Report of Personal Project: Implementing mental poker without a Trusted Third Party

Dimitrios Mistriotis
dimitrios.mistriotis@kcl.ac.uk

September 1, 2009

Abstract

The aim of this project is to implement a security related protocol using secure coding techniques and paradigms, mainly in respect with information flows.

The security protocol chosen is a specific reference of Mental Poker, a cryptographic protocol defined in 1979 with many interesting attributes. The reference language chosen is java and it's extension, java with information flows, implemented in Cornell University.

This was proven to be an ambitious project with many different angles and approaches and the whole experience was beneficial and educational, while at the same time layered the foundations for future research and implementation.

Contents

1 Introduction

According to the description of the project [6], the aim of this project is to produce a mental poker implementation, without the use of a Trusted Third Party and then use it as a basis for an implementation of a bigger security-based implementation in a secure language in respect of information flows. The language chosen for this project is JIF. The aim of this project is to measure how easy or difficult it is for an MSc student to write a useful application in such programmatic environment, accustom and evaluate the concepts and the tools available.

2 Mental Poker

2.1 Definition

Mental Poker was originally introduced in the "Mathematical Gardener" by Rivest, Shamir and Adleman [16]. Paraphrasing the original paper:

> "Mental Poker" is a poker played by two players, without a Deck, but by using a communication channel on which they exchange messages. At the beginning of the game, a fair deal of the deck must be made. During the game each player must know the cards that are in his hand, but must have no information about which cards are on other players' hands. While drawing no card should be selected twice. At the end of the game each player must be able to check that none of the other players have cheated.

There are some important elements of the definition such as:

- The absence of a Trusted Third Party (TTP), which means that there is no external authority involved, which will ensure that none of the players tried to cheat.
- That the only way players can communicate is through the exchange of messages, later in the article some properties of the messages are being discussed such as that the messages must be encrypted as well as the properties of the encryption algorithm.

During the initial phases of the project many algorithms of playing mental poker extending the original one where identified, such as [21] or [9]. The one chosen for implementation during this dissertation was the one titled:"Practical Mental Poker without a TTP Based on Homomorphic Encryption" [5]. We chose that particular for a number of reasons such as: It is the base for many other algorithms that actually deprive from this, or offer alternatives to some of it's properties. Therefore someone can implement the application if many other algorithms available, without much mental and programming effort. It has also been studied extensively for it's cryptographic properties and structure therefore it is a much safer choice. Finally a number of projects of implementing this particular algorithm with JIF, such as "Security-Typed Languages for Implementation of Cryptographic Protocols: A Case Study" [3] exist, offering a basis for comparison and measurement.

Except mental poker there are many other applications for these family of protocols such as other card games, games that require random numbers and random number generation from a predefined set of numbers, without the need for a Trusted Third Party orchestrating and

coordinating the process. This makes them a very nice example of producing an application where the study of individual information flows and it's security implication is being study and this is the main reason that many JIF projects decide to do an implementation based on a Mental Poker algorithm.

2.2 Assumptions

In addition of the choice of the algorithm some further choices have to be made such as the most appropriate crypto-system. In section 3 of the algorithm description [5] it is stated that:

> ... Permuting (i.e. shuffling) encrypted cards requires encryption to be homomorphic, so that the outcome of permuting and decrypting (i.e. opening) a card is the same that would be obtained if the card had been permuted without prior encryption (i.e. reversal) ...

After referencing the relevant literature, such as [15] or [12], as well as other implementations, the most appropriate algorithm was **Elgamal**. A proof of Algorithm's homomorphism is here: [7].

Elgamal is the prime example of homomorphic encryption a stated in [8], used in other relevant projects and had enough documentation and implementations available.

An implication of this choice is the fact that Elgamal produces for each cleartext number m a pair of numbers, which as a result that different data structures should be used for encrypted messages, increasing the code size significantly.

2.3 Existing Work

- At the moment of writing this report there are some projects with the same concept Askarov's and Sabelfeld's implementation of Mental Poker in JIF [4],
- A Toolbox for Mental Card Games [14] implemented in C++.

2.4 Data Structures

In order to implement the mental poker algorithm some data structures specific for this certain application are introduced:

- Distributed Notarization Chain 2.4.1
- Card Vector Representation Section 2.4.2
- Card Permutation Matrix 2.4.3

2.4.1 Distributed Notarization Chain

This is an expansion of the concept of the Lamport Password Chains [11], where the expansion takes into account the existence of more than one entities, in this case the different mental poker players. DNCs are used in order to have each player to "sign" each of her actions so that if evaluation needs to occur a path can be constructed that appoints which player did what. In case of tampered or corrupted data, the player responsible for the event can be pinpointed since the revealed data at the end of the game will give different signatures than

the ones originally supplied. Every chain link m_k consists two elements: The Data Field D_k and the Chaining Value X_k.
Every Data Field further consists of three subfields:

- Timestamp (T_k),
- Concept (C_k) with the information that the link contains and
- Attributes (V_k) with the relevant to C_k information.

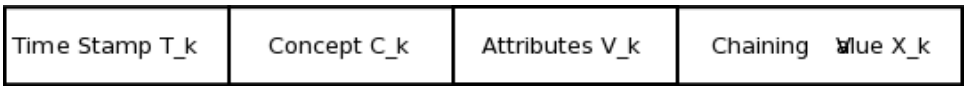

Figure 1: fields of a DNC

Addition of a new link to a chain can be done by each player individually since the player needs to use it's own X_{k-1} for signing, therefore the operations in the mental poker can be done in parallel. For the same reason the players do not have to accept the links in the same order with one another: they can only watch and check each signature in comparison with the previous link.
When there is the need for players to synchronize a special "Chain contraction" link is being computed by a certain entity (which in mental poker's context will be the Croupier player). This is a mark that a milestone has been reached and the players can continue to an another part of the algorithm.

Illustrated in the following example:

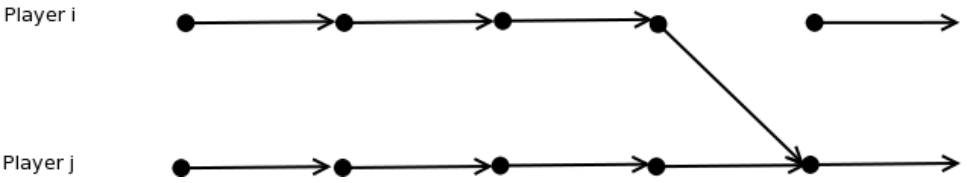

Figure 2: example of a DNC, here player j computes a concatenation link

The following Concepts are being used in this implementation:

Concept	Phase Used
z_i	Initialization 2.5.2
Permutation Commitment	Initialization 2.5.2
D-set	Initialization 2.5.2
E-set	Initialization 2.5.2
Encrypted Deck	Initialization 2.5.2
First chain contraction	Initialization 2.5.2
w_i Card vector representation	Card Draw 2.5.3
$w_i^{'}$ Encrypted Card vector representation	Card Draw 2.5.3

2.4.2 Card Vector Representation

Definition

Central in the mental poker algebra is the notion of the card and it's representation υ. In order to represent a card under this implementation, we need the total number of cards in the deck, **t**, and a prime number **z**. Each player p has her own prime number z_p. Each card is represented as a vector:
$\upsilon = (a_1, a_2, ..., a_t)$

In each vector υ exists an **i**, $1 \leq i \leq t$, for which there is only one a_i for which $a_i \pmod{z_p} \neq 0$. For that card vector, the value of the card represented is **i**.

