CHALMERS

Practical implementation of information flow in Paragon

THESIS FOR THE MASTERS DEGREE SECURE AND DEPENDABLE COMPUTER SYSTEMS

JAVED NAZIR

Division of Computer Engineering
Department of Computer Science & Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2012

Practical implementation of information flow in Paragon

Javed Nazir

Department of Computer Science & Engineering

Division of Computer Engineering
Chalmers University of Technology
SE-412 96 GÖTEBORG, Sweden
Phone: +46 (0)31-772 10 00

Author e-mail: nazir@student.chalmers.se

Printed by Chalmers Library
Göteborg Sweden, 2012

Practical implementation of information flow in Paragon

Javed Nazir
Division of Computer Engineering, Chalmers University of Technology

ABSTRACT

Paragon is a newly developed language by the security research group of Chalmers University of Technology, Sweden. The major task of my thesis work is practical implementation of a real world problem in Paragon. I have chosen mental poker game to implement in Paragon. Another part of my thesis work is to compare Paragon with Jif, with respect to how convenient implementation is in the two languages, and how well Paragon achieves the goal of improving on the limitations of Jif. After implementating the case study in Paragon, and comparing Paragon and Jif, I conclude that Paragon is able to provide more precise guarantees than Jif.

Keywords: Information flow, Paragon, Mental poker implementation, Jif implementation, Jif Vs Paragon

Acknowledgments

All praises to almighty Allah for the strengths and His blessing for providing me this opportunity and granting me the capability to proceed successfully. I am grateful to the following people for what they have done for me, for my career, and for this thesis.

- ▷ Niklas Broberg for his supervision and constant support. His inestimable help of constructive comments and suggestions throughout the thesis works have contributed to the success of this research.
- ▷ David Sands my examiner who has provided great research topic to work on.
- ▷ I would like to express my appreciation to all my colleagues and friends for their kindness, technical and moral support during my study.
- ▷ Last but not the least, I would like to thank my family members especially my parents and Sajid Hussain for always encouraging and believing in me.

Javed Nazir
Göteborg, November 2012

Contents

List of Figures

1 Introduction

Today, securing sensitive information from malicious use is becoming an increasingly important problem to overcome. This problem is emphasized as computer and internet technology becomes more and more prevalent. A user with limited access to a system can use vulnerabilities to attack the system, to gain or manipulate sensitive information. Often these vulnerabilities are caused by inadequate implementations of software in the system. Examples include faulty implementations of security protocols, improper use of cryptographic primitives, or flawed security models.

There are generally three types of attacks[4,5]:

- Confidentiality, which means that the attacker tries to obtain secret data.
- Integrity, which means that the attacker tries to change sensitive data.

- Availability, which means that the attacker tries to interrupt users who can legitimately access the system.

These three attacks are related to each other in the sense that results from one attack can also be used to help another attack. During program execution, information flow analysis tracks how information propagates to ensure that the information is securely handled in the program. In information flow analysis the confidentially and integrity categories are typically related.

It is important to apply some policies to data when information is moving from one place to another for the purpose of secrecy. Consider for example a cryptographic protocols, where a fresh shared key is used between two peers to prevent an attacker from stealing information by eavesdropping on network traffic.

In secure information flow illegal flow of data is not allowed. A policy specifies what flows are legal, and the security levels of the data. A legal path means information can flow among given security classes.

In a simple example, variables are categorized into two security levels. These two levels are public, and secret (in other words low, and high). The purpose of this categorization is to keep secret data away from public variables. In more complex scenarios we could work with a lattice of security levels, where it is ensured that sensitive information is only flows upwards in the lattice[6, 7].

1.1 Information flow security

Information flow security [8] means that the computations of programs involves an information flow analysis. An information leak can occur by an illegal assignment, e.g. `low := high`, or through termination or nontermination of a program, for example
`while high != 0 do skip`
It is also possible to leak information through the program control flow:

```
if(high == 0)
{
    low=1;
else
    low := 1;
}
```

Explicit and implicit flow of data is leaked through program variables as described above. There are also other, so called covert channels which can be divided into different categories for signaling information[8]:

- Termination or nontermination of a computation in a program, called termination channels.
- Time at which some actions occur, called timing channels.
- When probability distribution of data is changed, called probabilistic channels.
- By checking the power consumption, called power channels.
- By checking exhaustion of shared resources, called resource exhaustion.

1.1.1 Information flow Vs Access control

Information flow security is a useful complement to traditional security mechanisms like cryptography and access control because it can enforce different security policies.

Access-control mechanisms grant or deny access to a piece of data at particular points during the system's execution. For example, the read-write permissions provided by a file system prevent unauthorized processes from accessing the data at the point when they try to open the file. Such discretionary access controls are widely used [10,11] in practice.

A security-typed language can provide end-to-end protection which means that the data is protected throughout the duration of the computation. To understand the difference between information flow and access control consider this policy: "the information contained

in this e-mail may be obtained only by me and the recipient." Because it controls information rather than access, this policy is considerably stronger than the similar access-control policy: "only processes authorized by me or the recipient may open the file containing the e-mail." The latter policy does not prohibit the recipient process from forwarding the contents of the e-mail to some third party.

The below examples describe scenarios where access control and cryptography are insufficient to protect confidential data, but information flow control can be used[12]:

1. A web-based auction service is used to bid on merchandise. Anyone may bid on items with their price, but they are not allowed to see other customer's bid prices. Because the customers do not necessarily trust the auction service, the customer's machines share information sufficient to determine whether the auction service has been honest. When the bidding deadline finished, the auction service reveals information about the winning bid. Security policies that govern how data is handled in this auction scenario can potentially be complex. Access control and encryption are definitely beneficial techniques for enforcing these policies, but the customer auction server and software can be developed in a security-typed language to ensure that bids are not leaked.

2. It would be difficult to enforce that a sender of an email regulates how a recipient uses email via access control because it is an information flow policy. While cryptography would almost certainly be used to protect confidential email and for authenticating users, a security-typed language could be used to write the email software.

3. A user who uses accounting software wants an assurance that the accounting software doesn't send her credit information to the Internet whenever it queries a database on the web. The software company does not want the user to download the database because then a competitor might get proprietary information. However, the accounting software, is available to download from the company's web site.
   Security-typed languages provide the possibility to verify user's home computer af-

ter downloading software. This verification process gives assurance that it will not leak any confidential data, even though it communicates with the database.

4. Many programs written in C are vulnerable to buffer overrun and format string errors but, C standard libraries do not check the length of the strings. As a result of it, if a string obtained from an untrustworthy source is passed to one of these library routines and parts of memory may be overwritten with untrustworthy data. This vulnerability can be used to execute an arbitrary program such as a virus.
   Security-typed languages can prevent these vulnerabilities by specifying that library routines require high integrity arguments [13,14].

