Secure Implementation of Cryptographic Protocols: A Case Study of Mutual Distrust

Aslan Askarov
aaskarov@cs.chalmers.se

April 18, 2005

Abstract

Security protocols are critical for protecting modern communication infrastructures and are therefore subject to thorough analysis. However practical implementations of these protocols lack the same level of attention and thus may be more exposed to attacks.

This thesis discusses security assurance provided by security-typed languages when implementing cryptographic protocols. Our results are based on a case study using Jif, a Java-based security-typed language, for implementing a non-trivial cryptographic protocol that allows playing online poker without a trusted third party.

The case study deploys the largest code written in a security-typed language to date and identifies insights ranging from security guarantees to useful patterns of secure programming.

Contents

Acknowledgements

First and foremost I would like to thank my supervisor Andrei Sabelfeld for introducing me to the field of language-based security. Without his energy, patience and great sense of humour this work would not be possible.

Thanks to all members of PROSEC group at Chalmers for the productive working environment and to Niklas Broberg for the comments on the draft of this thesis. I would also like to thank Daniel Hedin with whom I have been sharing office for the last two months.

Last but not the least, I want to thank my parents for supporting me from Home all the time I have been in Sweden.

Chapter 1

Introduction

1.1 Security protocols

Security protocols are central to every modern security-critical computer system. Examples of security-critical applications where these protocols are used include but not limited to are

- e-commerce
- home-banking
- military applications
- online gambling
- e-voting

These applications rely on the protocols for user authentication and authorization, online transactions, ticket reservations and money transfers. Often they involve multiple parties at a time - tens of users may play at an online poker table, hundreds bid in an online auction and thousands participate in electronic elections.

1.2 Attacks

Naturally, these systems attract various malicious activity. Attackers are trying to find vulnerabilities in the security protocols and thus breach into these systems. Vulnerabilities that attackers try to exploit may be of these kinds:

1. Weaknesses in the design of a protocol
2. Weaknesses in the implementation of a protocol.

The first type of vulnerabilities is very dangerous, but also relatively rare. The description of security protocols are usually open to the public and are thus subject to thorough analysis by cryptographic researchers. As a consequence, the industry tends to use the protocols that are well established over the time and believed to be strong.

However, the high rate of successful attacks on the computer systems and their analysis indicates that the major threat comes not from the design of security protocols but rather from their inadequate implementations. The work by Gutmann [14] covers some common ways in which cryptographic and security software can be misused by programmers.

1.3 Secure implementation

Our contribution to this field is that we suggest using security-typed languages for implementation of security protocols rather than just relying on techniques and methodologies for conventional programming languages. Security typed languages have been known for quite a long time in the programming languages community. However, being quite restrictive they have been inappropriate for real-world programming and thus been little known by general public. The recent progress in the development of practical security-typed languages makes their usage more real. These languages have appealing feature of controlling how information about sensitive data (e.g. private keys) propagates in a given program.

In particular, this work presents a case study of securing implementation of cryptographic protocols. The goal of this work is to investigate what benefits one can gain by using security-typed languages for security protocols; what possible problems may occur and how they can be resolved.

This project also targets the problem of feasibility of realistic programming in security-typed languages. For past three decades security-typed languages have remained an attractive technique for providing confidentiality in programs. Theory of secure information flow concentrating on controlling how sensitive data propagates in a program have been developed and improved during this time. The research in the area have mostly consolidated on designing type systems that prevent secret information from affecting publicly observable behavior of a system. A comprehensive survey by Sabelfeld and Myers [26] goes in details through the work that has been done. These theoretical achievements eventually lead to the development of real-life languages [23, 30] but "yet despite this large body of literature and considerable, ongoing attention from the research community, information-flow based enforcement mechanisms have not been widely (or even narrowly!) used" [32].

The presented implementation is the largest program in a security typed language written so far as it is known to the author.

1.4 Case study, protocol building blocks

For this study we picked the mental poker problem. Despite the fancy name and direct application in e-gambling, this problem is interesting because its security goals are similar to those of other protocols. These goals are mutual distrust environment, absence of a trusted third party, confidentiality, auditability, fairness, high probability of cheating detection and computational efficiency. These properties are observed in other applications as well. For example, in online voting, it is important that every vote remains confidential but at the same time, the result becomes known to the public after the election is over.

Recall the attack kinds mentioned in Section 1.3. For our problem they can be rephrased as follows:

1. Weaknesses in the design of the protocol or *how can one be sure that the other parties are not malicious?* For this question we assume that the protocol we are using is strong with respect to our goals.

2. Weaknesses in the implementation of the security protocols or *how can one be sure that the program implementing the protocol running on one's computer does not contain vulnerabilities*.

As it has been mentioned before, this work addresses the second problem and tries to identify what benefits one can gain from using security-typed languages in addition to conventional common-sense techniques.

In order to measure the effectiveness of a security-typed language, we have implemented the protocol in Jif and compared it with a baseline implementation written in a conventional programming language (Java). This experiment has shown that Jif's static checking captures dangerous informations leaks that were present in the Java program. These leaks are related to some side-effects such as exceptions that depend on sensitive program variables.

In the next phase, the Jif implementation has been made more realistic by adding distribution. This has shown that the language is capable of supporting the features real-world programs rely on.

1.5 Contributions

This study identifies different categories of controlled information release that may influence future work on defining finer policies for information downgrading. In particular, we stress that there is a need for more expressive declassification mechanisims capable of covering different reasons for intentional information release.

This report discusses insecurities that have been found in the baseline implementation and how secure programming may prevent them. Additionally we have discovered some practical problems and vulnerabilities in Jif that need further improvements.

We also present patterns for secure programming that have been developed during this work and that may be useful when writing programs in security typed programming languages.

1.6 Overview

The rest of the report is organized as follows: Chapter 2 provides the reader with some background information about the protocol for mental poker and the Jif language. Chapter 3 discusses in detail the three different implementation that have been done. The discussion and evaluation of security assurance is provided in Chapter 4. This section also contains a number of programming patterns that emerged during this work and may be useful for Jif programmers. Chapter 5 discusses the related work. Chapter 6 concludes this report.