Example

If $z = 7$ and $t = 4$ (player's prime number z is seven and deck has four cards), then the vector:
$\upsilon = (42, 21, 28, 50)$,
represents the card with value 4 ($i = 4$). That's because only for $a_4 = 50$, holds that $a_4 \pmod{z_p} \neq 0$ ($a_4 \pmod{mod z_p} \neq 0 \Rightarrow 50 mod \pmod{7} \neq 0 \Rightarrow 1 \neq 0$). For all the other $i \neq 4$, $a_i \pmod{z_p} = 0$ ($42 \pmod{7} = 21 \pmod{7} = 28 \pmod{7} = 0$).

Implementation

For the implementation of the Card Vector Representation the following abbreviated java code was compiled:

```
public class CardVectorRepresentation {
    BigInteger[] uRepresentation; // Castella-Roca paper Definition
        1
    /* ... */
}
```

Note the following implementation details:
uRepresentation: is an array of Big Integers storing the actual a_i values.[1]

Constructors: A Card vector can be instantiated in three ways:

- Construct a card based on it's value sing the prime number z provided.
- By deserialisation from a string, which might originate from network communication and
- as a the product of a card with a Permutation matrix (see Permutation Matrix Algebra) 2.4.3.

2.4.3 Card Permutation Matrix

Definition

[1] Since the protocol requires very big numbers with a large number of digits, "Big Integers" (in java instances of the *BigIntegerer* class) were used

A deck of **t** cards is projected into a Permutation Matrix, which is a $t \cdot t$ matrix.

$$\Pi = \begin{pmatrix} \pi_{1,1} & \pi_{1,2} & ... & \pi_{1,t} \\ \pi_{2,1} & ... & ... & ... \\ ... & & & \\ \pi_{t,1} & \pi_{t,2} & ... & \pi_{t,t} \end{pmatrix}$$

where each i^{th} row of Π is a card $\pi(i)$.

Example

Suppose a permuted deck with four cards: $\pi = (4, 2, 3, 1)$ and that player's chosen prime number z is 7. A possible Permutation Matrix is the following:

$$\Pi = \begin{pmatrix} 21 & 14 & 28 & 51 \\ 42 & 19 & 7 & 35 \\ 36 & 49 & 42 & 14 \\ 35 & 28 & 44 & 7 \end{pmatrix}$$

The first row represents card 4, because the 4^{th} element is the only one for which $\pi_{1,4} \pmod z \neq 0$ or $51 \pmod 7 = 3 \neq 0$, therefore $\pi(1) = 4$. For the same reason $\pi(4) = 3$, since the 3^{rd} element of the fourth row, is the only one for which $44 \pmod 7 = 2 \neq 0$.

Permutation Matrix Algebra

The properties of the Permutation Matrix and the Vector representation of the cards allow by construction the matrix multiplication of a vector card, (an $q \times n$ array) representation with a permutation matrix (an $n \times n$ array). The result is a card with a different value.

If for example we calculate the card from the example 2.4.2 with the previous 2.4.3 permutation matrix we have:

$$\upsilon \times \Pi =$$

$$= (42, 21, 28, 50) \times \begin{pmatrix} 21 & 14 & 28 & 51 \\ 42 & 19 & 7 & 35 \\ 36 & 49 & 42 & 14 \\ 35 & 28 & 44 & 7 \end{pmatrix} =$$

$$= (4522, 3759, 4699, 3619)$$

The result Vector representation equals with the card with value 3, since the third element is the only one non-zero mod z_i (which is seven).

Equivalent Card Permutation Matrix

For each player the equivalent person matrix towards another player, is a Permutation Matrix that has the same card values, but is being calculated using the other player's prime number z. Rephrasing from the [5]:

$\Pi = \{\pi_{i,j}\}$ is equivalent to $\Pi^{'} = \{\pi^{'}_{i,j}\}$ **iff**
for each $i, j \in \{1, ..., number of cards in deck\}$:

$\pi_{i,j} \pmod z \neq 0 \iff \pi'_{i,j} \pmod{z'} \neq 0$ and $\pi_{i,j} \pmod z = 0 \iff \pi'_{i,j} \pmod{z'} = 0$

Equivalent Permutation Matrices are being used in the Card Draw phase of the protocol 2.5.3.

Implementation

```
class CardPermutationMatrix {
    private BigInteger[][] permutationMatrix = null;

    CardPermutationMatrix(VectorDeck inputVDeck) { /* ... */ }

    CardPermutationMatrix(Deck permuatedDeck, BigInteger zI) { /*
       ... */ }

    public CardPermutationMatrix getEquivalentPermutationMatrix(
       CardPermutationMatrix myMatrix, BigInteger myZ,
             BigInteger otherPlayerZ) { /* ... */ }

     public void modifyRowNonModuloZ(int row,BigInteger primeZ,
        SecureRandom rand) { /* ... */ }
    /* ... */
}
```

Again a two dimensional array of Big Integers is being used for internal representation, while all the actions described above on the matrix have their appropriate implementation.

2.4.4 Delta and Epsilon Sets

Definition

Two more special types of sets are used in Mental Poker, δ, ϵ, D and E. Those are defined for each player i as follows:
First a value s, such as $s > t$ is decided (t is the number of cards in the deck)
δ is a set of s numbers such as for each number $n \in \delta$:

$n \pmod{z_i} = 0$, where z_i the z of player i.

In the same fashion, ϵ is a set of s numbers such as for each number $n \in \epsilon$:

$n \pmod{z_i} \neq 0$, where z_i the z of player i.

D is a set generated from δ, with the key of player i, K_i, each d of D is the encrypted counterpart of δ:

for j element in δ, $d_j = E_{K_i}(\delta_j)$.

Exactly the same for the E set (for j element in ϵ, $e_j = E_{K_i}(\epsilon j)$.)

Example

Assuming that player's i prime number is 7 and we have 5 cards in deck, the following is an example of δ_i:
$\delta_i = [56, 35, 63, 21, 28, 42]$

The set contains of six numbers (one more than the deck size) all of them equal to zero mod 7. Similarly an ϵ set of the same player would be like:

$\epsilon i = [71, 9, 17, 30, 51, 37]$

Implementation

```
class LcaseDeltaSet {
    protected BigInteger[] numCollection;

    LcaseDeltaSet(BigInteger primeZ) { /* ... */ }
}
```

As we see δ is represented as an array of Big Integers. ϵ class inherits it.

```
class DeltaEpsilonSet {
    MPEncryptedMessage[] BigDelta;

    DeltaEpsilonSet(LcaseDeltaSet lDelta, MPElGamal cryptoSystem) {
        /* ... */}
}
```

Similar to the δ implementation, D and E share the same code since both represent encrypted Big Integers.

2.4.5 Elgamal

Although not a data structure, a custom implementation of Elgamal's cryptosystem had to be coded. The rationale behind this decision has to do with the non-availability of the appropriate libraries in the JIF environment. Therefore there is no other way to do so without a custom implementation. The algorithm is an implementation from the pages 86 to 88 of [15], taking into account secure coding techniques such as those described for coding in cryptography in [10]. Also some similar implementations were taken into account such as [1]. A proof that Elgamal has the desired mathematical properties is here:
`http://www.cs.ucla.edu/~rafail/TEACHING/WINTER-2005/L8/L8.ps`

Implementation

```
public class MPElGamal {
    public MPKeyPublic getPublicKey(){ /* ... */ }

    public MPKeyPrivate getPrivateKey() { /* ... */ }

    public BigInteger getZPlayer() { /* ... */ }

    public MPEncryptedMessage encrypt(BigInteger message) { /* ...
       */ }

    public BigInteger decrypt(MPEncryptedMessage mpEnc) { /* ... */
        }

    public String signString(String message) { /* ... */ }

    public MPSignedInteger sign(byte[] hashedMessageByte) { /* ...
       */ }

    public static void main(String args[]) { /* ... */ }
}
```

The algorithm supports the encryption of a Big Integer into a pair of Big Integers as the algorithm implies, while the decryption is also supported. Additionally this is the place were the Elgamal capability to sign a message is utilized as well as the management of each player's private prime z_i. The main method is used for testing since a number is encrypted and decrypted in order to compare the results.