1.2 Noninterference

Goguen and Meseguer introduced non-interfering in 1982 [9] as the property that "one group of variables/users/processes by using a specific commands is noninterfering with another group of users if and only if these commands does not effect on another group".

Noninterference is a basic information flow policy enforced by security-typed languages. All implicit and explicit flows are prohibited from Secret to Public. Noninterference also holds for integrity. Tainted variables should not be able to influence the contents of Untainted variables. So, security analysis should also rule out implicit and explicit flows from Tainted to Untainted.

Data confidentiality demands that private information should never be revealed to unauthorized users. A program can maintain its data confidentiality property by preserving data to reveal in public outputs. If public data is influenced by confidential data then variation on secret data will produce the difference in output, which is observable.

Security-typed languages are designed to ensure noninterference, but noninterference is often not the desired policy in practice. Many useful security policies can leak confidential information. For example, passwords are Secret but the operating system authentication

mechanism reveals information about the passwords - namely whether a user has entered the correct password.

Practical security-typed languages include declassification mechanisms that allow controlled release of confidential data, relaxing the strict requirements of noninterference.

1.3 Security Policy

Security policies are sets of rules that describe the sensitivity level of data and how data should be secured. Therefore, security polices are very important to design and enforce for secure information flow. The purpose of a policy is to maintain data integrity and data confidentiality. If the correct policy is designed and enforced on sensitive data then misuse of sensitive data by an unauthorized intruder can be prevented. Different types of security policy models are available such as the Military security policy, the Commercial Security policy and the Bell and La Padula policy model. Each of them resists read/write of data from unauthorized users.

1.3.1 Bell and La Padula security model

The Bell and La Padula security model provides secure information flow by providing specifying paths of information flow. Two properties characterize the secure flow of information. Before understanding properties consider a security system with the following properties. The system covers a set of subjects S and a set of objects O. Each subject s in S and each object o in O has a fixed security class C(s) and C(o) where C is a Clearance. A clearance is an indication that a person is trusted to access information up to a certain level of sensitivity and that the person needs to know certain categories of sensitive information.

Simple Security Property

A subject s may have read flow to an object o only if C(o) <= C(s). The purpose of the simple security property is to restrict read flow of an object if the object wants to read

higher clearance level information.

***-Property**

A subject s who has read access to an object o may have write flow to an object p only if C(o) <= C(p). The * property restricts write flow of data towards those objects which are lower in clearance level.

1.4 Language based security

Conventional programming languages do not provide security for information flow. In programs there are different ways to leak information from a running program. Typically confidentiality and integrity can be achieved by language based security while availability is dependent at system level security. Language based security uses security type systems to check flow of program.

Explicit and Implicit flows leak information. Using language based security we can handle implicit and explicit flow of information. Here is an example for leakage of information flow. Let us consider variable secret with a highly secret and leak with a low secret value. Variable secret should not assign directly to variable leak.

1.4.1 Explicit information control

If value of secret is directly assigned to variable leak then this is an explicit flow and explicit flows are not allowed in secure information flow.

```
Leak=secret;
```

Conventional programming languages permit this assignment, because this is a legal assignment, but language based security will not allow this statement as secret data is assigned to public data.

1.4.2 Implicit information control

Indirect flow of information is called implicit flow. Conventional programming languages do not catch implicit flows. Implicit flow could be described by the following example.

```
if(secret % 2 == 0)
{
    Leak=0;
else
    Leak=1;
}
```

Implicit flows are often controlled by a program counter (pc) in security-typed languages. This policy tracks dependencies of the program context. In the above example, the pc in the branches of the if statement captures the dependency on `secret`. The assignment statement to `leak` is rejected by the compiler because the affected variable is less secure than the pc.

1.5 Security typed language

Security-typed languages provide precise ways of describing complex policies. A security-typed language can help software developers detect security flaws in their programs and security-typed languages can rule out programs that contain potential information leaks or integrity violations. Security-typed languages provide more confidence that programs written in them are secure.

1.6 Objective

JIF is an extension of Java that is used for security-typed programming. Jif supports information flow control and access control and enforces them at compile time and run time. Jif statically checks the information flows in programs. Jif enforces control over the flow of information throughout the life-cycle of a program. Confidentiality and integrity of

information can be handle with a static information flow control in a system. The Jif compiler translates Jif program to Java program. The Jif compiler produces secure executable programs by using an ordinary Java compiler. Jif implements information flow through a policy language called the DLM, which will be described in Chapter 2.

Paragon is a newly developed language by the security research group of Chalmers University of Technology, Sweden. Paragon came into existence as the result of the research work on Practical, Flexible Programming with Information flow Control by Niklas Broberg[1]. Like Jif, Paragon aims to overcome problems with information flow, thereby protecting confidentiality and integrity of sensitive data. Policies and locks are the main pillars of Paragon. I will describe policies and locks in Chapter 4.

In this Master's Thesis, I have done a case study of mental poker by implementing it in the Paragon language. The major task of my thesis work is practical implementation of a real world problem in Paragon. I have chosen mental poker game to implement in Paragon. Aslan Askarov has done mental poker Implementation in Jif by the title " Cryptographic Protocols: A Case Study of Mutual Distrust" for his master thesis in 2005. He has implemented the Java source for mental poker in Jif to show secure information flow. Another part of my thesis work is to compare Paragon with Jif, with respect to how convenient implementation is in the two languages, and how well Paragon achieves the goal of improving on the limitations of Jif.

1.7 Outline

In the next chapter i am going to discuss Jif (Java Information Flow), and how the decentralized label model is used in Jif. In chapter 3 I will give an overview of the Mental poker protocol used by Aslan Askarov along with an analysis of the protocol. I will also compare the Java and Jif implementations of mental poker done by Aslan Askarov. Chapter four discusses the basis of Paragon, handling the syntax used in Paragon. It also contains concepts of different objects used in Paragon. Fifth chapter discusses the main task of my thesis, the

methods that I have used to bring out the strength of Paragon. It also discusses the problems found in the Jif implementation and how I have solved them to show how Paragon can provide stronger and more expressive rules of information flow. The last Chapter contains Conclusions and Future work.

2

Java Information Flow

Jif is an extension of Java implementing the decentralized label model (DLM)[16]. Jif adds information flow annotations in the form of DLM labels.

2.1 Decentralized label model

The DLM hold its own vital properties and each section of data has flow policies added by principals. Labels provide detailed information about the flow policies of all principals, policies never to be violated by systems. DLM controls information flow in a system with mutual distrust. In a DLM there is no any central authority who define and decide security policies. Security policies are defined and controlled by everyone, who own them, which are then enforced by the system in accordance with all of the security polices that have already been defined. A principal can perform data declassification by modifying its flow

policies in the label, but arbitrary classification is not possible as the policies of other principals are not weakened and remain the same. This guarantees the successful working of the model even when the principals do not trust each other. For the purpose of maintenance and security, run time checks are done to avoid information leaks and so declassification is done in a safer way as described by DLM[1,2].