Chapter 2

Background

This chapter contains some background information, the description of the protocol which is used in the implementation, as well as an introduction to Jif and security-typed languages.

2.1 Landscape of protocols for Mental Poker

2.1.1 Mental Poker

The poker game

The following brief introduction to poker is an extract from [2].

Poker is a card game, one of the most popular forms of gambling, in which players with fully or partially concealed cards make wagers into a central pot, after which the pot is awarded to the remaining player or players with the best combination of cards. The history of the game is a matter of some debate — the first direct reference to poker is dated back to 1843. The game is played in hundreds of variations, but the following overview of game play applies to most of them.

Depending on the game rules, one or more players may be required to place an initial amount of money into the pot before the cards are dealt. Like in most card games, the dealer shuffles the deck of cards and appropriate number of cards are dealt face-down to the players. After the initial deal, the first of what may be several betting rounds begins. Between rounds, the players' hands *develop* in some way, often by being dealt additional cards or replacing cards previously dealt. After the first betting round is complete because every player called an equal amount, there may be more rounds in which more cards are dealt in various ways, followed by further rounds of betting (into the same central pot). At any time during the first or subsequent betting rounds, if one player makes a bet and all other players fold, the deal ends immediately, the single remaining player is awarded the pot, no cards are shown, no more rounds are dealt, and the next deal begins. At the end of the last betting round, if more than one player remains, there is a *showdown* in which the players reveal their previously hidden cards and evaluate their hands. The player with the best hand according to the poker variant being played wins the pot. Some deals may not reach the showdown phase if all players drop out except one.

History of the Mental Poker

The history of the mental poker goes back to 1981, when Shamir et al. first asked the question "how to play a poker over the telephone?" [29]. They proposed the first solution to the problem for the scenario with two players. Later ([21, 7]) the protocol has been shown to be weak allowing the cards to be marked. The problem has continued to attract researchers and many solutions have been proposed [13, 6, 8, 9, 28, 4, 18, 17]. The short presentation of some of these will be given later in this chapter.

The protocols for mental poker can be generally divided into two large groups:

- Protocols that assume presence of a trusted third party (TTP): the protocols in this group are usually efficient and fair provided that the TTP is fair as well.

- Protocols that don't assume presence of a TTP: this family of protocols are designed for environments with mutual distrust.

2.1.2 Protocol objectives

In 1985 Crépeau formulated the objectives for the mental poker protocol [8], which we repeat here

1. Uniqueness of cards: every card must appear once and only once, either in the deck or in the hand of one player. The only case when a card may appear more than once must be the result of some detectable cheating.

2. Uniform distribution of cards: the hand of each player must depend on decisions made by every players, so that none of them has any control on his hand or on his opponents'. Every possible hand must have an equal probability and be accessible to all players.

3. Absence of trusted third party (TTP): no trusted party may be assumed, since any human can be bribed, and no machinery is entirely safe because no tamper proof device can be achieved.

   Generally, TTP is not desirable in any security protocol as it raises the question whether a TTP can gain something from cheating or not. For example, in e-gambling online casinos act as TTP and at the same time may participate in the game, making other players basically helpless.

4. Cheating detection with very high probability: any attempt to cheat must be detected. The probability that a player may cheat without being detected must decrease very fast (exponentially) with respect to some security parameter that the players must decide before the game. Also the amount of work to accomplish the protocol should increase reasonably (polynomially) with respect to this parameter.

5. Complete confidentiality of cards: no partial or total information about any card from the deck may be obtained without the approval of every opponent. Also, no information may be obtained from a player's hand without his approval.

6. Minimal effect of coalitions: when more than two players are involved, some players could establish secret communication and exchange all their knowledge about the game, the protocol or any secret data involved in the protocol. Nonetheless they should not be able to take advantage of this. This information should be equivalent to the cards they separately have in their hands. In other words, as long as one player is not cheating, nobody can learn more about his hand, or about the cards in the deck, than what they can deduce from the cards in their coalition.

7. Complete confidentiality of strategy: it is strategically very important in the game that the loosing players may keep their cards secret at the end of a hand. This is also absolutely necessary for preserving the element of bluffing in the game: when a player raises aggressively while holding a hand that is likely to be inferior, hoping that all other players will fold and award the player the pot without a showdown. Therefore, an ideal protocol forces the players to reveal neither their hands nor any information leading to some knowledge about them.

Even though cryptographers have succeeded in designing protocols that theoretically achieve all these properties, the proposed solutions([9, 18, 17]) tend to be demanding to the computation time and are generally inacceptable in practice [11].

2.1.3 Overview of protocols

In this section we will briefly sketch a few protocols that have been proposed for the mental poker game and will provide their short evaluation.

Shamir-Rivest-Adleman protocol

Shamir-Rivest-Adleman protocol [29] is the first, proposed for this problem. It relies on the commutative cryptosystem

$$E_k(x) = x^k \text{mod p}$$

where k is a secret key, and p is large public prime. The commutativity of the protocol can be expressed as

$$E_A(E_B(x)) = E_B(E_A(x))$$

The protocol consists of the following steps:

1. Alice and Bob choose 52 values $\{x_1, \ldots, x_2\}$ corresponding to cards in the deck.

2. Bob encrypts each x_i with private key k_B:

$$c_i = E_{k_B}(x_i), i = \{1, \ldots, 52\}$$

   Bob permutes the $\{c_i\}$ and sends them to Alice.

3. Alice chooses five c_i and sends them to Bob. These will be the cards of Bob.

   Alice chooses five more c_i, uses her private key k_A to encrypt them and obtains

$$d_i = E_{k_A}(c_i) = E_{k_B}(E_{k_B}(x_i)), i = \{1, \ldots, 5\}$$

   Alice permutes $\{d_i\}$ and sends them to Bob

4. Bob decrypts $\{d_i\}$ with k_B^{-1} to get

$$e_i = D_{k_B}(E_{k_A}(E_{k_B}(x_i))) = D_{k_B}(E_{k_B}(E_{k_A}(x_i))) = E_{k_A}(x_i), i = \{1, \ldots, 5\}$$

   The five decrypted e_i's are sent to Alice.