2.5 Algorithm Description

The two steps of the algorithm with diagrams and State Charts. Describe end expand from the Garcia-Loca paper what we did, quoting their text and explaining why of each step and how it contributes to the result.

2.5.1 introduction

The implemented protocol consists of two parts:

- Initialization 2.5.2, and
- Card Draw 2.5.3

Players communicate **only** through messages 2.1, which in this algorithm is Distributed Chain Links.

2.5.2 Initialization

In the Initialization phase of the protocol players initialize some data structures which will be used later in the game, and broadcast the results of the computation to the other players.

When the result of the computation should be known to other players, then it is transmitted "as-is". In the case that the result should be kept secret, a hash-signature is transmitted as the attribute of the DNC chain.

More specifically:

1. Each player i generates an initial permutation of the Deck and a Secret Key K_i.
2. Each player i generates a large prime z_i, which is broadcasted to the rest of the players as a DNC-link.
3. A Card Permutation Matrix (Π_i) is generated out of the initial permutation from each player. Now a commitment of Π_i is broadcasted as a DNC-link. The player will also store the commitment for future evaluation
4. Each player will compute two sets: δ and ϵ 2.4.4. Out of them the **D** and **E** sets will be produced. **D** set will be broadcasted as a DNC-link.
5. Then **E** will be broadcasted as a DNC-Link 2.4.4.
6. A vector representation of the deck is being generated and then encrypted from each player.
7. The encrypted deck is being permuted and the next DNC-link containing the encrypted deck.
8. Finally a player selected acting as group croupier [2] computed the first DNC-link contraction and broadcasts it.

A croupier broadcast, signals the end of the initialization phase.

2.5.3 Card Draw

Card draw is being initiated whenever a player decides that a hard should be drawn on his behalf [3].

Whenever a player decides to draw a card the following process [4] occurs:

1. a u_0 number is chosen, such as $a \leq u_0 \leq t$ and u_0 has never been used before. Player requesting a card (PL_i) computes a w_0 representation of the card with value u_0 and broadcasts a w DNC-link.

[2] In our implementation the first player is the croupier.

[3] Card Draw is simplified since we run a simulated version of the game where two players can not decide that they want a card at the same time.

[4] the process has some minor modifications than that in the referenced algorithm, therefore stated here.

2. The first player (PL_1) receives the link and computes $w_1 = w_0.\Pi_1^{'}$, as described before 2.4.3 where $\Pi_1^{'}$ is PL_1's equivalent permutation matrix of player that requested the card in the previous step. w_1 is broadcasted in a w DNC-link.

3. each player PL_j before i in the list responds with a w_j DNC-link calculated as in the previous step.

4. When player i receives a w_{i-1} DNC-link does the following:

   - Permutes w_i using it's own Permutation Matrix (Π_i),
   - Modifies the m-th row of Π_i, where m the value of the recieved DNC-link
   - choses encrypted card $w_i^{'}$ which corresponds to w_i from the link
   - Broadcasts $w^{'}$ DNC-link

5. Players "after" player requesting card after PL_i until the last one do the following:

   - Using the prime number z of player i (z_i) and the published key of the same player, computes an encrypted version of $\Pi_j^{'}$, named Π_j^c which involves as well the usage of the D and E sets computed in the initialization phase (the process here is exactly the same with [5] and therefore not mentioned here).
   - a $w_j^{'}$ is computed using the equivalent multiplication of $w_{j-1}^{'}$ with Π_j^c. The result is being broadcasted as a $w^{'}$ DNC-link.

6. When player i receives the broadcasted $w^{'}$ DNC-link from the last player, decrypts the link with it's private key and gets the value of the card. Card Draw finishes here.

2.6 Implementation of Mental Poker in Java

Following the advice in [3], the algorithmic part of the mental poker was implemented abstracting the communication channels between the various entities.

Initial versions of the algorithm included a communication mechanism that coordinated the different mental poker players, without the use of threads, implementing a round robin policy on broadcasting 6.2. This was abandoned in order to focus on the algorithmic aspects of Mental Poker, simplifying the development. A demonstration video and the source code are provided for future reference.

In the current version each mental poker player is represented as a separate class with it's own private data. Communication between players is being handled by the main(...) method of the program. Since players can communicate only through Distributed Notarization Chains, which additionally should be broadcasted, each players "next"-DNC is being requested and then passed as parameter to all players (including the one that broadcasted it). With this implementation we emulate a no-fault channel. In a real-world implementation this should be done by a dispatcher process from the host of the game, or in players reside inside a LAN (as in the initial December 08 implementation), through broadcasted UDP packets.

Having the data structures compiled, the java implementation looks more straightforward as the algorithm is "mapped" to java equivalents using object orientation programming techniques. Each player is being represented by an instance of a Player class, while the "communication" is abstracted in the main method. Also for the needs of the protocol each player

should know it's place in the game, therefore either in the distributed or in the centralized implementation players are being assigned "positions", so that they can respond to appropriate DNC broadcast links.

Because each player must remember at every given moment in which part of the initialization or the card draw protocol is, a state machine approach was used, in order to formalize the code. Each player stores it's current state in a enumerated structure.

2.6.1 Initialization States

"State-wise" Initialization is simple since players work in parallel, without co-ordination just broadcasting the results of their computation. Only deviation is at the end of the phase where one player, the Croupier (in this implementation it is assumed that this is the first player), will create a contraction link. So players move from one state to the next while one of them will go to one more.

For initialization we have:

```
public enum InitializationState {
    START,
    ZIBROADCAST,
    PERMUTATIONMATRIXBCAST,
    UCASEDELTASETBCAST,
    BIGEPSILONSETBCAST,
    VECTORDECKBCAST,
    PLAYERCROUPIERBCAST,
    END
}
```

Where the last one (end) signals the transition to the card draw phase.

2.6.2 Card Draw States

Card Draw is different from Initialization because players have different roles in a draw. As described in the protocol 2.5.3, we have:

1. The player requesting a card (assume PL_i),
2. Players before (PL_j, where $j < i$) and
3. Players after (PL_j, where $j > i$)

Player Requesting Card

Modeling the protocol produces the following diagram:

After a player requests a Card, waits for the appropriate DNC-link, while storing the ones before it for verification purposes, when the appropriate DNC-link is received then after the computations an encrypted vector card is broadcasted. After that the player goes again into recording DNC-link broadcasts until receiving one from the last player. This signals the end of Card Draw phase, so player returns back to being idle.