The primary weakness of the DLM (ultimately of Jif) is that it comes without a semantic characterisation of security like eg. the old model by Bell and LaPadula. Since the DLM allows declassification, it is clear that it cannot guarantee non-interference and non-interference is too restrictive[1].

2.1.1 Principal

Users and other authorities like roles and groups are Principals in the Jif. Principals own, and update information. Information is also released to principal in DLM. In Unix both users and groups are represented by principals.

In the Jif principals can also act for other principals with their full authorization. For example, if a principal P acts for another principal P', then principal P possesses all rights and privileges of P'. This characteristics is transitive and reflexive. For example, if a group possesses an authorization to act for all principals then a group member can act for group principals. A role is a restrictive form of user authority which gives permission for an authorized user to act only on role principal.

2.1.2 Label

A label provides privacy requirements with policy sets[16]. When a program is executed, the derived and computed value has a related label and the program execution also has its own label (program counter). The privacy policies have two parts, an owner and set of readers and which in the following form, {owner: Readers}. The owner of the policy is a principal, whose data is utilized for constructing the value that the label annotates and

permits itself to read the data. The readers are the set of principals who are permitted by the owner only to read the data. Each policy which is in the label must obeyed as data flows within system. Only owner can release information to other Principals by his/her permission. A principal which exists in the all policy labels, either as an owner or a reader, can read the data. This is because the intersection of all policies are enforced and adding more policies to a label will restrict the propagation of labeled data. The bottom security level in Jif corresponding to public data has label { }, which is an empty list of policies.

{bob:;}

A policy with no readers and principal means that only the owner of the policy is to be able to read the data and principal p does not care how the data propagates. An example of such a policy without reader list and principal is given above.

{bob:alice,charlie}

Above example has bob as owner of policy and two readers. Bob has given read rights to alice and charlie.

{bob:alice; alice:charlie}

This is an example of an associative label, which contains two policies. A principal who is available in both label policies can read the data. In this label Alice is the only principal which is present among the readers of both policies, so only Alice can read the data.

2.2 Jif syntax

This section discusses how variables, methods and exceptions are declared in Jif and how labels are used with them.

2.2.1 Variable declaration in Jif

A local variable declaration in Jif may contain a label annotation, which is called the declared label of the variable. The label of a local variable is the join of the program counter label for the declaration, and its declared label. If a local variable is declared without label, then the local variable's label is inferred[17]. Here are two variables example with initial-

ization expression and label.

For both variables Alice is the owner of the label and can perform operations with these

```
int {Alice:} x;
int {Alice:} y=1;
```

variables. A local variable declaration may contain an initializing expression. The rules for label checking require that the normal value label of the initializing expression is no more restrictive than the label of the local variable.

2.2.2 Method declaration

In Jif a method declaration may be annotated with two labels, the begin label and the end label. The begin label reflects the lower bound of the methods in terms of side effects on them. If no begin label is specified in method a declaration, then it is assumed that the method has no side effects. Methods with side effects must explicitly indicate their begin label. Labels can also be assigned to the arguments of the method. The label of an argument denotes the lower bound on the security level of the argument.

The end label of a method carries information about whether the method terminates normally or raises an exception. End-labels are necessary if the termination path of the method may give out some information to the caller. Individual exceptions and return values may be labeled separately as well. An example of a method declaration is

```
public boolean{Alice:Bob} Check{Alice:}(String{} name, int{} link):{Alice:}
```

In this example, the function Check takes two arguments, both of which are of the bottom security level. The return value has label {Alice:Bob}. Both the begin and end labels are {Alice:} .

2.2.3 Method constraints

Jif allows three different constraints in method declarations:

- authority(p1, ... , pn) specifies the list of principals that this method is authorized to act for.
- caller(p1, ... , pn) specifies the list of principals whose authority the caller of the method is required to possess in order to run this method.
- actsFor(p1, p2) means this constraint prevents the method from being called unless the specified p1 acts for p2 relationship holds.

2.2.4 Exceptions

Java exceptions and Jif exceptions are semantically different. In Jif all runtime exceptions have to be handled otherwise it would be possible to leak information via them. Here is an example of how runtime exceptions can be maliciously used
If variable `secretvalue` is zero ArithemeticException is thrown in the method division.

```
public class IntLeak {
private int {Alice:} secretvalue;
public int{Alice:} division(int{} b) { return b/secretvalue;
}
}
```

That is why it is necessary to handle runtime exceptions.

2.2.5 Parameterized classes

Jif allows classes and interfaces to be parameterized over labels and principals. This introduces another level of polymorphism. For example, instead of writing two separate Player classes for Alice and Bob one can write a single class Player[P] parameterized over principal variable P. Later in the instantiation the principal parameter is substituted with the actual principal. An example of a parameterized class definition is:

```
public class Player[principal P, label L] {
        public String{L} name;
        private final KeyPair{P:} keyPair;
        public void finishCardDraw{L}():{L} throws MPException where caller(P)
        {
            statement(s);
        }
                                          }
```

This class is parameterized over a principal P who owns sensitive information stored in an instance of this class and a label L is used to denote low data. The variable `name`, the name of the player, in this example is low, thus it is labeled as {L}. Sensitive data like `keyPair` contains a pair of encryption keys both public and private. Therefore, `keyPair` has a {P:}.

2.2.6 Array labels

Arrays in Jif also have two labels: one for the elements of the array, the other for the length of the array. Only one label for arrays is not enough, since arrays are mutable data containers. A variable `lowArray` of type int[]{} could be assigned to a variable `highArray` with the labeled type int[]{L} for some more restrictive label L. Then it is safe to assign a variable secret labeled as {L} to an element of array `highArray`. Here are examples to describe legal assignments and information leakage assignments.

```
int[]{} lowArray;
int[]{L} highArray;
int{L} secretValue;
highArray = lowArray; //allowed
highArray[0] = secretValue; //Not allowed leakage now low\_array[0] == secretValue
```

Here are two examples how to declare array using different labels with them.

```
private int{Alice:}[]{} hand;
private boolean{}[]{} available;
```

The first array is Alice's hand of cards. The bottom label {} stands for the size of the array and the values of the actual cards are secret to others. Therefore, elements of the

array are labeled as {Alice:}. The second array available is declared with low elements so, labeled as {}.

3

Mental Poker Protocol (MP)

A simple poker game is played using cards among players,but mental poker is played without the use of cards and verbal communication. The exchange between players is accomplished by using messages. In this scenario any player may try to cheat other players. A protocol for mental poker must detect or avoid cheating if a player tries to cheat on another player. It must guarantee fairness of the game. Protocols for mental poker are divided into two groups based on security levels.

- Mental poker protocol in the presence of trusted third party (TTP).
- Mental poker protocol without presence of a TTP.