5. Alice decrypts the cryptograms e_i with k_A^{-1} and obtains her five cards

$$x_i = D_{k_A}(e_i) = D_{k_A}(E_{k_A}(x_i))$$

6. Players reveal their keys in the end of the game for verification.

Evaluation This protocol satisfies the first four objectives. It is restricted to two players. Besides that, it has been shown [22, 7] to *leak partial information* about cards if

- $p-1$ has a small prime divisor, or
- random-looking bits used for padding are selected in a special way.

The latter weakness has been fixed [7], but still there is no formal proof that the resulting protocol is secure.

Protocol by Zhao *et al*

The protocol by Zhao *et. al* [34] is based on the protocol by Shamir *et. al* but uses a different cryptosystem. The protocol has an advantage of being computationally efficient and fast. However, the usage of this cryptosystem makes the protocol insecure [5]. An attacker may gain the player's card without knowing the encryption key.

Goldwasser and Micali protocol

Protocol by Goldwasser and Micali [13] is based on probabilistic encryption and quadratic residuosity problem and has an advantage of providing a formal proof of correctness.

In this protocol, if p and q are two large prime numbers, $N = p \cdot q$ is public, and a card's binary representation is [010010], then Alice publishes six numbers $(q_1, q_2, q_3, q_4, q_5, q_6)$ such that q_2 and q_5 can be represented as

$$q_i \equiv x_i^2 \text{mod N}$$

and there is no such representation for (q_1, q_3, q_4, q_6). The difficulty for Bob is that he can't distinguish this fact without knowledge of the factorization of N. This is *known as the quadratic residuosity problem*. In the initialization step of this protocol, for every card i in their shuffled decks players publish values of N_i and encrypted representations of cards. Goldwasser and Micali prove that at this step an adversary can not decode card values. When dealing a card, parties exchange messages so that only one of them obtains the card's value. For verification players release their factorization of every N_i.

Evaluation The Goldwasser and Micali protocol is limited to two players only. It is also inefficient since a full RSA-style block is required to encrypt a single bit of information.

Toolbox for Mental Games by Schindelhauer

In 1986, Crépeau proposed a protocol that was first to achieve all desired properties [9]. It is based on the quadratic residuosity problem and *zero-knowledge proofs*: convincing the verifier the knowledge of a secret without revealing it. The protocol was shown to be practically infeasible. Thus, the implementation of it in 1993 took 8 hours to shuffle a deck of cards for 3 players [11]. The idea of Crépeau was extended in 1998 by Schindelhauer [28]). This protocol is also based on the quadratic residuosity problem and zero- knowledge proofs. It has the advantage of providing a protocol wide range of operations on cards and deck: e.g. possibility to control that players obey rules even when they make hidden moves without disclosing their secrets. This allows to use the protocol not only for poker, but also any imaginable card game. The cryptographic backbone of this protocol is the operation of *masking a card*: the card retains the same value, but cannot be read without everyone‘s help (that is, the encrypted value of the card changes).

Evaluation The protocols satisfies all the objectives listed, and provides a rich set of operations over the deck of the cards, but the performance is still unacceptable [15].

Protocol by Barnet and Smart

This protocol [4] is based on zero-knowledge proofs and introduces the set of Verifiable *l-out-of-l* Threshold Masking functions, consisting of four protocols:

1. Key generation protocol
2. Verifiable masking protocol
3. Verifiable remasking protocol
4. Verifiable decryption protocol

The protocol has the advantage of being independent of the number of players and number of different cards

Evaluation Barnett-Smart's mental poker protocol provides the same set of operation as in Schindelhauer's "Toolbox...". It also proposes improved data structure. Actual performance is yet unknown.

2.1.4 Castellà-Roca *et al.* TTP-free protocol based on homomorphic encryption

For this study we chose the protocol by Castellà-Roca et al, which achieves the first six goals, is zero-knowledge proof free and has modest computational requirements. The protocol description and exposition (except Figure 2.1) is borrowed from [6].

In this protocol deck shuffling is carried out by the players themselves. All players cooperate in shuffling, so that no player coalition can force a particular outcome, i.e. determine the card that will be obtained after shuffling. Every player generates a random permutation of the card deck and keeps it secret; the player then commits to this permutation using a bit commitment protocol. The shuffled deck is formed by the composition of all player permutations.

Shuffling a card corresponds to mathematical operations over the card's representation and permutation. Reversing cards in the physical world translates into encrypting cards. 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.

When a hand of a game is over, players reveal their encryption keys and their permutations for validation.

Evaluation This protocol does not require any TTP and reaches the first 6 desired goals for mental poker. In addition, eliminating the TTP does not result in dramatical increase of computation, thus making the protocol more practical. Thus, we chose this protocol for our implementation.

Requiring disclosure of players' strategies after the game is a limitation of this protocol, but on the other hand raises an interesting security issue: how can one enforce dangerous revelation not to happen earlier in the game?

Privacy homomorphism

The protocol relies on an additive and multiplicative homomorphic cryptosystem to shuffle a deck of cards and maintain the privacy of the cards. Such homomorphism can be defined as encryption function $E_k : T \to T'$ which allow performing addition and multiplication on encrypted data without knowledge of the decryption function D_k. For this implementation we use the cryptosystem suggested in [10].

