for the Texas hold'em game. The HoldEmGame object implements four interfaces. The first interface that it implements is the HoldEmTableFrameListener which allows the HoldEmTableFrame object to interact with the HoldEmGame object. This interface requires the HoldEmGame object to implement the function playTurnClicked allowing the HoldEmTableFrame object to tell the HoldEmGame that a user has clicked on the “Play” button. The second interface is the PostWithdrawListener. This is used by the game to detect when a user may have lost connection. The TupleBoard will interact with the HoldEmGame object when using the PostWithdrawListener. The third interface is the PlaceBetFrameListener which allows the PostBetFrame to interact with the HoldEmGame object. The PostBetFrame is able to call three functions implemented by the HoldEmGame object to communicate the user's intentions. The three functions are callClicked, foldClicked, and raiseClicked. The last interface that the HoldEmGame object implements is Runnable allowing the HoldEmGame object to run on its own thread.

The classes shown in the Figure 42 show the classes that handle the actual game Logic. Most of the logic resides in the HoldEmGame object. The HoldEmGame object relies on the tuple board to communicate with other HoldEmGame processes. The SecureDeck object is used to keep the cards secure and to allow them to be viewed when necessary. The HoldEmGame relies on HoldEmTableFrame (Figure 27) object and the PlaceBetFrame (Figure 28) object to interact with the user.
The PokerStart object also is required for one important task that is not really part of initialization. At the end of every round of poker the KeepPlayingFrame (Figure 29) allows the user to decide whether or not to keep playing. This is communicated back to the PokerStart object which allows that object to launch a new HoldEmGame object for the next round.

*Figure 42: Classes during Texas hold 'em game play.*

The PokerStart object also is required for one important task that is not really part of initialization. At the end of every round of poker the KeepPlayingFrame (Figure 29) allows the user to decide whether or not to keep playing. This is communicated back to the PokerStart object which allows that object to launch a new HoldEmGame object for the next round.
7 Performance Testing

7.1 War Game Performance

The performance of the War game is good. The game seems to operate for a relatively long time with no problems. Since War is such a long game, however, the phones do sometimes freeze. This freezing usually does not take place until after thirty minutes or so.

<table>
<thead>
<tr>
<th></th>
<th>Processor</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dell</td>
<td>Intel Dual Core 1.66 GHz</td>
<td>1.5 GB</td>
</tr>
<tr>
<td>Ibm</td>
<td>Pentium 3 866 MHz</td>
<td>384 MB</td>
</tr>
<tr>
<td>Imac</td>
<td>Mac G4 800 Mhz</td>
<td>256 MB</td>
</tr>
<tr>
<td>Phone</td>
<td>ARM 208 Mhz</td>
<td>61 MB (16.6 Free after startup)</td>
</tr>
</tbody>
</table>

*Table 1: Specs of the devices used for evaluation*

The specifications of the devices used for testing are shown in Table 1. The Dell vs Dell case shown is the game running in two processes on the same computer. For the War game I timed how long it took after both devices hit the play button to when the devices were able to declare who had won the round. When either device was able to update its status with the winner the timer was stopped. Usually both devices updated within a couple of seconds of each other.

As the results in Table 2 show in a setup where phones are used the game play goes slower. The Dell vs IBM case establishes somewhat of a baseline to determine the effect of the network on slowing down game play. The rest of the slow down when the phone(s) get involved is possibly caused by the phones having a slower processor.

As a player my opinion is that the war game displays decent performance that does not inhibit play.

7.2 Texas hold 'em game performance

The performance of the Texas Hold 'em game on the phone is slow and sometimes unstable. The game seems to work well when being run on personal computers. The game seems to freeze when the phone is involved. The more players that are attempting to play on a phone the worse the game gets. To better understand how much slower the phone makes the game I timed the game using different hardware. Times are from when the table frame is displayed (Figure 27) on all devices to when the first device displays the end of round window (Figure 29). In an attempt to keep the times as equal as possible every time the play button is enabled I simply hit the call button. All times have been recorded in seconds.
Table 3 shows a strong correlation between the game being much slower when run on the phone as compared to being run on the computers. Not only is the game slow but it also becomes unstable. Once the phone freezes it doesn't seem to be able to recover. The only way to get the phone to continue is to shut it off and start all new games. Overall this is a disappointing result for the Texas hold 'em game's performance.