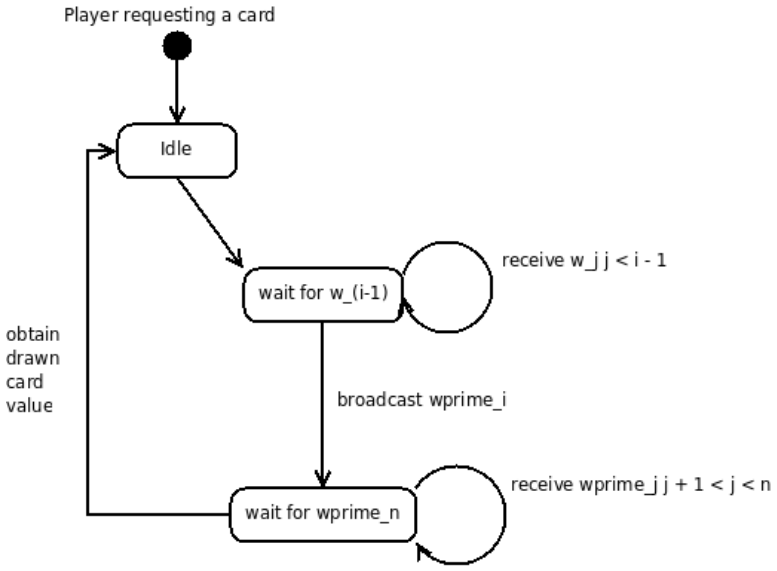

Figure 3: State Diagram of player drawing Card

Player Before Player Requesting Card

Modeling the protocol produces the following diagram:

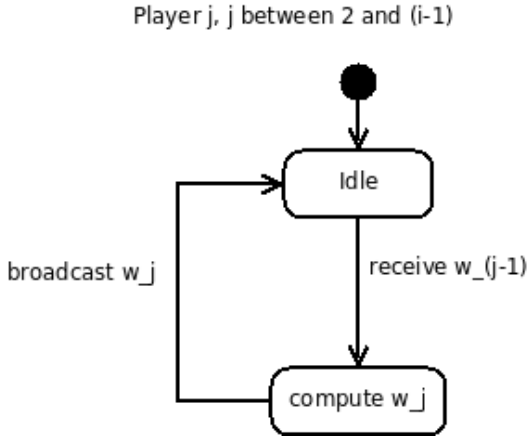

Figure 4: State Diagram of players before the one drawing a card

Players before the one requesting a card wait for their previous one to broadcast the appropriate DNC-link 3 and then they respond with theirs. After that they return back to being idle.

Player After Player Requesting Card

Modeling the protocol produces the following diagram:

Players who are following the one requesting a card wait for their previous one to broadcast the appropriate DNC-link 5 and then they respond with theirs. After that they return back

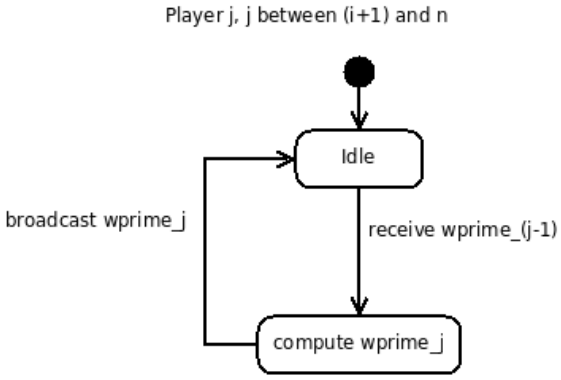

Figure 5: State Diagram of players after the one drawing a card

to being idle.

So this makes the Card Draw phase more complicated, since a player has to decide in which state to switch next. The states for Card Draw are the following:

```
public enum CardDrawState {
    PLAYERIDLE,
    PLAYER1RCVW0,
    PLAYERJRCVJMINUSONE,
    PLAYERJRCVWJPRIMEMINUSONE,
    REQUESTCARDWAITWMINUSONE,
    CARDREQUESTWAITWPRIMEN,
}
```

3 JIF

3.1 Introduction to JIF

JIF [19] according to it's home page is:

> ... a security-typed programming language that extends Java with support for information flow control and access control, enforced at both compile time and run time.

JIF is different than other similar programming language concepts in the way that the checks of information flow can occur both at compile time and at run time allowing a more flexible modeling of information flow. Theoretically this leads to simpler implementations which are also more flexible since they can solve more problems unsolved by other paradigms which deploy static, compile-time only flow checks. This is being achieved by implementing the Decentralized Label Model 3.2, discussed in the next section.

3.2 Decentralized Label Model

In the relevant paper of Andrew C. Myers [2] in contrast with other traditional models. Briefly we have inside the code the concept of a Principal, who generally represents an entity or a user on which acts behalf of. Values 3.2.1 have attached Labels which define the access that different principals have to attached values. Each element as well of the Decentralized Label Model as well as the interactions among those elements are being described in the following sections.

3.2.1 Values

Values represent classes or basic types of the programming language. Additionally to the traditional model we have the "input channesl" and the "output channels". "Input channels" represent points of entry of information in the program, and therefore can only be read. "Output channels" represent points where information is leaving the program, therefore can only be written.

3.2.2 Principals

Principals represent users or groups of users that interact within the system. Each value belongs to a principal, as we will see with the label concept in the following section 3.2.3. The additional feature that JIF has in comparison with other paradigms is the capability of a principal to delegate it's authority to others, with *acts-for* relationships. By default there are two principals available the "noone" () and the "everyone" the former "acts-for" any principal, the latter is "acted-for" by any principal.

3.2.3 Labels

Policies are implemented in JIF with labels. Each label constitutes of two sets of Principals, the reader set (those who can write-to a value) and the writer set (those who can write-to a value). The union of all principals in the reared set constitutes the effective reader set, which represent all principals that can read a value, while the intersection of the principals of the writer set, constitutes the principals that can write to a value.

These concepts allow two interesting actions in code and therefore in run-time level: the declassification and the restriction. With the declassification the readers set (and therefore the effective readers) can be extended, allowing more principals to write to the value. While with a restriction the restriction removes readers and/or adds owners. With these actions policies can be implemented inside the code at run time as well as compile-time (which is the only way in other relevant paradigms). The additional ways to define and implement a policy constitute the additional capabilities of JIF and theoretically lead to more efficient and easy to read (and therefore evaluate) code, in order to detect implicit and explicit flows.

3.2.4 Relabeling

A question rises on how declassification occurs internally. This is being done with the concept or re-labeling: a new value with a different label is being generated and the contents of the previous value are being copied to the new.

3.3 Implicit and Explicit flows

JIF also aims to solve the issue of explicit and implicit information flows:

- we have an **explicit flow** when information flows from a more restrictive to a less restrictive label,
- **implicit flows** occur when mixing variables of two different information classes and contents of one can be assumed from the values of the other.

Example of an explicit flow from Carnegie Mellon Information Security classes[5]:

```
public class SecretMessages[principal alice, principal bob]
{
 String{alice:} aliceInstructions;
 String{bob:} bobInstructions;

 public SecretMessages(String{alice:} ai, String{bob:} bi) {
 aliceInstructions = ai;
 bobInstructions = bi;
 }
 public String{bob:} leak() {
 bobInstructions = aliceInstructions;
 return bobInstructions;
 }
}
```

Example of an implicit information flow from [2]:

```
x = 0;
if b
{
    x = 1;
}
```

Here from the value of x, we can assume implicitly information about b.

3.4 Decentralized Label Model in JIF

In this section by presenting some working code or syntax examples, the way the theory behind the Decentralized Label Model will be illustrated and explained.

[5] http://www.ece.cmu.edu/~ece732/lectures/18732-jif.pdf

3.4.1 Example: Variable Declaration

First example is a variable declaration from [19]:

```
int {Alice: Bob} x;
```

Value x, an integer is owned by principal Alice. Principal Bob can read it.