Protocol description

Every player PL_i generates a random *permutation of the card deck* π_i and keeps it secret. The protocol then introduces the mapping from scalar representation of the cards to a vector representation:

- *Let t be the number of cards in the deck. Let z be a prime number chosen by a player. A card can be represented as a vector*

$$v = (a_1, \ldots, a_t) \tag{2.1}$$

  where there exists a unique $i \in \{1, \ldots, t\}$ *for which* a_i *mod* $z \neq 0$*, whereas* $\forall j \neq i$ *it holds that* a_j *mod* $z = 0$*. The value of the card is i; assuming a prescribed ordering of the cards, i is interpreted as a rank identifying a particular card.*

- *A permutation* π *over a deck of t cards is a bijective mapping that can be represented as a square matrix* Π *with t rows called* card permutation matrix*, where rows and columns are vectors of the form described by expression 2.1.*

$$\Pi = \begin{pmatrix} \pi_{11} & \pi_{12} & \cdots & \pi_{1t} \\ \vdots & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ \pi_{t1} & \pi_{t2} & \cdots & \pi_{tt} \end{pmatrix}$$

  The i-th row of matrix Π *is card* $\pi(i)$*, i.e. the card resulting from applying permutation* π *to the card having rank i. Thus, all elements in the i-th row of* Π *are 0 mod z except* π_{ij}*, where j=*$\pi(i)$*.*

With the above representation for cards and permutations, the result $w = \pi(v)$ *of permuting a card v using a permutation* π *can be computed in vector representation as* $w = v \cdot \Pi$ mod z*, where* $\cdot$ *denotes vector product. For this computation to work properly, the same value z must be used to represent* v *and* Π*.*

The protocol introduces a tool called *distributed notarization chain (DNC)*. Each operation performed during the game is notarized as a *link* of DNC. The DNC link is formed by two fields: a *data field* D_k and a *chaining value* X_k. The data field D_k itself consists of three subfields: *timestamp* T_k, link *subject or concept* C_k, which describes the information contained in the link, and additional *attribute* V_k, which depends on the subject C_k.

Chaining is guaranteed by chaining values X_k included in each link. First, the chaining value X_{k-1} of the previous link is concatenated with the data field D_k of the current link; then the hash value of the concatenation is computed and signed with the private key of the author of the k-th link, i.e.

$$X_k = S_{author}\{D_k | X_{k-1}\}$$

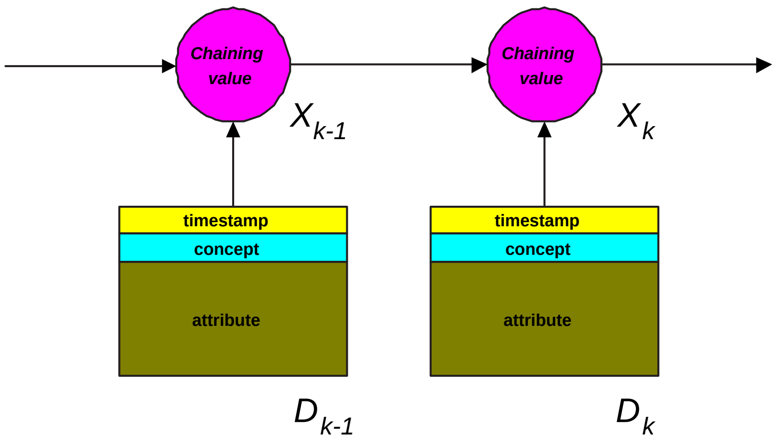

Figure 2.1: Distributed notarization chain

Now we present description of protocols 1,2 and 4 from [6]. Description of protocol 3 as well as crypto analysis of the protocol is not described here since we are not referring to them in later discussion.

Protocol 1 (Initialization)

1. *Each player PL_i is assumed to have an asymmetric key pair (P_i, S_i) whose public key has been certified by a recognized certification authority. Assume the card deck consists of t cards.*
2. *PL_i does:*
   (a) *Generate a permutation π_i of the card deck and keep it secret.*
   (b) *Generate a symmetric secret key K_i corresponding to a homomorphic cryptosystem allowing algebraic operations (additions and multiplications) to be carried out directly on encrypted data. Security requirements on this homomorphism are limited to resistance against ciphertext-only attacks.*
   (c) *Choose a prime value z_i which falls within the range of the clear-text cards space of the homomorphic cryptosystem used.*
   (d) *Build a link of the DNC which contains the value z_i used by PL_i*
   (e) *Build the card permutation matrix Π_i corresponding to π_i, using z_i.*
   (f) *Commit to this permutation Π_i using a bit commitment protocol. Denote the resulting commitment by Cp_i.*
   (g) *Build the next link of the DNC following the previous link. The new link contains the commitment Cp_i*
   (h) *Choose s values $\{\delta_1, \dots, \delta_s\}$ such that $\delta_j \bmod z_i = 0, \forall j \in \{1, \dots, s\}$ and $s > t$*
   (i) *Choose s values $\{\epsilon_1, \dots, \epsilon_s\}$ such that $\epsilon_j \bmod z_i \neq 0, \forall j \in \{1, \dots, s\}$ and $s > t$*
   (j) *Homomorphic-ally encrypt the previous values under the symmetric key K_i to get $d_j = E_{K_i}(\delta_j)$ and $e_j = E_{K_i}(\epsilon_j), \forall j \in \{1, \dots, s\}$*
   (k) *Build the next link of the DNC which contains the set $\mathbf{D} = \{d_1, \dots, d_n\}$*
   (l) *Build the next link of the DNC which contains the set $\mathbf{E} = \{e_1, \dots, e_n\}$*

(m) *Generate the vector representation for the t cards in the deck $\{w_1, \ldots, w_t\}$ and encrypt them under K_i using the aforementioned homomorphic cryptosystem to obtain $w'_j = E_{K_i}(w_j)$*

(n) *Randomly permute the encrypted cards $\{w'_1, \ldots, w'_t\}$*

(o) *Build the next link of the DNC which contains the card deck encrypted and permuted by PL_i*

Protocol 2 (Card draw)

1. *PL_i does:*

   (a) *Pick an integer value v_0 such that it falls within the range of cards in the deck, i.e. $1 \leq v_0 \leq t$, and which has not previously been requested. This operation is simple because it is public. All participants know the initial values chosen in previous steps.*

   (b) *Build the next link of the DNC which contains the vector representation w_0 of card value v_0 chosen by PL_i.*

2. *PL_1 does:*

   (a) *Check the validity of the link sent by PL_i, compute her equivalent card permutation Π'_1 for the modulo z_i published by PL_i in the DNC and permute w_0 to obtain $w_1 = w_0 \cdot \Pi'_1$*

   (b) *Build the next link of the DNC, which contains w_1 and the name of the next player PL_2 in the computation.*

3. *For j=2 to i-1, player PL_j does:*

   (a) *Check the validity of the link sent by PL_{j-1}, compute her equivalent card permutation matrix Π'_j for the modulo z_i published by PL_i and permute w_{j-1} to obtain $w_j = w_{j-1} \cdot \Pi'_j$*