A mental poker protocol in the presence of TTP is efficient and provides fairness of the game: that no one cheated in the game and game is finished without cheating. On the other hand protocols without presence of TTP are designed for environments with mutual distrust.

3.1 Protocol objective

Crépeau formulated the requirement and objectives for the mental poker protocol [3], which are as follow.

1. **Uniqueness of cards:** Each card from the deck must appear uniquely either in the deck or in the hand of a player. In case the same card appears more than once, if is result of cheating. So each card must appear once and only once to avoid cheating.

2. **Uniform distribution of cards:** Traditionally in poker one player shuffles cards while the other players can see it, and the player who shuffles the cards distribute cards to each player. In MP cards are uniformly distributed between players so that the card set of a player is not dependent only by opponent player actions. Each hand of a player must be dependent on a decision made by every player.

3. **Absence of trusted third party (TTP):** It is not realistic to fully depend on a trusted third party. This is obvious that any human can be bribed, and no machinery is completely safe because no entirely tamper-proof device has yet been produced.

4. **Cheating detection with very high probability:** The probability that a player may cheat without being detected must be very small, and decrease very fast, and mental poker protocol must catch those players who try to cheat.

5. **Minimal Effect of Coalitions:** If two or more players are involved then some player may start a secret communication to share all their knowledge about the game, the protocol or any other secret data. A mental poker protocol should reduce the effects of a coalition, so that any player in the game cannot take advantage of other player's hands or cards in the deck.

6. **Complete confidentiality of cards:** It is very important in a poker game that information of any card from the deck either partial or total is obtained without the permission of every opponent. It is also very important for mental poker protocol to check when a player draws a card then other players must be denied to get information of that card without that player's permission.

7. **Complete confidentiality of strategy:** In the game the player who loses a game must not reveal their cards at the end of a game. So, an ideal protocol allows the players to reveal neither their cards nor any other information leading to some information about them.

3.2 Protocol Analysis

Aslan Askarov's[2] case study of mental poker protocols suggests that Castellá-Roca et al[4] is the best protocol for mental poker, and achieves the first six goals mentioned above. This protocol is a TTP-free protocol and it reaches the first 6 required goals for mental poker. One more reason to choose this protocol by Aslan Askrov was that elimination of TTP does not increase the computation.

In this protocol, players carry out deck shuffling themselves and cooperate with each other in shuffling, so that no player coalition can force a particular outcome. Each player generates a random permutation of the card deck and keeps it secret. After permutation, cards are encrypted with the player's key so that no other player can obtain information about cards until the game is finished. When the game is over then all players reveals their keys and their permutations for validation.

This protocol introduces a tool called the distributed notarization chain **(DNC)**. The importance of DNC can be understood by the fact that a new link of DNC is build after each operation, until game is not finished. DNC contains two fields: a chaining value X which is a hash of all previously built links, and a datafield D. The datafield D contains three subfields which are link subject S, timestamp T and another attribute B which dependent on subject S.

3.3 Protocol process

Aslan has analysis different protocols for mental poker and on the basis of best results he chosen protocol described by Castellá-Roca et al[4]. Aslan has selected four best protocols, and I am going to describe protocol 1, 2 and 4 from [6] for mental poker. Description of protocol 3 is not done here because I am not referring to it in later discussion. These protocols 1, 2 and 4 are not driven from the mental poker protocol objectives that I discussed above. These 3 protocols are.

- **Initialization**
- **Card draw**
- **Game validation**

I will describe now purpose of each protocol.

Protocol 1 (Initialization)
Each player has an asymmetric key pair (Pi,Si) whose public key component is certified by a recognised certificate authority. The initialization protocol is very important because in this protocol random permutation of cards is generated by each player and each player keep it secret so that no other player can access it. After permutation each player generates a symmetric key K which will be used in the signing of cards. Each player chooses a prime number which is used to create DNC link which contains the value of prime number chosen by the player. Now a card permutation matrix is built, and the next link to DNC is build after committing permutation matrix using a bit commitment protocol. Next part of this protocol is very important because vector representation of cards in deck is generated and encrypted it with Player's K. After generating a random permutation of encrypted cards a new link of the DNC is built which contains all previous processes performed by the player. In the whole process of the initialization protocol, information flow is controlled by encryption process and DNC links.
Protocol 2 (Card draw)

In this protocol each player draws a card. After drawing a card players must be unaware of other player's cards. First each player checks the validity of the link by computing her equivalent card permutation and choose a card to draw and build new link of the DNC which contains id of next player in the computation. Now all other players also check the validity of the link sent by the previous player after computing their equivalent card permutation and build a new link of the DNC. Each player encrypt card with their corresponding symmetric key K and build a new link of the DNC. After each link been computed then the player checks the validity of the link and obtain the drawn card contained in the link, by decrypting it with her private key. After this process card draw protocol is finish. In the whole process of card draw each player checks the validity of the link sent by previous player, so this validity check will avoid information leakage for those players who intended to get information about card.

Protocol 4 (Game validation)

Each player check that the permutation which she committed with the bit commitment protocol in the first step of initialization is the same as revealed, and used by each player. If the permutation is correct then it decrypts cards which are published by each player in initialization protocol and checks that the card deck is correct. After this, players decrypt the card permutation performed in protocol 2 to check that permutations were performed correctly. Each player checks that during the game those cards which are discarded by other players have not been used. Each private key is released at end of the game. Other actions performed in protocol 1 and 2 are validated to check cheating in the game.

3.3.1 Declassification

In this protocol, four parts are very important where declassification has to be considered. Aslan Askarov has pointed out these four parts in his thesis, which I am referring here. These points are as follow:

1. **Public data declassification**
   This declassification occurs before the game starts. A player's key pair contains both sensitive (Private key) and non-sensitive (Public key) data. Here the sensitive data should not be available to the other players. Another point occurs when players draw

cards. Drawn cards are encrypted and contain sensitive data. If declassification is performed properly with Sensitive data then it should not be leaked because it will not flow to low level.

2. **Declassification related to building links in DNC**
   As i have discussed before, when a new operation is performed in the protocol then a new link is added to the DNC. Thus, declassification must be handled when a link is computed. The signature of DNC link involves a private key so the result becomes high level. Thus, the cryptographic properties of the signature must be declassified.

3. **Finishing Card Draw**
   When the Card Drawing process is finished then a flag which carries information about the finished status must be declassified.

4. **Declassification for verification's sensitive information**
   Protocol 4 (Verification) states that when the game completes then players are required to exchange their private keys and secret permutations, which they have created in the first protocol (Initialization), in order to verify the fairness of each other in game and to detect cheating. The importance of declassification can be easily understood at this stage. If data exchange between players is not declassified then any player can obtain the secret permutation and private key of a player to do cheating in a game.