3.4.2 Example: Declassification

Second example, is an example of a declassification 3.2.3 which leads to an output to screen[6]:

```
PrintStream[{}] output = declassify(runtime.stdout(new label{})
    , {});
if (output == null) return;

int{Alice:} iAlice = 3;
int aliceDec;
aliceDec = declassify(iAlice, {});
output.println("aliceDec: " + aliceDec);
```

This is in a way both a "Hello World" code example as well as a declassification one. First an output channel is acquired label with the less restrictive label {}. Then the *iAlice* variable (named that way to identify an integer that belongs to principal Alice), is declassified to the less restrictive principal as well. Note that this is being done through relabeling 3.2.4. After that, the print-line call is legit and can be executed.

3.4.3 Example: Array Handling

As stated in [4] arrays are different in JIF due to the need of haivng two labels, one for the elements of the array and one for the array itself. This is being done in order to avoid the so-called "laundering attack". This example is from the JIF-exercises [17]:

```
String {L}[]{L} larr;
```

Here *larr* is an array of Strings where both (array and elements) belong to label L.

3.4.4 Example: Classes, Method signatures and Exceptions

[6]Code was created as part of this report, is publicly available and can be located here: `http://stackoverflow.com/questions/1037635/java-with-information-flows-output-to-screen`

Apart from the method signatures, JIF code is generating a large number of Exceptions that should be handled inside the procedure or thrown. In this example we have a setter, which in a casual "setter" procedure.

As we see in this figure from [2]:

$$
\begin{array}{rcl}
\textit{procedure} & \rightarrow & \textit{id} \left[\textit{authority} \right] \\
 & & (\ \textit{arguments}\ ) \\
 & & \left[\textbf{returns}\ (\ \textit{id} : T_r\ \{\ L_r\ \}\ ) \right] \\
 & & \textit{body}\ \textbf{end} \\
\textit{authority} & \rightarrow & \ll \textbf{caller} \gg \\
\textit{arguments} & \rightarrow & a_1 : T_1 \left[\{L_1\} \right], \ldots, a_n : T_n \left[\{L_n\} \right]
\end{array}
$$

Figure 6: Procedure Syntax definition in JIF

The above concept is illustrated from an example from the JIF-exercises [17], as well:

```
public void setAt{L}(int {L} i, String {L} s):{L}
throws ArrayIndexOutOfBoundsException, ArrayStoreException,
   NullPointerException {
    /* Code Ommited */
}
```

3.5 Criticism of JIF

As with every programming concept or tool in the software sphere, JIF is not immune to outside and inside criticism. Critique can be basically summarized into two positions:

- "JIF is too immature" as well as "JIF will never happen" [7] and
- "JIF provides a false sense of security" which usually parts with "JIF is expensive"

For the rest of this section these two arguments will be briefly presented.

The "immaturity" argument has to do with the learning curve a programmer needs to be productive in writing effective code in JIF. This happens for two reasons, which together create a chicken and egg situation. Apart from JIF itself programming with information flows into account is by definition hard, since first some additional concepts have to be mastered. Some, but not all, of them are the explicit and implicit flows, the concept of the side-effects, the high and the low level of the flow, the Principal algebra, etc. For this reason it is difficult to introduce these concepts into inexperienced or not too experienced programmers (they first need to know how to program and then on top program in some concepts), moreover amateurs or self-taught professionals. This more or less drives to the situation that in order for someone to engage into such programming experience, MSc level is required. This results to small numbers of potential community by itself. The situation does not improve with

[7] in the "IPv6 will never happen" concept

the landscape of the current toolset which is inconsistent and has different stakeholders with different aims and desire for participation.

Hence if we had more people interested, there would be a bigger demand for quality tools or tighter integration. Similarly a better programming experience would attract more developers or generally people interested since the barrier of entry would be lower. This is a classic example of a vicious circle. The only way to break such a circle is by introducing big investments financially. The situation in this space is beyond the aims of this essay.

Another "immaturity" argument has to do with the fact that some concepts in order to immature needed more than 15 years of existence. The examples are numerous, the more obvious are version control, where many good tools were available in the early 1990's or Object Orientation, which was available many years before it became mainstream. Perhaps JIF is still in that incubation period, or does not yet have the critical mass it needs in the secure software development community.

About the "false sense of security" argument: Since the effort needed to become productive is as described above, it is very easy for a developer/project leader to fall into the fallacy of believing that a correct information flow modeling is a panacea that will solve all the project's security issues. Although this initially seems like an exaggeration, it is easy to fall into this category and ignore other aspects of a security related project, as the correct choice and implementation of crypto-systems or other secure coding approaches and practices. It is very easy to dwell into an information flow approach, something that JIF does not discourage with the limited availability of ported programming libraries from the Java Runtime Environment. An obvious example is the custom implementation of a cryptographic algorithm from this project, something that can be considered a classic secure programming "mistake" (duplication of effort, use of an unproven algorithm, etc).

3.6 Tools

In this section a small review of tools available for creating programs in the JIF environment, at the date of writing this report is being presented.

Jif Distribution: The main distribution from [19] offers the main compiler and the runtime environment. There are two issues here:

It is a source-only distributions and some time and familiarity with java compilation processes are needed in order to make JIF usable and some custom tweaks of the host system. The whole process is described in an Appendix of this report 6.1, since such a tutorial is not available. Different versions have different features and different level of maturity, considering bugs and capabilities. Therefore concerning the needs of the particular project, the choice must me made on the version that will be used. Also IDE (Eclipse) support is only available into one of them.

JIF Eclipse [20]: JIF eclipse is a plugin for the popular eclipse platform. It assists the programmer with the creation and manipulation of programs, as well as pre-compilation analysis so that the programmer can immediately see an error before compilation, saving development time. It also offers suggestions which may help fixing such issues.

SIGGEN:Quoting from it's homepage [8], *"Siggen can automatically generate signatures for your Java and Jif files. It does not examine the bytecode to provide labels that match the security behavior of a library function"*

3.7 Learning JIF

For the course of this project the following JIF learning material was identified:

- **JIF's reference manual** [18]: Which has reference material on each version.
- **JIF Examples**: A set of examples like the well known Battleship game are included in the compiler package.
- **JIF Exercises** [17]: From the Chalmers University a series of exercises and tutorials demonstrated various concepts.
- **Blog entries**: Some relevant references on internet: `http://www.napes.co.uk/blog/decentralized-label-model/`
- **other projects**: In the course of this project another MSc project was identified on implementing a "real-world" application in JIF, particularly a model of the French health care system [13].
- **JIF mailing list**: A low-activity mailing list by Cornell University.

Generally the learning material available falls into two categories: First there is the material available inside academic papers submitted to conferences, publications etc. This generally emphasizes to the aspect that the author of the paper wants to for each relevant subject, without getting into the general context. Then there are some large scale implementations, such as Civitas [9], or Fabric [10].

The problem for the newcomer to the field is that there is no "middle ground" material available to aid in the course of getting the elementary concepts and combine them to a working application.

4 Mental Poker in Jif

4.1 Introduction

This project originally aimed at producing a mental poker implementation in Java with Information Flows. As it will be discussed later, this was extremely difficult to produce any usable result in the course of this project. In this section a summarization of the effort done will be presented as well as the material produced and the accompanying source code.

[8] http://www.cse.psu.edu/ dhking/siggen/

[9] Civitas: Toward a secure voting system. In Proc. IEEE Symposium on Security and Privacy, pages 354368, May 2008.

[10] "Fabric: A Platform for Secure Distributed Computation and Storage", Department of Computer Science, Cornell University

4.2 Methodology chosen

The methodology chosen to implement this project was inspired from [4], which is a paper based precisely on the experience of implementing the mental poker in JIF, as well as [13] which describes and suggests a similar paradigm. The first objective was the production of working code in java, so that the programmer can be accustomed with the protocol and the concepts around it. At some point in parallel, training in JIF takes place. After both of those have finished the programmer is capable of implementing the given algorithm in JIF. Different approaches lead either to very lengthy development times or just to project failure.