   (b) *Build the next link of the DNC, which contains w_j and the name of the next player PL_{j+1} in the computation.*

4. *Player PL_i does:*

   (a) *Check the validity of the link sent by PL_{i-1} and permute w_{i-1} with her permutation matrix to obtain $w_i = w_{i-1} \cdot \Pi'_i$*

   (b) *Modify the m-th row of Π_i where $m \in \{1, \ldots, t\}$ is the value of card w_{i-1}. All values in the m-th row are changed to values that are nonzero modulo z_i*

   (c) *Pick the encrypted card w'_i corresponding to clear card w_i. Note that the encrypted deck has been published in the last step of Protocol 1*

   (d) *Build the next link of the DNC, which contains w'_i and the name of the next player PL_{i+1} in the computation.*

5. *For j=i+1 to n, player PL_j does:*

   (a) *Check the validity of the link sent by PL_{j-1} and compute her equivalent card permutation matrix Π'_j for the modulo z_i published by PL_i. Use Protocol 3 below to encrypt Π'_j as Π^c_j under the key K_i corresponding to PL_i*

   (b) *Permute the encrypted card w'_{j-1} using her encrypted matrix Π^c_j to obtain $w_j = w_{j-1} \cdot \Pi^c_j$*

   (c) *Build the next link of the DNC, which contains w'_j. If $j < n$, the link also indicates the name of the next player PL_{j+1} in the computation.*

6. *When PL_i sees the link computed by PL_n, she does:*

   (a) *Check the validity of the link computed by PL_n*

   (b) *Decrypt the card w'_n contained in the link computed by PL_n under her private key K_i to obtain the drawn card $w_n = D_{K_i}(w'_n)$. This finishes the card draw protocol.*

Protocol 4 (Game validation) Each player does the following:

1. *Check that the permutation revealed and used by each other player PL_i is the same permutation π_i to which she committed when publishing the commitment Cp_i in Protocol 1 (initialization). This check implies verifying the bit commitment for player PL_i.*

2. *Decrypt cards $\{w'_1, \ldots, w'_t\}$ published by each other player PL_i in the last step of Protocol 1 and check that the card deck is correct.*

3. *to decrypt the result of permuting encrypted cards at Step (5b) of Protocol 2. Check that permutations were correctly performed.*

4. *Check that cards discarded by other players have not been used during the game.*

5. *If necessary, use DNC to prove any detected misbehaviour by any other player to a third-party (casino, court, etc.).*

2.2 Security typed languages & Jif

This section describes the secure information flow problem and introduction to Jif programming language.

2.2.1 Problem of secure information flow

An information flow from object x to object y occurs whenever the value of y gets affected by the value of x. Flows can be explicit or implicit. Examples of *explicit* flows are assignment statement, I/O statements or value returns in functions. In explicit flows the operation that generates the flow does not depend on the value of x such as in assignment `y=x`. By contrast, implicit flow occurs whenever a statement specifies a flow from some arbitrary z to y, but execution depends on the information stored in x. For example, in the following code the `if` statement causes an implicit flow from variable x to variable y.

```
y = 1; if (x == 0) y = 0;
```

The problem of information flow becomes security-relevant if, for example, x stores sensitive information and y is a system output. In this case, controlling how the information propagates in the program is crucial for protecting confidentiality. To approach this problem, the data in the program is associated with its *security level*, and security levels of data

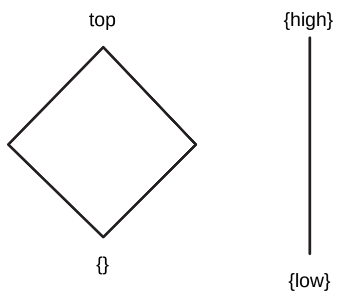

Figure 2.2: Examples of lattices

constitute a *security lattice*. The higher the security level is located in the lattice - the more sensitive is the information associated with this label. Example of lattices are presented in Figure 2.2.

The problem of *secure information flow* is concerned with preventing information leaks from high-security (more confidential) to low security (less confidential) program variables.

2.2.2 Security typed languages

Security-typed languages implement the information flow analysis by type checking. In these languages the variables and expressions in a program are augmented with annotations that specify their security level and policies on their usage. The compiler enforces these security policies at compile-time type checking phase. This involves little or no run-time overhead.

The most realistic programming languages supporting information-flow policies that have been developed so far are Jif, by Myers *et al* [23], and FlowCaml by Simonet and Pottier [25, 30].

2.2.3 Decentralized label model

The decentralized label model (DLM) [24] is a security model that allows *principals*, which are the entities whose information is protected by the model, to express their privacy concerns via *labels*. In DLM, principals (e.g. users, groups, roles) own, update and release information. Labels are used to guarantee confidentiality - every label consists of a set of *policies* that express privacy requirements. A privacy policy has two parts: an owner, and a set of readers, and is written in the form: *owner:readers*. By definition, an owner is implicitly contained in its readers set. A principal is allowed to read data if and only if it is contained in the reader sets of all policies of the label attached to the data.

Principal hierarchy In DLM, *the principal hierarchy* determines the *acts-for* relationship between principals: when principal p_1 can act for principal p_2 (e.g. `Professor` acts for `FacultyMember`).

2.2.4 Jif

Jif [23] is an extension of the Java language implementing the decentralized label model. In Jif, methods can be granted an *authority* to act for some set of principals. The authority controls the ability of the method to *declassify* the data: weaken or remove a policy in a label. This is possible if and only if a policy is owned by a principal that is part of the process authority.

An example of a label written in Jif syntax is *{Alice:Bob, Joe;}*. This label contains a single policy in which Alice is the owner; and Alice, Bob, and Joe are the readers. The label *{Bob:Alice; Alice:Joe;}* contains two policies. In this label Alice is the only principal present among readers of both policies. Hence, only Alice can read the data.