3.4 Java implementation of mental poker

The class diagram in figure 3.1 describes the Java implementation of mental poker. All arithmetic operations in the Java implementation uses the java.math.BigInteger class.
The distributed notarization chain (DNC), its links and data attributes that are used in DNC are implemented in the classes DNCChain, DNCLink and DataFieldAttribute. DataFieldAttribute is a interface which is implemented by many classes.

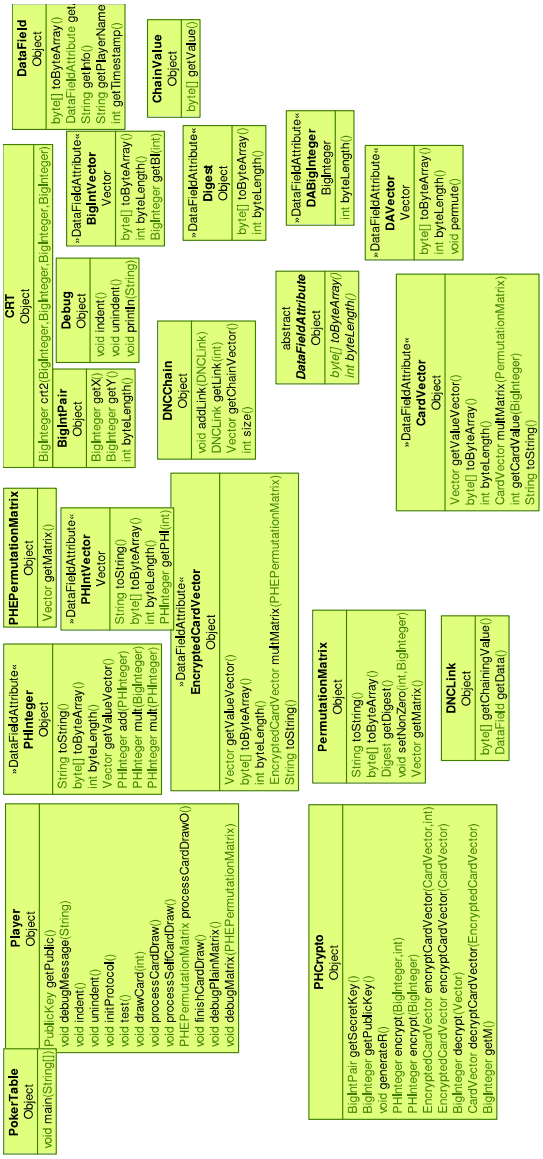

Figure 3.1: Java implementation of mental poker.

PHInteger is an extented class from Integer for storing encrypted values and arithmetic operations on them. PH stands for privacy homomorphism. The PHCrypto class generates the cryptosystem keys, and encrypting/decrypting of values. The CardVector class is used to store the card vector in plain form and EncryptedCardVector is the vector representation

in encrypted form.

A major and important class in the mental poker java implementation is the Player class, in which players are initialized and perform actions, and on basis of these actions new links are added to the DNC.

The class MPTable coordinates the process of the game. In MPTable there is a function play() which is called when the application starts.The play() function controls the order of player instructions for protocol 2 (Card draw) and also for protocol 4 (Validation), in which players perform verifications of links and the permutation matrix.

3.5 Jif implementation

To fulfill the security requirements of mental poker, sensitive information of players carries the labels by their name in Jif. For example {Alice:;} and {Bob:;}. The data passed between players is downgraded to the bottom level {}.
Java class signatures are as Jif signatures which add labels and principals. The class Player is parameterized over player principal P and the label of the output channel L. Any variable, method declaration or exception with label {P:;L} corresponds to the high level and L to a low one.
In figure 3.2 some examples are given for method signature and variable declaration in Jif. First write the Java signature and then the corresponding Jif signature.

3.5.1 Java signatures Vs Jif Signatures

As I discussed before, PHCrypto handles encrypting/decrypting of values and keys, and all cryptographic operations must be performed at a high security level. For this reason the PHCrypto class is instantiated with argument {P:;L}. An array in Jif is declared with two labels. The `p` byte array corresponds to the player's secret permutation so each element of the array must be secret. That is why the first label of this array keep elements of array

Java signature	Jif signature
PHCrypto ph = null;	PHCrypto[{P:;L}]{P:;L} ph = null;
Byte [] p;	byte{P:;L}[]{P:;L} p ;
KeyPair keyPair	KeyPair{P:} keyPair
Boolean[] available	boolean{L}[]{L} available
String name	String{L} name
DNCChain chain	DNCChain[L]{L} chain

Figure 3.2: Java signatures Vs Jif signatures for secret and public variables.

secret and the second label keeps the array `length` secret. In the last example the public component of the KeyPair is not secret, but even so the whole KeyPair has to be labeled as high.
Boolean variable `available`, player `name` and `chain` in Jif are public and any player can read them, as they are only labeled with L. The absence of the principal shows that every user has the right to read these variables. This is how public variables are modeled in the Jif implementation of mental poker.

3.6 Declassification in the Jif implementation

In the Jif implementation of mental poker there are 14 declassification points identified by Aslan Askarov. These 14 points are divided into four different groups based on their nature. I will discuss the results of these declassification groups individually.

- **Public data declassification**
  This declassification occurs before game starts. The public and private keys for sig-

natures are generated using the **KeypairGenerator** Java class. The key pair contains both sensitive and non-sensitive data. The sensitive data should not be available to other players at any time so the program first obtain a separate copy of both keys and then declassifies it. Now private key will not be accessible before and during game. In the **drawcard** method the public parameter of PHCrypto class is extracted from an instance and again new copy of this parameter is obtained and declassified separately. In both steps of key pair generation and draw card, sensitive data is declassified because it contains information sensitive and non-sensitive information. Sensitive data will not be leaked because it has been declassified and it will be hidden from other player and it will not flow to low level.

- **Declassification related to building links in DNC**
  As i have discussed before, when a new operation is performed in the protocol then a new link is added to the DNC. Declassification is needed when the method, which is used to built links, is called, protocol is initialized, and when cards have been drawn. The signature of a DNC link involves a private key so the result becomes high level. Thus, the cryptographic properties of the signature are declassified and the obtained declassified value is used to build the next link.

- **Finishing Card Draw**
  When the Card Drawing process is finished then it updates a success flag that signals that the process finished. The success flag is then declassified.

- **Declassification for verification's sensitive information**
  In Protocol 4 (Verification), when the game completes then the protocol requires players to exchange their private keys and secret permutations which they have created in the first protocol (Initialization) in order to verify the fairness of each other in game and to detect cheating. At the end of the game permutation matrix is generating to verify matrix that is generated by players at start of the game.
  In his Jif implementation of mental poker, Aslan Askarov [3] has used a "seal", a boolean flag, to handle and verify game end because the declassification mechanism of Jif is not powerful enough to support temporal properties.

4

Paragon Background

I will now describe the different concepts, annotations and terminology used in Paragon.