But even in this case, the workload can not be considered easy. Quoting:

> ... The baseline implementation consumed around 60 man-hours of development work. The JIF implementation and distributed JIF implementation consumed 150 and 80 man-hours respectively, excluding the time to learn JIF. The case study indicates that although lifting Java code to JIF takes some experience to master, the security-typed result is not significantly distant from the original code. Further- more, we have developed patterns for secure programming (cf. Section 5) to make programming with security types clearer and more convenient. ...

What is missing from the quoted text is an estimation of "time required to learn JIF" and more importantly for this project, "how" can someone learn JIF. It appears from anecdotal evidence that usually before someone is assigned to a similar project, a MSc level semester course in secure programming with information flows is provided, usually with coursework and laboratory exercises in JIF. The process of self-teaching JIF, which was followed on this project in addition with the level of the educational available material 3.7, made the process harder which some times resulted to a trial and error approach with minimum, at the best, results.

4.3 Annotations

In the introductory paper to JIF [3] there were two motivational examples (a bank application and a medical research one). In those examples a flow annotation was used to illustrate the flow of information on the various principals.

In the examples provided the following annotations are being used:

Symbol	**Usage**
Circle	Principal Within the system
Arrows	Information flow between principals
Square	Information flowing or Database
Double Circle	Trusted Agents that declassify information

In the first example 4.3 we have the most simplified version of Mental Poker initialization phase among two players, Alice and Bob. Alice is assigned the role of the Croupier, therefore at the end of the initialization phase, she has to endorse Bob's contraction chains and produce a contraction link. That link has to be declassified and send back to Bob (which at this example represents all other players).

The following step was to generalize this figure, into a multiplayer scenario, utilizing more JIF capabilities. Players mean of communication Distributed Notarization Chains 2.4.1, we wanted each player to have a coordinated picture of the game with each of the other mental

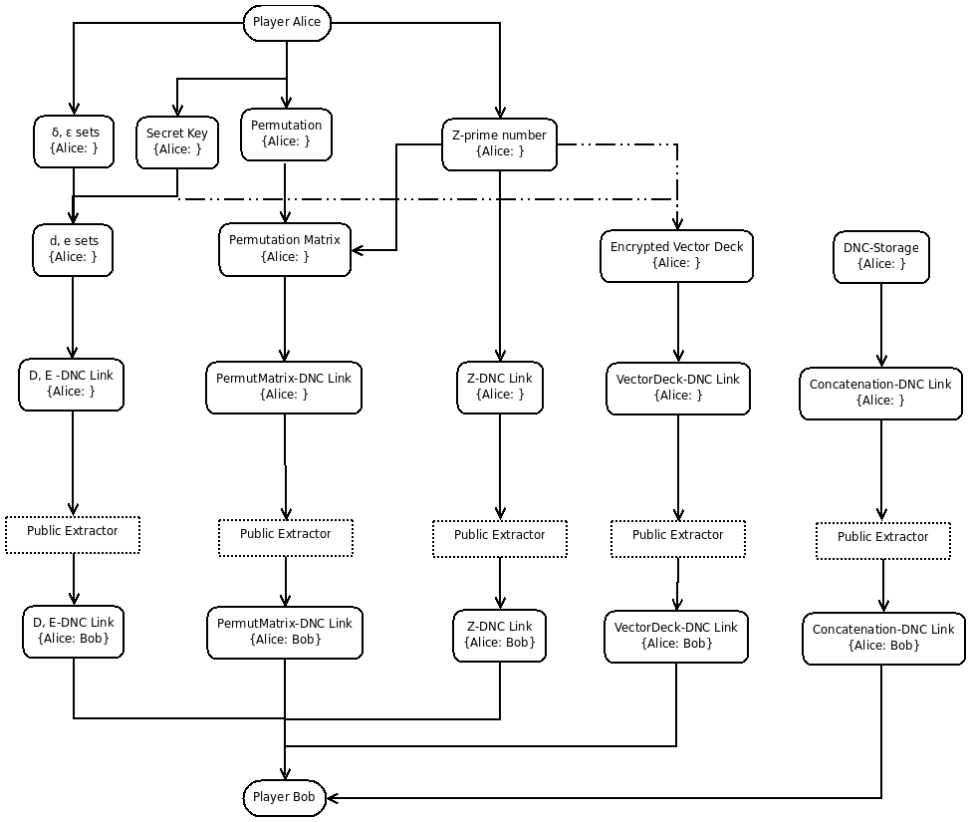

Figure 7: Mental Poker between Alice and Bob

poker players represented as a principal within the system. So each player j, would hold an array of Players i, where $i \neq j$. Each of them would hold their DNC-chain and would release what information is needed to current player j. In order for this to happen each DNC-received link, from an Input Channel 3.2.1 has to be identified, checked for validity and then elevated to be assigned to player's i DNC-chain. The only way to do this, as we see in figure 4.3 is by having an input channel with the lowest principal, {*}, and then direct the input to an Extractor which will elevate it to higher levels. Then when the current player (Alice) needs a specific piece of information, it can be declassified and assigned.

4.4 Implementation

Some examples of implementing parts of the mental poker protocol in JIF will be presented here. This will help to demonstrate the programming experience, the concepts and some language idiosyncrasies.

4.4.1 Uplifting a Java class

This is one way of porting an already existing java library to JIF. By supplying only the signatures, after compilation, the runtime environment can map the JIF signature to a java one and execute it as desired. Please note that there is no check if the implementation maps to existing code and that some times the compilation process is rejected. Here we can see the

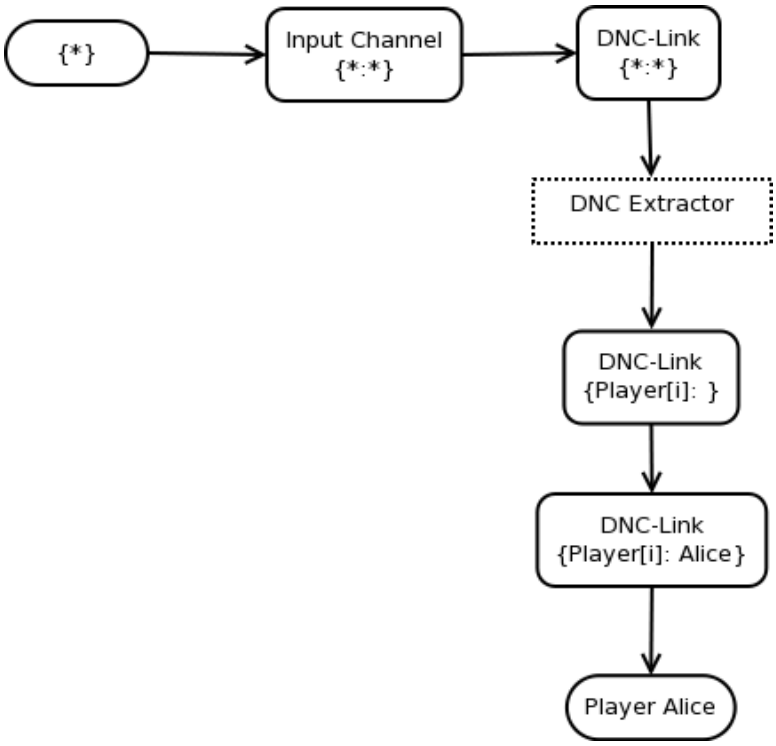

Figure 8: DNC link extraction

"{this}" principal, which implies the current class status. Also the line:

private static int __JIF_SIG_OF_JAVA_CLASS$20030619 = 0;