Variable types in Jif are composed of two parts: a regular Java type, such as `boolean`, and a security label, indicating how the value stored in this variable may propagate. For instance, the type `boolean{Alice:Bob}` represents a boolean that Alice owns and Alice and Bob can read. The bottom security level corresponding to public data has label {} (with the empty list of policies).

In security-typed languages implicit flows are controlled by introducing a *program-counter label (*$\underline{\text{pc}}$*)*. This label tracks dependencies of the program counter (a register in the CPU that contains the address of the next instruction to be executed).

Recall the example in the previous paragraph which we repeat here with the following variable definitions:

```
int {Alice:} x;
int {} y = 1;
...
if (x == 0)
      y = 0;
```

Here, the $\underline{\text{pc}}$ in the branches of the `if` statement captures the dependency on x and thus has the label {Alice:}. The assignment statement is rejected by the compiler because the affected variable is less secure than the $\underline{\text{pc}}$.

Method declarations

Method declarations in Jif may be annotated with two optional labels, called the *begin-label* and the *end-label*. Begin label is essentially a lower bound on the side-effects of the method. That is, Jif prevents calling a method if $\underline{\text{pc}}$ at the invocation point is higher than the begin-label of the method being invoked.

By default, if no begin-label is specified in the declaration of the method, it is assumed that the method has no side effects, and that it can be called regardless of the $\underline{\text{pc}}$ of the caller (i.e. from any context). Methods with side-effects, however, should have explicitly indicated begin-labels.

Arguments of the method may also be labelled. Label of an argument denotes the lower bound on the security level of the argument.

The end-label of a method carries information about how much can be learned by observing the method's termination behaviour — whether the method terminates normally or raises an exception. After the method invocation the $\underline{\text{pc}}$ of the caller becomes affected by the end-label of the method being called. End-labels are necessary if the termination path of the method may give out some information to the caller. Individual exceptions and return value may be labeled separately as well. An example of a method declaration is

```
public boolean{Alice:Bob} validate{Alice:}(String{} s, int{} hash):{Alice:}
```

In this example, the function `validate` 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:}` .

Method constraints

Jif allows three different constraints in method declarations:

- `authority(`$p_1, \ldots, p_n$`)` — list of principals that this method is authorized to act for.
- `caller(`$p_1, \ldots, p_n$`)` — list of principals whose authority the caller of the method is required to possess in order to run this method.
- `actsFor(`p_1, p_2`)` — this constraint prevents the method from being called unless the specified actsFor(p_1 acts for p_2) relationship holds at call site.[1]

Exceptions

There is one semantical difference between Jif exceptions and Java exceptions: in Jif all runtime exceptions have to be handled, since otherwise it would be possible to leak information via them. An example of how runtime exceptions can be maliciously used

```
public class IntegerLeak {
    private int {Alice:} secret;
    public int{Alice:} div(int{} a) {
        return a/secret;
    }
}
```

If variable `secret` is zero `ArithemeticException` is thrown in the method `div`. Observing whether this exception takes place exposes some information about the value of `secret`. That is why it is necessary to treat runtime exceptions as ordinary exceptions.

Parameterized classes

Jif allows classes and interfaces to be parameterized over labels and principals. This introduces another level of polymorphism and is useful for building reusable data structures. For example, instead of writing two separate `Player` classes for `Alice` and `Bob` which would differentiate only in the labels of the corresponding variables one can write single class `Player[P]` parameterized over principal variable `P`. Later in the instantiation the principal parameter is substituted with 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;
    private PermutationMatrix[{P:;L}]{P:;L} matrix;
    ...
    public void finishCardDraw{L}():{L}
    throws MPException where caller(P){
       ...
    }
}
```

This class is parameterized over a principal `P` who owns sensitive information stored in an instance of this class and a label `L` - that denotes the label that will be assigned to low data. The variable `name`, that in this example stands for the name of the player, is low. Thus it is labeled as `{L}`. Sensitive data like `keyPair` contains a pair of encryption keys - both public

[1]This feature is not used in this work.

and private. Because the private key which is stored in that variable is of a high security level, `keyPair` has a label `{P:}`. The variable `matrix` contains the secret permutation matrix. Note that the class `PermutationMatrix` is another class parameterized over a label. There are two labels that appear in the definition of the variable `matrix`. The first one is a parameter of the class, while the second is the label of the object `matrix`. Use of parameters in methods is presented in the declaration of the method `finishCardDraw`. For example, the begin-label is expressed via `{L}`, which implies that side effects of this method are visible on the level `{L}` and that the method can not be called from a context that is higher than `{L}`. Similarly, the end-label `{L}` says that the $\underline{pc}$ of the caller will be affected by the label `{L}` after the call. Constraint `where caller (P)` requires the caller of this method to have an authority `P`.

This class may be instantiated with principal Alice and bottom label as follows:

```
Player[Alice, {}] player1 = new Player[Alice, {}]();
```

Now `player1.name` has type `String` and label {}. Similarly `player1.keyPair` has label `{Alice:}`. In this example the class `PermutationMatrix` is instantiated over `{P:;L}` which for the variable `player1` is essentially `{Alice:}`.

Class parameterization is also important for preventing leaks about information. This is best illustrated in the following example.

```
class X {
   private int {this} p;
   public int {this} getP() {
     return this.p;
   }
   public void setP{this} (int {this}  n) {
     this.p = n;
   }
}
...
int {high} secret;
X {high} x1
X {low} x2 = new X();
x1 = x2; // looks safe, since the flow is upwards
x1.setP(secret); // leakage: secret is now visible as x2.p!
```

Listing 2.1: Leakage example

The label `{this}` is the label of an instance of the class. The label `{this}` is an example of a *covariant label*. Labels that were discussed in previous paragraph are *invariant labels*. The difference between the two kinds of labels is that non-final variables may be only labeled with invariant labels, while final variables may be labeled with both invariant and covariant labels. Thus, example above can be rewritten by introducing invariant label that fixes the flow.