4.1 Actor

Actors are values of a primitive data type, actor. The actor data type neither allows literal values, nor provides expressions that create new actors. There is no default value (like e.g 0 for int) for actor variables, instead all variables of the actor data type are assigned with a value implicitly which is unique for each actor variable. Variables of actor data type are typically declared as final. The purpose of declaring actors final is to ensure consistency in the analysis. All actors are unique by their value because actors declared with actor data type cannot be initialized with any type of value. Variables are only declared by their name.

```
private static final actor bob;
```

Here in the above example actor annotation is used to declare an actor in paragon and the name of the actor in example is bob. Actors are used in policies and mentioned actors can obtain information.

4.2 Locks

Locks in Paragon are boolean guards having open or closed state. Locks are not values: they cannot be stored in variables nor can they be passed as argument to methods. If locks are being used in a Paragon policy then data is only flowed towards actors if and only if lock is open. For this reason the lock state (open or closed) controls the flow of information, either it can flow to actor or not.

```
private lock Encrypted;
```

This is how locks are declared in Paragon. Locks can be declared with any access modifier (public, protected or private). `Encrypted` is a lock having private access modifier, and thus can only be accessed within the class where it is declared.

```
open Encrypted
{
statement (s);
}
```

Locks can be opened by a Paragon `open` statement. Within the body of lock statement(s) can be written as shown above. There is another way to open lock which is describe below.

```
open Encrypted;
```

In first example the `Encrypted` lock is open and inside its body statements will be executed. after the lock's body THE state of the lock will be changed to the state which it was before opening the lock. In second example lock state is open but the state of the lock `Encrypted` will remain open until it is not closed explicitly. The lock can be closed like this `close Encrypted`.

4.3 Policy

A paragon policy describes the sensitivity level of some data. A policy is built from actors and locks.
Each policy is a set of clauses separated by semi-colons, where each clause is written with the head first, possibly followed by a colon and a list of conditions (locks) to be open for the actor in the head to observe data annotated with this policy. Typically policies are marked as final to ensure that the policy remains consistent throughout the program.

```
public final policy publicdata = {'x:};
```

In the above example a final policy is declared with the name of `publicdata`. All data annotated with this policy may flow to everyone, as this includes a polymorphic actor. A polymorphic actor is marked with a preceding ' followed by an identified, e.g 'x in above example.

```
public final policy bobdata={bob:};
```

In the above example a final policy is declared with the name `bobdata`. Data annotated with policy `bobdata` can only flow to actor bob. Any actor other than bob tries to obtain data with this policy will be rejected.

```
public final policy privatedata = {bob: Encrypted};
```

In the above policy, `privatedata` is the name of a policy declaring that bob can access data annotated with this policy if `Encrypted` lock is open. If `Encrypted` lock state is closed then data flow annotated with `privatedata` policy will be denied to actor bob.

4.4 Modifiers in Paragon

In Paragon fields, variables, methods and exception have a policy which describe that how information contained in them may be used. There are two policy modifiers for policy and three modifiers for locks in Paragon.

Lock	Policy
+lock says that method *will* open the specified lock(s).	?policy specifies the policy on data in an information container e.g. a field, variable or exception. When the modifier is used on a method then it is called a return policy, as it is the policy on the value returned by the method.
-lock says that the method *may* close the specified lock(s), for some execution.	!policy denotes the write effect and it is used to annotate methods and exceptions. When a field or variable with some policy is modified by an expression, that policy is part of the write effect of that expression. They are also used to signal the write effects of throwing an expression and also to catch implicit flows.
~locks says that the specified lock(s) must be open whenever the method is called.	

4.4.1 Read policy with variables

```
Private ?privatedata int bitlength = -1;
```

Variable `bitlength` is annotated with a policy with read effect on it.

4.4.2 Read policy with arrays

```
Private ?privatedata int[]<privatedata> val;
```

In arrays two things are important to handle. One is the length of an array and second is the elements of the array. Here the array is annotated with two policies, first with `?privatedata` to describe the policy on the length of the array, and second with `<privatedata>` to describe the policy on the elements of the array.

4.5 Policy-polymorphic methods

Method `sum` takes two parameters `a` and `b`. The parameter `a` has a declared read policy `privatedata` while parameter `b` is not having any policy so by default policy of `b` is inferred from the policy of the argument passed for parameter `b` . Return policy of method

```
?(privatedata*policyof(b)) public static int sum(?privatedata int a, int b)
{
return a+b;
}
```

sum is annotated by joining of parameter's policies.
If a parameter to a method has no policy declared, then a polymorphic policy is assigned to that parameter. To access the polymorphic policy of a parameter we use the policyof(b) method which takes a parameter of method as its parameter and gives the policy of the argument that method is called on. Here we use policyof(b) because no policy is assigned to b and for return policies of this method, we need a policy for parameter b.

```
public !privatedata static int setX(?privatedata int a)
{
this.x = a;
}
```

The write effect of the method in the example above is declared to be privatedata which means that the method setX has modified value of variable x annotated with privatedata policy in the class. The purpose of using write effects is to control implicit flow.

4.6 Parameterized class

A class in paragon can be parameterized with a policy. We can also parameterize a class with actor and lock. I haven't implemented class parametrization with locks and actors that is why I will not write detail about it here. A class with a policy parameter can be declared as

Any object create this class will require a policy as parameter, i.e
`Player<p> newobject= new Player<p>();`
This introduces another level of polymorphism and is useful for building reusable data

```
public class Player<policy privatedata>
{

}
```

structures. For example, instead of writing two separate Player classes with different policy for Alice and Bob we can write only one class and can pass different policy for both players. Later in the instantiation the policy parameter is substituted with an actual policy.

5
Implementation Methodology

In this chapter I will discuss the work I have done as part of solving the task, which is to write a case study in Paragon. I will describe my first implementation, a mechanical translation, of Aslan's Jif implementation. I will then describe my Paragon implementation of mental poker, and problems encountered while doing these implementations. I will also describe the problems in the Jif implementation of mental poker by Aslan Askarov. The main purpose of doing a Paragon implementation is to show how Paragon is more expressive and can give stronger information flow guarantees.

Here are three implementation problems which are not handled in the Jif implementaion. The first problem is trusted declassification. Trusted declassification is a concept in which different kind of data is declassified by different declassifiers and boolean guards, which are handled by methods and locks in Paragon. Player can achieve secret data after it has been passed through a specific declassifier to make sure it's trusted declassification. I

will describe later in this chapter that I have achieved trusted declassification.

The second problem which I have noticed is the java library signatures written by hand, not by compiler, and used in the Jif implementation. This could lead to a serious attack which I will describe later in this chapter.

The third and last problem is the lack of ability to specify temporal properties. Temporal properties can be used to reflect the state of a game. Jif does not support temporal properties and Aslan Askarov has implemented the end game analysis by using a so called "Seal", a programming pattern.

The first and last of the above problems of significant importance not be handled in a Jif implementation. In this chapter I will discuss how I overcome these problems and what I have concluded after implementing these in my Paragon implementation.