Which accordinf to JIF's creators [18],

> ... is a hack to let the JIF compiler to know that the class provides a JIF signature for an existing Java class.

```
package java.util;

public
class Random /*implements java.io.Serializable*/ {
    private static int __JIF_SIG_OF_JAVA_CLASS$20030619 = 0;

    /** use serialVersionUID from JDK 1.1 for interoperability */
    static final long serialVersionUID = 3905348978240129619L;

    public Random() {}

    public Random(long{this} seed) {

    }
```

```
    synchronized public native void setSeed(long{this} seed);

    public native void nextBytes(byte{this}[]{this} bytes);

    public native int{this} nextInt();

    public native int{this;n} nextInt(int n);

    public native long{this} nextLong();

    public native boolean{this} nextBoolean();

    public native float{this} nextFloat();

    public native double{this} nextDouble();

    synchronized public native double{this} nextGaussian();

}
```

4.4.2 Porting a Java class

In this section an example of porting a java data structure to it's JIF equivalent will be provided along with some commendation on the programming experience. The class chosen is a simple data structure that holds three big integers:

```
class MPKeyPublic {
    public BigInteger p; //El Gamal's p, prime number
    public BigInteger g; //El Gamal's g, random number less than
    public BigInteger y; //El Gamal's y, y = (g^x)mod(p)
}
```

Can only be ported to JIF in the following way:

```
import java.io.PrintStream;
import java.lang.Object;
import java.math.BigInteger;

class MPKeyPublic[label L] {

  private static int __JIF_SIG_OF_JAVA_CLASS$20030619 = 0;

  public BigInteger {L} p; //El Gamal's p, prime number
  public BigInteger {L} g; //El Gamal's g, random number less than
     p
```

```
  public BigInteger {L} y; //El Gamal's y, y = (g^x)mod(p)

  public void setP{L;newP}(BigInteger{L} newP): {L;newP} {
    this.p = newP;
  };

  public void setG{L;newG}(BigInteger{L} newG): {L;newG} {
    this.g = newG;
  };

  public void setY{L;newY}(BigInteger{L} newY): {L;newY} {
    this.y = newY;
  };
}
```

We can see programming elements such as the flagging of JIF for the class and the label: {L; newX}, which points that the higher and lower level of the caller must be the intersection of the label set (L) and the variable ($newP$ or $newG$ or $newY$).For this reason a "setter" function must be added, which was not obvious in the Java implementation.

4.4.3 Implementing a JIF class

This is an example in a so called "from scratch", where the Elgamal algorithm had to be re-implemented since the underlying data structures had changed significantly. For simplicity the initialization is demonstrated.

```
...
MPElGamal{L; this}() {
    int{L} randomBL = this.RandomBitLength; //saves from side
       effect

    //initialization
    Random{L} rnd = new Random();
    publicKey  = new MPKeyPublic[L]{L}();
    privateKey = new MPKeyPrivate[L]{L}();
    //assignments
    BigInteger{L} biPublic;
    try {
        BigInteger{L} pubP =  new BigInteger{L}(randomBL, rnd);
        publicKey.setP(pubP);
        BigInteger{L} pubG = new BigInteger{L}(RandomBitLengthLess,
            rnd);
        publicKey.setG(pubG);
        BigInteger{L} privX =  new BigInteger{L}(
           RandomBitLengthLess, rnd);
        privateKey.setX(privX);
```

```
        BigInteger{L} pubY =  pubG.modPow(privX, pubP);
        publicKey.setY(pubY);

    } catch (java.lang.NullPointerException npExp) {
        //do nothing
    } catch (java.lang.IllegalArgumentException ilargExp) {
        //do nothing
    } catch (java.lang.ArithmeticException arExp) {
        //do nothing
    }
}
...
```

The code is so different from the initial java implementation that it had to be completely rewritten.

5 Evaluation

Although the desired result was not one hundred per cent reached, the experience was definitely positive, beneficial and educational in every aspect. Due the course of this project there was exposure to diverse aspects of research and computer science concepts, such as programming, cryptography, mathematic, language based security and their implementations.

Additionally there was exposure to areas that are still under development which gave the experience of how concepts, ideas and implementations evolve in course of time.

Difficulties faced had to do with solving problems that in some times required knowledge of the domain in many aspects, such as programming, while in other cases there was need to invest in depth by reading the same references again and again, without an indication of a solution or knowing that the effort is pointed to the right direction.

The project was over ambitious in it's targets and did not reach it's goals in full extent. However there has been a tremendous amount of work done, which can be utilized in further attempts as a foundation for future extensions. It also gave a fair amount of exposure on how research is being conducted and how outcomes and conclusion of a person's or team's research propagate to the rest of the academic community and influence future publications.

5.1 Future Work

In case of further expansion to this project the following actions could be considered as logical next "steps":

In respect with mental poker, further implementation of the protocol can be made from the protocol's specification [5]. Game Verification can be coded. There is also room for Exceptional cases as a termination sequence when a player leaves the game [11], or when a connection is dropped. There is also room to create a "real world" situation with formal network code instead of interprocess communication.

In respect to Java with information flows aspect, there are unfortunately many actions that could be done. After some training in the language, the programming environment

[11] Action which is not covered by this protocol

and some handy JIF patterns additionally with familiarization with the tool-chain, a formal implementation can be produced from the mental poker code emphasizing on the aspects of Information Flow.

6 Appendices

6.1 Appendix A - Install JIF on ubuntu linux

A. Install Sun's java

Due to issues with Sun's openness of Java packages, it is possible that a sun-java, which is the most preferable for JIF, is not installed by default on various linux distributions, including Ubuntu. This is about to change but not at the moment. If you are reading this a year or more after 2009, then probably this section is obsolete.

Following the instructions from url: `https://help.ubuntu.com/community/Java`, (which is to be merged in the server guide, so another location to check is: `https://help.ubuntu.com/8.10/serverguide/C/index.html`) and
`https://help.ubuntu.com/community/JavaInstallation`, type in a new terminal:
sudo apt-get install sun-java6-bin
for the JRE, which asks for accepting sun's terms and then:
sudo apt-get install sun-java6-jdk
for the SDK.

This should be OK from a clear installation of the OS, where there is no previous version of Java installed. If there is another version already installed then there should be a switch, described in the first url (6.1), which involves the following:

Open a Terminal window
Run
sudo update-java-alternatives -l
to see the current configuration and possibilities.
Run sudo update-java-alternatives -s XXXX to set the XXX java version as
default. For Sun Java 6 this would be
sudo update-java-alternatives -s java-6-sun
Run **java -version**
to ensure that the correct version is being called.

JAVA_HOME variable

JAVA_HOME variable, which needs to exist for various scripts, at the moment of installation java was installed at: **/usr/lib/jvm/java-6-sun-1.6.0.10/**, so the /etc/bash.bashrc should be edited:
sudo gedit /etc/bash.bashrc
and the following line has to be added:
export JAVA_HOME=/usr/lib/jvm/java-6-sun-1.6.0.10/

In order to check for corectness use the following command:
echo $JAVA_HOME
in a new terminal window.