### 8 Future Work

I believe that some other areas of research could be pursued from this point:

- Replace session passwords as key by using a key exchange.
- Try applications on a phone besides the Sony Ericsson 990. The 990 seems to have many negative reviews about its firmware so there may be similar phones that can play the games better. The phone was very new when I started my project so I was not aware of the shortcomings at the time.
- Port more than the basic functionality of tuple board to JME CDC platform, things such as the control panel that can be used for debugging, etc.
- Port card games to work with J2SE version of tuple board
- Make secure deck of cards use authentication to actually prove that device that wants card decrypted is actually supposed to be able to see that card.
- Port applications to Google Android seems to have more functionality including the .nio packages. I believe the J2SE tuple board may actually be able to run on this platform.

### 9 Conclusion

I have shown that J2ME devices can be used to run tuple board programs. As these devices get a little more powerful I assume some of the performance issues that my games suffered from will be resolved. I was also able to create a secure deck API that can be used by card games in an ad hoc environment to play cards without knowing what devices have what cards.

I learned how to port an existing library from Java 2 standard edition to Java 2 micro edition. I now understand some of the differences between these editions of Java. I learned how to setup a J2ME development environment. Since J2ME is similar to J2SE 1.3 it has also given me some experience programming in a legacy version of Java. This could be very beneficial if I ever have to work on such a system in my career.

Using the debuggers was something that most of my previous programming did not require. One significant area of debugging for this project involved a lot of low level network IO issues.

Working on a non PC platform was something this project let me explore. I found that many times things that seems to work in my project on the PC and emulator would not work as well on the actual phone hardware. I guess the main lesson that can be learned from this is test software early and often on the intended hardware.
Working with a hardware that was not very open was also a challenging aspect of this project. It is extremely frustrating to try and figure out what is happening when the phones freezes. The phone doesn't have a console so error messages are not readily available. The phone was pretty much a black box and it was difficult to understand why software that works well on both PC architecture and emulator would freeze on the phone. Was it my actual software? Does the phone just freeze? Combination of both? While frustrating I think this experience will possibly be similar to real world examples of writing software that may have to run on a custom system and should serve me well. Also having to deal with getting my programs signed every time I wanted to try a slight mod on the phone was very frustrating and something I hope will become easier in the future.

Documenting my work in this report also allowed me to learn a lot about how to use sequence diagrams to more effectively demonstrate the flow of a program. As well as using UML type class diagrams to more effectively show the design of individual classes in my program.
Appendix A. Developers Manual

A.1 Development Environment Setup

The setup associated with development for the Sony Ericsson 990 is documented on the NetBeans CDC Emulator Platform Setup Guide. Netbeans and the Java Development Kit (JDK) will need to be downloaded and installed on the computer you are going to use for development.

Next the emulator for the 990 will need to be installed. To install the emulator the UIQ 3.0 SDK (software development kit) needs to be downloaded from the UIQ Technology website. After it is downloaded it needs to be installed in it's entirety. Only installing the Java development portion does not work, because it is not possible to configure the installation when only the Java portion is installed. Next the P990 Extension Package for the UIQ 3 SDK needs to be downloaded and installed.

Finally, Netbeans has to be configured to use the UIQ platform. Go to the Netbeans tool menu and select Java platforms. Click on the add platform link. Select Java ME CDC Platform Emulator and click next, then navigate to where UIQ 3.0 is installed, usually this is C:\Symbian\UIQ3SDK. At this point click next. Netbeans will probably say it hasn't been configured. You need to then go to the configuration tab and click on the SDK configuration tool. On the styles tab select the 990 flip open style. Apply the style and then close the SDK configuration window. Finish the configuration and the development environment is ready.