5.1 Mechanical Translation

By "Mechanical translation", I mean a process of translating code written with Jif annotations into code with Paragon annotations, simply replacing Jif annotations with corresponding Paragon annotations. The purpose of the mechanical translation is to check whether Paragon is no more restrictive than Jif. If after mechanical translation Paragon is providing the same results as the Jif implementation then we can say that Paragon is also a strong and secure language that can be used to implement programs with information flow concerns. Another purpose of the mechanical translation is to check whether the Paragon compiler is working properly or not. The compiler is supposed to show errors and warnings if illegal syntax is used or if a leak is found.

My mechanical translation also helped find a number of bugs in the compiler. These bugs have been reported to the compiler bug tracker [15] and subsequently removed by the Paragon team. After the mechanical translation I conclude that this solution is giving the guarantees as Jif implementation provides. But, during the mechanical translation I found

some aspects which were not handled in the Jif implementation, and I decided to perform a more detailed implementation with more expressive policies and locks that can overcome these problems.

5.2 Java library

I have translated a number of Java libraries into Paragon code. The purpose of this is to let the compiler generate correct Paragon signature files for these libraries. Most of the classes are parameterized over a policy and this policy is used throughout the class. This introduces another level of polymorphism and is useful for building reusable data structures. It allows to reuse the class with different policies.

Aslan Askarov has used some Jif signatures written by hand, not generated by the Jif compiler. We found a bug in one of the these hand written signatures, which leaves the code vulnerable to leaks. I changed java library signatures into paragon signature to have the compiler ensure that information flows correctly.

One problem I faced when creating paragon signature files for java libraries is related to the dependencies of java files. For example, to complete java.util.Random I must create signatures for all classes which are used by it. The java.util.SecureRandom must be done first, and to complete SecureRandom I must do SecureRandomSpi, and so on. I created files for dependencies up to depth 3, and beyond that trusted my hand written versions.

5.3 Paragon Implementation of mental poker

As noted I initially did the mechanical translation to check whether Paragon could provide sufficient guarantees. I reached the conclusion that the mechanical translation provides same result as the Jif implementation, but found some aspects that were not possible to implement in Jif. So, I decided to do a detailed implementation, using more fine-grained locks and policies for different types of methods and variables depending on their security

level. I did an analysis of such variables and the data returned by the methods, which are secret and require tight security and policies, to give them precise and descriptive policies.

I will describe the types of locks that i used in my paragon implementation, as well as the policies that include these locks to make precise policies for variables and data returned by methods depending on their intended use.

5.3.1 Temporal properties and trusted declassification in Paragon implementation

Locks	Policies
Private final lock GameStart	Public final policy publicdata={'x:}
Private final lock GameRunning	Public final policy privatedata={'x:Encrypted,GameRunning,GameStart}
Private final lock GameFinished	Public final policy policyforkey={'x:GameFinished}
Private final lock Encrypted	

Figure 5.1: Temporal properties and trusted declassification in Paragon.

Figure 5.1 shows the locks used in the Paragon implementation. The first three locks control the temporal properties of the game. The game process contains different kinds of data and then, this data is declassified through the use of different locks. Locks make sure that data is accessed only when the locks are open, and the lock are opened at their required places. These locks are declared as private so they can be used within declared class and can not be used or opened in another class or package.

The `Encrypted` lock is used for trusted declassification in the Paragon. This lock is declared in the PHCrypto class and is used within the encryption method. The Encrypted

lock is declared as private so that it cannot be accessed outside of class PHCrypto. This assures that all players have access to the data after it has been encrypted by the encryption method. In my Paragon implementation the encryption method thus acts as a trusted declassifier.

Figure 5.1 further shows the policies that are being used in my paragon implementation of mental poker. These polices control the flow of data to actors or players who fulfill the conditions of the policies. I will describe these policies in detail to understand their purpose.

Lets look at the policy `privatedata` from figure 5.1 to understand what it does and its purpose. This policy is used with the variables and data returned by methods which are supposed to be kept secret. During the game other players can read this data if and only if both `Encnrypted`, and `GameRunning` locks are open.If either of these locks is closed then public access to information which is annotated with this policy is rejected.

The second policy is `policyforkey`. A player's asymmetric Key contains two parts: one of them is public and the second is private. The private key must be revealed when the game is finished, but not before, so for this key pair I created a policy which contains the lock `GameFinished`. This key is only accessible when the `GameFinished` lock is open, and I have ensured that this lock is only going to be opened after all processes of the game are finished. This policy assures that there is no attack on the private key and no one can access it before the game is finished. If the private key is not leaked, consequently cards are safe from cheating as they are encrypted. They need a key to decrypt them and the key is available only when game is over.
The last policy is `publicdata` which contains only the `'x` polymorphic actor. This means anyone can access information which is annotated with this policy.

Jif Code	Paragon Code
private final DNCChain[L]{L} chain	private ?publicdata DNCChain<publicdata> chain;
private byte{P:;L}[]{P:;L} p	private ?privatedata byte[]<privatedata> p;
private PHEPermutationMatrix[{P:;L}]{P:;L} matrix_o = null	private ?privatedata PHEPermutationMatrix<privatedata> matrix_o;
private PHCrypto[{P:;L}]{P:;L} ph = null	private ?privatedata PHCrypto<privatedata> ph;
private final KeyPair{P:} keyPair	private final ?publicdata KeyPair<publicdata,policyforkey> keyPair;
private boolean{L}[]{L} available ;	private ?publicdata boolean[]<publicdata> available;
private String{L} name;	private ?publicdata String name;

Figure 5.2: Jif implementation vs Paragon implementation.

5.3.2 Jif implementation vs Paragon Implementation

This is a comparison table of Jif and Paragon policies. Most of the code in figure 5.2 shows same policies for both languages. For example with the array variable `p` in both languages there are equivalent corresponding policies. The main difference is the `KeyPair`. In Jif code only P is used but, in Paragon I used different policies for the public and private parts of the key.

It may appear that Paragon is more verbose, since there are more words in Paragon declaration. This length complexity is not inherent in Paragon syntax, rather, it is my implementation which has used long and descriptive policy name. We can use any legal variable name instead of these long name. In the Jif code Aslan has used L and P to describe Label

and Principal respectively. This approach is little difficult to understand, what is P and L here? On other hand in the implementation in Paragon I used descriptive names of policies clearly, like `privatedata` and `policyforkey`, which informs the purpose of these policies and where thet are to be used.

Jif Code	Paragon Code
private byte{P:;L}[]{P:;L} generatePermutation{P:;L}():{P:;L}	**private** !privatedata ?privatedata byte[]<privatedata> generatePermutation()
public void processSelfCardDraw{L}():{L} throws IllegalArgumentException	~(Player.GameRunning) **public** !publicdata void processSelfCardDraw() throws !publicdata IllegalArgumentException
public void finishCardDraw{L}():{L}	~(Player.GameRunning) **public** !privatedata void finishCardDraw()

Figure 5.3: Jif and Paragon Methods and Exception comparison table.