```
class X[label L] {
   private int {L} p;
   public  int {L} getP() {
      return this.p;
   }
   public void setP{L} (int {L} n) {
      this.p = n;
   }
}
...
```

```
int {high} secret;
X[{high}] x1;
X[{low}]  x2 = new X[{low}]();
x1 = x2; // not allowed, because x1 and x2 are parameterized over
         // different labels
```

Listing 2.2: Leakage example

The flow that is introduced in previous example by assignment `x1=x2` is now prevented by Jif's type system. The cost for this is that both operands of an assignment statement must be parameterized over exactly the same labels.

Covariant labels, on the other hand, do not have such restriction. Indeed, final variables are the only ones that may be used with covariant labels. They can not change their values in any method except a constructor. Thus, it is safe to make assignments between objects at different security levels, since the final variables of these objects won't be affected after the assignment anyway.

Array labels

Arrays in Jif also have two labels: one for the elements of array, the other for the array itself.

Having only one label for arrays is not sufficient. Arrays are mutable data containers. Suppose that arrays did not have a separate label parameter. Listing 2.3 illustrates a laundering attack that is possible in this case. 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`. However this value can be observed through the original array with unrestricted label {}.

```
int[]{}  lowArray;
int[]{L} highArray;
int{L} secret;
...
highArray = lowArray; // allowed
highArray[0] = secret; // leakage! now low_array[0] == secret
```

Listing 2.3: Possible leakage with arrays having only one label

This argument also applies to the parameterized classes discussed earlier.

Examples of array declarations are

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

The first array denotes Alice's hand of cards. The bottom label {} stands for the size of the array — indeed it is publicly known *how many* cards a player has received during a game. In contrast, the *values* of the actual cards are secret to others. Therefore, elements of the array are labeled as `{Alice:}`. The second declaration shows an array `available` with low elements.

Declassification

Many secrets have a lifetime, after which they are not secrets anymore. Controlled information release or *declassification* is an important aspect of security-typed languages. This is a very powerful feature that Jif supports and that allows one to write practical programs. Without the possibility to declassify data the domain of applications that benefit from security-typed languages would be very limited.

There may be many reasons why one needs to declassify high data. For example, the underlying security protocol may require a secret key to be revealed in a concluding phase of the protocol. Or, a restrictive label of a private encryption key may propagate into a digital signature which has to be attached to a public document. In this case, the calculated signature has to be declassified. When checking a password, the result of comparing a user's guess with the actual password has to be returned to a user who tries to login.

In security-typed languages it is safe to move data to a higher position in the security lattice. With declassification it is possible to relabel program variables so that the resulting label is less restrictive than the original.

The declassification in Jif is expressed via `declassify` statement. The process is required to possess sufficient authority to declassify an object. An example of a declassification statement is:

```
y = declassify (x, {});
```

Here the `declassify` statement returns the value of x relabeled to {} that is assigned to y. The authority required by the process is essentially the "difference" between the label of x and the label which is specified as an argument to the declassify statement.

One can also declassify the $\underline{\text{pc}}$ label of the current process. An example of such declassification is