To setup a project go to file → new project and choose the Mobility Category and CDC application and click next. Name the project and decide where to save it. The last screen should show the platform that was setup and you will be able to click next.

A2. Installing Programs on phone

After you have written code and want to try it out on the 990 phone you will have to install the program on the phone. Before you can install it Netbeans must create a .sis file for you. Running a clean and then build (located under the build menu in Netbeans) will produce a .sis in the dist folder located in the projects directory. Not running clean first will cause the .sis file to be larger then necessary.

An application only has to be signed if it needs to use some of the restricted capabilities of the Symbian operating system. The J2ME edition of the tuple board requires the “NetworkServices” capability. In Netbeans you must explicitly say you are going to use this capability before you build your application. Not specifying this capability means the program will not be able to use the network. Figure 45 shows where the capabilities need to be filled in. This screen is located under the project properties.

For tuple board applications the NetworkServices capability is required. The build process can be be started by simply hitting shift F11 or by going to build menu and building the project. After a successful build a .sis file will be created and saved in the dist folder (On my computer C:\Documents and Settings\Travis\My Documents\NetBeansProjects\War\dist the War folder is my Netbeans project folder). Applications that use restricted APIs must be signed. To sign the application the .sis needs to be uploaded onto the Symbian Signed website. The website at the time of publication could be found at: https://www.symbiansigned.com/app/page/public/openSignedOnline.do

To get the application signed the phone's IMEI (international mobile equipment identity) number is required. This number can be found behind the battery or by entering the following sequence into the the phone *#06#. The signed file will only be valid on the phone with this IMEI number. If
you want to install the same program on a second phone you will have to repeat this signing process and install the proper file on the proper phone. The next piece of information you need is an email address he signed file will be sent to this address. You will then need to browse to and upload the .sis produced by Netbeans. After that step put a check box next to the capabilities that you specified in Figure 45. Read and accept the license agreement. Wait until the Symbian Signed website sends a confirmation email. In the email Symbian will send you click on the link. After you click on the link in the confirmation email they will send another email with a link to download the signed file.

To install the file on the phone the PC suite software that came on the CD with the phone needs to be used. It can also be downloaded from the Sony Ericsson website. The phone can be connected to the computer using the USB cable that came with the phone. The pc suite software will detect that the phone has been connected. Clicking on the application installer icon will allow the signed .sis to be installed on the phone. Follow the wizards directions to install the signed .sis file onto the phone. It should now be possible to run the downloaded program on the phone.

![Figure 43: PC Suite for Sony Ericsson Smartphones](image)
A.3 Building the Card Game Suite

I have combined the War game and Texas Hold 'em game into a game suite. When the suite is launched the first thing it asks the user to do is pick which game they want to play. This appendix will describe the steps that are required to build this game suite. The first step would be to read section A.1 of this document to make sure your development environment is setup.

Before building the application for the first time some project properties need to be setup. Getting to the project properties is done by right clicking on the project and by selecting the properties items as shown in Figure 43.
The first option that needs to be set up is shown in Figure 45. The NetworkServices application capability must be entered so that the device can use WIFI on the phone.
Figure 45: Where you setup application capabilities

Another option that needs to be setup is shown in Figure 46. This option tells the compiler where the entry point for this program is. This allows the game to launch from the correct class.
Now that the NetworkServices capability and the programs entry point are setup the program can be built. To build the program you need to select the Clean and Build option shown on Figure 43. Not selecting clean and build makes the generated .sis used by the phone larger.

To install the program on the phone follow the steps in appendix A.2. To run the program on the phone you have to do one more step. Netbeans has produced a .jar file that can run the program on a personal computer. It does not enter the entry point properly into the .jar file. To update the jar file so it can be double clicked and launched you need to enter this command in the command window. (Cards.jar is the name of the file that Netbeans produced with my setup.)

`jar ufe Cards.jar edu.rit.SecureDeck/GameSel`
Sources