Figure 5.3 describes how methods and exceptions are used in Jif and Paragon. Each policy in Jif and Paragon correspond to each other. Each label in Jif has a corresponding policy in Paragon. The Label `{P:;L}` used after `generatePermutation` is write policy of the method in Jif and `privatedata` is write policy of method `generatePermutation` in Paragon. The policy right after the method is used for exceptions in Jif, and a write effect policy is used with exceptions in Paragon as seen in the `processSelfCardDraw` method. In Paragon the write effect is used to assure that when we catch exceptions we still do not leak. That is why exceptions are annotated with write effect policies.
Methods that return data also have return policies. If a method writes something in its body then a write effect policy is used with this method to ensure that we can catch implicit flow.

An "expects" lock modifier can also be used to check the status of a lock. Before calling a method, the lock status must be the same as it is described in the method declaration. In the above example of `processSelfCardDraw` method `GameRunning` lock is

mentioned which says that this lock must be open when this method is called.

5.4 Implementation Conclusion

Paragon is a newly developed language and I have aimed to show a strict and sound implementation of information flow. Therefore, I have implemented mental poker in Paragon and compared it with Aslan Askarov's Jif implementation. As discussed before the Jif implementation had some limitations which I have removed in my Paragon implementation. By implementing Java library signatures in Paragon I have overcome signature problems of the Jif implementation. By using the `Encrypted` lock I modeled trusted declassification, and by using locks `GameFinished` and `GameRunning` I have modeled temporal properties.

In short i can now say that Paragon can do better and more expressive work compared to Jif.

6

Conclusion and future work

This chapter includes conclusion of this thesis and future work related to this.

6.1 Conclusion

Information flow is an important topic that deserves full attention from industries and academics. Previously much work related to information flow has been done in the context of Jif. Paragon is a newly developed language, which needs a strong case study that shows its strength for implementing information flow control. I have evaluated the strength of Paragon by implementing a real world problem. For this purpose, I choose mental poker so that I can verify that Paragon is robust enough, and can provide strong and more expressive solutions than other languages, such as Jif. The reason for selecting mental poker is

that the secure information flow of mental poker has already been checked in Jif by Aslan Askarov.
As the first step to check whether Paragon can provide the same results as Jif, I did a mechanical translation of Jif mental poker into Paragon, which is to change Jif annotations into Paragon annotations. After this whole process I compiled the Paragon implementation of mental poker which gave the same result of information flow control.
However, during this process I found some points which have not been covered and I decided to do a more detailed implementation. These problems are listed below:

1. Generation of java libraries into Jif by hand, not by Jif compiler
2. Lack of implementation of temporal properties in Jif
3. Lack of modeling trusted declassification.

In my detailed implementation of mental poker I first did standard java libraries in Paragon and compiled them that so that I can have checked Java libraries available in Paragon, which ensured that there is no leakage of information from java libraries. Aslan Askarov made java library files in Jif manually, not created by the Jif compiler. Jif did not provide temporal properties of game, which include game running and end. I overcame this problem of Jif implementation by making different locks and used them to mark particular states of the game and with particular types of data. Lastly I ensured that different types of data is declassified with different types of locks which are open in those classes where they are supposed to be opened and used. The `encryption` method played the role of trusted declassifier and `Encrypted` lock is used to declassify for secret data, and I have assured that `Encrypted` lock is only open in `encryption` method.
After my mechanical translation of the Jif implementation into Paragon the results were strong enough to prove the strength of Paragon. However the problems found in the Jif implementation gave a chance to show better results in Paragon. Hence, I solved these problems in Paragon implementation of poker, which is a proof that Paragon can implement and provide a better information flow solution.

6.2 Future work

I did mental poker implementation in the Paragon to check information flow constraints, which I successfully achieved. Besides this implementation Paragon needs more case studies and practical works to show its strength for other kinds of constraints. JPmail is a secure email client which uses the Jif to get information flow control guarantees. JPmail has been fully implemented in Jif and it is a good case study after mental poker to implement in Paragon to show more strength of Paragon. Besides JPmailm big industries and companies need a collaboration work with academic research group to keep their data more confidential, and secure by implementing information flow control on them via Paragon.

Bibliography

[1] N. Broberg and Institutionen för data-och informationsteknik (Göteborg), *Practical, Flexible Programming with Information Flow Control*, Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology and Göteborg University, 2011.

[2] 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.

[3] C. Crépeau, "A secure poker protocol that minimizes the effect of players coalitions.," in *Advances in Cryptology: Crypto'85*, December. 1986, vol. 218 of LNCS, pp. 73–86.

[4] Dorothy E. Denning and Peter J. Denning, "Certification of programs for secure information flow. commun.," July. 1977, vol. ACM, 20, pp. 504–513.

[5] Rajeev Joshi and K. Rustan M. Leino, "A semantic approach to secure information flow," *Science of Computer Programming*, vol. 37, no. 1-3, pp. 113–138, 2000.

[6] D. E. Denning, "A lattice model of information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, May 1976.

[7] D. Bell and L. La Padula, "Secure computer systems: Unified exposition and multics interpretation," Tech. Rep. MTR-2997, MITRE Corp., Bedford, MA, July 1975.

[8] Andrei Sabelfeld and Andrew C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.

[9] Joseph A. Goguen and Jos'e Meseguer, "Security policies and security models.," in *IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[10] B. Lampson, "Protection," in *Proc. Fifth Annual Princeton Conference on Information Sciences and Systems*. Princeton University, 1971, pp. 437–443.

[11] G. S. Graham and Peter J. Denning, "Protection: Principles and practice.," in *In Proc. of the AFIPS Spring Joint Conference.*, 1972, pp. 417–429.

[12] Stephan Arthur Zdancewic, *Programming languages for information security*, Ph.D. thesis, Ithaca, NY, USA, 2002, AAI3063751.

[13] Jeffrey S. Foster Umesh Shankar, Kunal Talwar and David Wagner, "Detecting format string vulnerabilities with type qualifiers.," in *In Proceedings of the 10th USENIX Security Symposium,*, 2001.

[14] David A. Wagner, *Static analysis and computer security :–new techniques for software assurance*, Ph.D. thesis, University of California, Berkeley, Fall, 2000.

[15] "http://code.google.com/p/paragon-java/issues/list," in *Google document for bug reporting*, May 2012.

[16] Andrew Clifford Myers, *Mostly-static decentralized information flow control*, Ph.D. thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1999.

[17] K. Vikram Lantian Zheng Stephen Chong, Andrew C. Myers, "Jif reference manual," in *http://www.cs.cornell.edu/jif/doc/jif-3.3.0/manual.html*, February 2009.