B. Install g++

Gnu g++ compiler is needed for compilation of JIF, therefore:
sudo apt-get install g++

C. Selection of jif version

Normally someone might want to run the latest version of JIF available, in order to use the latest features, there is only one reason for doing otherwise: Eclipse-IDE.
The plugin for Eclipse, jifclipse, which allows the use of the IDE in developing JIF applications, is unfortunately only compatible with jif version 3. In the case that eclipse is not needed, any version can be installed. Either way compilation is exactly the same.

B. JIF compilation

JIF compilation scripts use ant for building mechanism, so if it is not installed:
sudo apt-get install ant

For the following installation steps, instructions in the "README" file inside the jif folder, are being followed. Before that it is useful do define JIF environment variable, in the same way as JAVA_HOME. Many developers prefer to install JIF in their own account and not system-wide, therefore the export command should be appended in /.bashrc. **gedit /.bashrc** and append directory's location. Example: *export JIF=/home/of/user/jif*
For system wide, edit /etc/bash.bashrc, the same file for JAVA_HOME and update accordingly.

For the rest of the installation this directory will be named **$JIF**, and the steps in the "README" file are followed:
In a new Terminal window
Run
cd $JIF
ant configure
which builds the runtime environment, for
the compiler:
ant

Now JIF runtime with the possible addition of compilation should have been installed. It is advised to compile and run the Battleship game provided with the language in order to check the installation.

E. JIFCLIPSE

If the eclipse platform is not installed: **sudo apt-get install eclipse**
After starting the IDE, instructions from
`http://siis.cse.psu.edu/jifclipse/`
can be followed.
Generally the installation is similar to one of a typical eclipse-plugin, with some additional actions of course.

1. Select: Help → Software Updates → Find and Install
2. Choose "Search for new features to install"
3. Click "New remote site"
4. **Name**: Jifclipse,
   URL: `http://siis.cse.psu.edu/jifclipse/update`
5. Click "Finish"
6. Select check box for "Jifclipse" and click "Next"
7. "Jifclipse feature 2.0.0" and read/accept license agreement
8. Click "Next" and "Finish"
9. Restart Eclipse

The only differences with the official instructions before was Eclipse's prompt to install an unsigned feature, followed by Eclipse prompting to restart itself.
The final step of Eclipse installation has to do with patching the JIF installation: [12]

1. Download the JIF compiler and follow the instructions provided to compile the JIF runtime (at least).
   Currently you must download JIF 3.0.0; we are working on integration of Jifclipse with more recent releases of JIF.
2. Download our updated jif.jar and build file: Jif updates for Jifclipse.
3. Unzip **jifclipse-aux.zip** into the JIF directory (the jifclipse-build.xml file should be in the main JIF directory, /usr/local/jif-3.0.0, e.g. and the jif.jar will be placed into /usr/local/jif-3.0.0/lib). You should be prompted whether you want to replace lib/jif.jar and you should answer "yes". Note: you MUST compile the JIF runtime before performing this update.
4. (If you needed to edit the build.xml file for JIF to add the headers include directory, the same edit will be required for the Jifclipse build file.)
5. Rebuild the JIF lib and SIG jars by running ant:
   ant -f jifclipse-build.xml
6. Start up Eclipse and use Project → New → JIF → JIF Project to create a new JIF Project

[12] Note that from the quoted text (1) and (2) have already been done in steps (A) (B) and (D)

7. When creating a new JIF project, it should be configured to point to the base directory of the JIF compiler. See the Startup Tutorial for a demonstration.

In step 7 there are no instructions, since the how-to is in video only format.
In the eclipse: File → New → Project,
then select: "New Jif Project" and provide a name for the project, then click next. After this step there are two dialog boxes: **Base directory** for jif compiler and **Path** to javac post compiler, where the appropriate directories must be set, in this case: */usr/local/jif-3.0.0* changed to */home/of/user/jif* and */usr/lib/jvm/java-1.5.0-gcj-4.3-1.5.0.0/jre/../bin/javac* to */usr/lib/jvm/java-6-sun-1.6.0.10/bin/javac*

6.2 Appendix B - Directory structure of deliverable

List of files included in final report:

additional folder

jif-exercises: solutions for JIF exercises [17].

MentalPoker.bib: comprehensive list of all papers about mental poker, compiled in 2007.

december08_report: The preliminary report submitted on September 2008 which includes the code for the first steps of mental poker and a communication mechanism that avoids the use of threads .

implementation_code: Holds the code generated for this project:

java_code: Mental poker implementation in Java

jif_code: Mental poker implementation in JIF

6.3 Appendix C - Java source code

6.4 Appendix D - JIF source code

References

[1] http://faculty.washington.edu/moishe/javademos/security/elgamal.java.

[2] Barbara Liskov Andrew C. Myers. A decentralized model for information flow control. 1997.

[3] Aslan Askarov and Andrei Sabelfeld. Security-Typed Languages for Implementation of Cryptographic Protocols: A Case Study. In Sabrina De Capitani di Vimercati, Paul F. Syverson, and Dieter Gollmann, editors, *Computer Security - ESORICS 2005, Proceedings of the 10th European Symposium on Research in Computer Security*, volume 3679, pages 197–221, 2005.

[4] Aslan Askarov and Andrei Sabelfeld. Security-Typed Languages for Implementation of Cryptographic Protocols: A Case Study of Mutual Distrust. Technical Report 2005-13, Department of Computer Science and Engineering, Chalmers University of Technology and Göeborg University, 2005.

[5] Jordi Castellà-Roca, Josep Domingo-Ferrer, Andreu Riera, and Joan Borrell. Practical Mental Poker Without a TTP Based on Homomorphic Encryption. In Thomas Johansson and Subhamoy Maitra, editors, *Progress in Cryptology - INDOCRYPT 2003*, volume 2904, pages 280–294, 2003.

[6] David Clark. Project description. `http://www.dcs.kcl.ac.uk/staff/david/msc_projects_08.html`.

[7] UCLA Computer Science Department. Elgamal's proof of homomorphism. `http://www.cs.ucla.edu/~rafail/TEACHING/WINTER-2005/L8/L8.ps`.

[8] Caroline Fontaine and Fabien Galand. A survey of homomorphic encryption for nonspecialists. *EURASIP J. Inf. Secur.*, 2007:1–15, 2007.

[9] Shaft Goldwasser and Silvio Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. *Computer Science Department, University of California - Berkeley.*

[10] Peter Gutmann. Lessons learned in implementing and deploying crypto software. In *Proceedings of the 11th USENIX Security Symposium*, pages 315–325, Berkeley, CA, USA, 2002. USENIX Association.

[11] Leslie Lamport. Password authentication with insecure communication. *Commun. ACM*, 24(11):770–772, 1981.

[12] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 2001.

[13] Emilie Nodet. Data-flow controll using jif in a heath care system. 2008.

[14] Christian Schindelhauer. A toolbox for mental card games. Technical report, 1998.

[15] Bruce Schneier. *Applied Cryptography.* John Wiley & Sons, 1996.

[16] Adi Shamir, Ronald L. Rivest, and Leonard M. Adleman. Mental Poker. *The Mathematical Gardner*, pages 37–43, 1981.

[17] Chalmers University. Jif exercises. `http://www.cs.chalmers.se/Cs/Grundutb/Kurser/lbs/JifLab2006/JifExercises.htm`.

[18] Cornell University. Jif reference manual. `http://www.cs.cornell.edu/jif/doc/jif-3.3.0/manual.html`.

[19] Cornell University. Jif's home page. `http://www.cs.cornell.edu/jif/`.

[20] Penn State University. Jif eclipse plugin. `http://siis.cse.psu.edu/jifclipse/`.

[21] Yi Mu Weiliang Zhao, Vijay Varadhara jan. A secure mental poker protocol over the internet.