```
declassify({}) {
   ...
}
```

In this example the code in the brackets is executed with the $\underline{\text{pc}}$-label downgraded to bottom.

Chapter 3

Implementation

This chapter discusses the implementation [3] that has been done in this work. Following the description of the mental poker protocol in previous chapter, we will now present three different implementation of this protocol. One of them is in Java, the two remaining in Jif.

3.1 Methodology

Java reference implementation The first implementation has been done in Java. The motivation for this is that this implementation shows how the protocol could be implemented in ordinary situations by Java programmers. Another reason is that debugging Jif programs often becomes too difficult. This program implements protocols 1-4 as they are described in the previous chapter for two players (Alice, Bob). It can be also extended to multi-player poker games. In this implementation, we have developed the main functional part of the program, where the emphasis is on implementation and debugging of cryptographic routines.

Jif implementation The second implementation consists of lifting the Java version to Jif. The goal of this implementation is to study the difference between Java and Jif programs; to find out what kind of assurance we have with Jif programs and possible leaks in Java implementation. In short, lifting Java program to Jif involves the following steps:

- writing signatures for necessary Java API classes
- changing some of the classes to Jif analogues
- parameterizing classes over labels and principals
- assigning labels to class fields
- assigning begin and end labels to functions and arguments
- handling runtime exceptions
- writing helper functions for declassification of large data structures

Note that, there is no particular linear dependency between these steps and that the process of lifting may consist of a number of iterations and repetitive refactoring.

Jif distributed implementation The third implementation is about distributing the game physically among two system processes, and using input/output to synchronize them. This includes

- writing helper functions for serialization
- writing a synchronization program (written in Python)
- changing the logic of the coordinating process to take care of the distribution

For both Jif implementations we assume the presence of two principals: Alice and Bob.

Next sections describe the structure of the code for each implementation and the approach taken to do every task.

3.2 Java reference implementation

Figure 3.1 depicts the class diagram of the Java implementation. Here follows a brief textual description of the classes and their purpose.

As a basis of all arithmetic operations this implementation uses the `java.math.BigInteger` class.

`PHInteger` is the primitive class for storing encrypted values and arithmetic operations over them (where abbreviation PH stands for privacy homomorphism). The class `PHCrypto` provides a set of operations for generating privacy homomorphic cryptosystem keys, and encrypting/decrypting values. The classes `CardVector` and `EncryptedCardVector` are vector representations of cards in plain and encrypted forms respectively.

The distributed notarization chain (DNC), its constituting links and data attributes are implemented in classes `DNCChain`, `DNCLink` and `DataFieldAttribute`.

The class `Player` contains the main logic of the application related to the game protocol, and implements protocols 1-4 of the card game.

The class `MPTable` coordinates the process of the game. The member function `play()` of this class is called when the application starts. As a part of the initialization, it creates an instance of DNC. The reference to this chain is later passed as an argument to the constructor of the `Player`. Thus, two players named `Alice` and `Bob` are created, both of them having reference to the same chain object. The `play()` function controls the order of instructions for the card draw protocol (protocols 2, 3) and for the verification phase (protocol 4).

Summary of class' description for the Java implementation is listed in Appendix A.1.

3.3 Jif implementation

Following the security requirements we have approached the problem of the secure implementation with the security lattice presented in Figure 3.2. The sensitive information of players carries the labels `{Alice:;}` and `{Bob:;}`. The data passed around between players is downgraded to the {} level (bottom).

3.3.1 Writing signatures

To compile against existing Java API classes that the Java implementation uses, the Jif implementation requires Jif signatures to be written for these classes. Note that even though

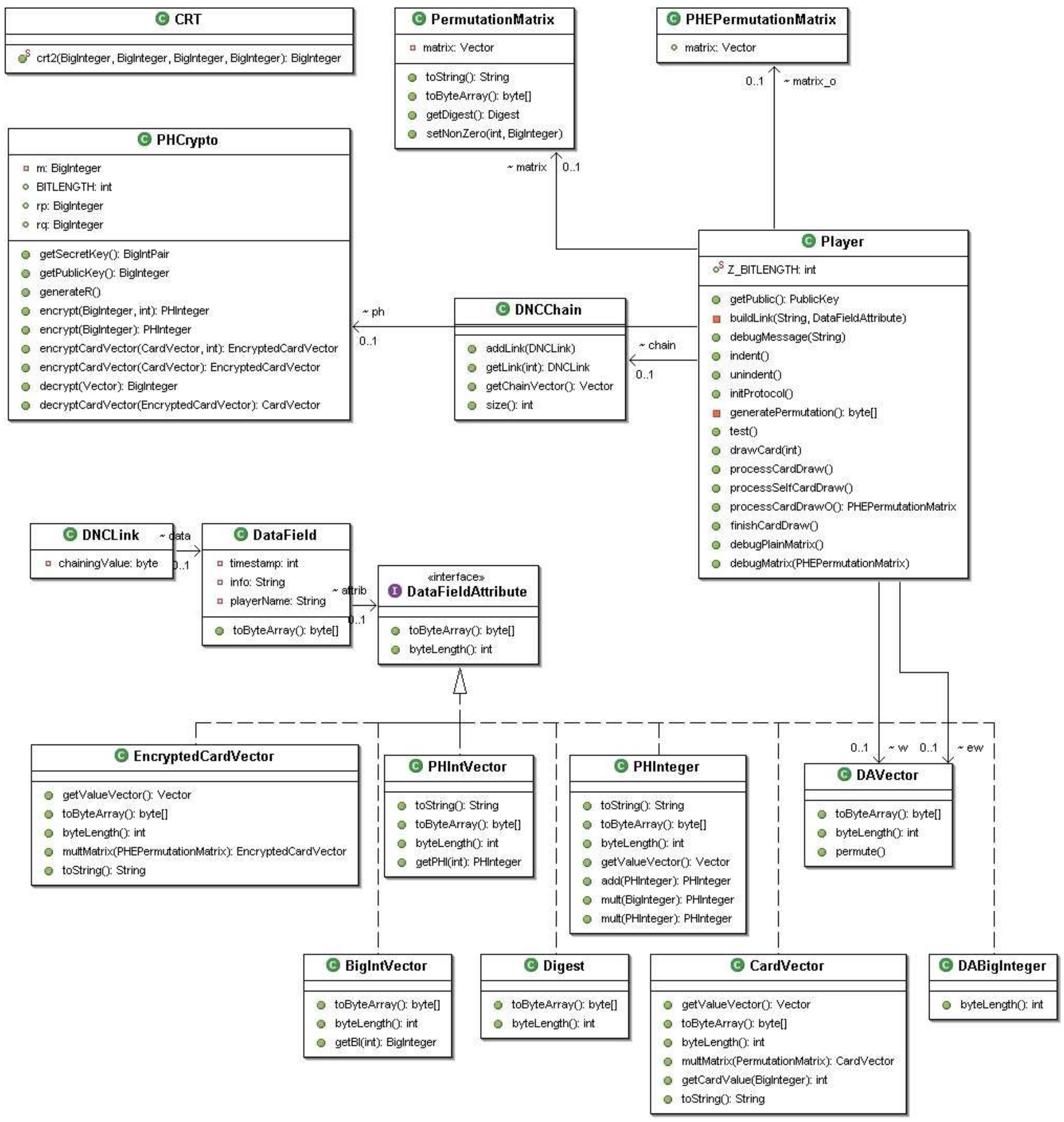

Figure 3.1: Class diagram for Java implementation

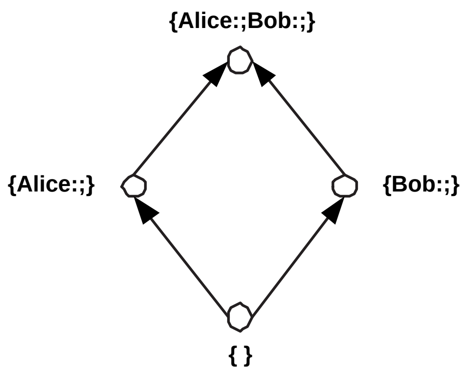

Figure 3.2: Security lattice for the Jif implementation

writing signatures is a relatively easy task compared to Jif programs, this should be done with care, since it is very easy to misuse this feature (see Section 4.4.2 about problems and vulnerabilities related to signatures).

List of signature written for the Jif implementation can be found in Appendix A.2.

3.3.2 Changing to Jif analogues

Sometimes it is possible to avoid mechanical process of writing class signatures, but instead with little code refactoring reuse existing Jif analogues for some classes. For example, in the Jif implementation all instances of `java.util.Vector` are replaced by `jif.util.ArrayList` which comes with the Jif distribution and provides nearly the same set of operations. The motivation for such a replacement is that `jif.util.ArrayList` is itself written completely in Jif and thus is more secure with respect to method labels and is more convenient for Jif programs than class signatures.

This change, in turn, causes most of the classes in order to be storable in the Jif `ArrayList`.

3.3.3 Parameterizing classes

This implementation uses class parameterization heavily. Classes that implement interface `DataFieldAttribute` are parameterized over an invariant label `L`, which stands for the security level of the information stored in the instances of these classes.

DNC related classes `DAVector, DNCChain, DataField` are also parameterized over label `L`.

The class `MPTable[principal P1, principal P2]` is parameterized over two principals - the players participating in the game.

The class `Player[principal P, label L]` is parameterized over the player principal `P`, and label of the output channel `L`.

The class `Declassifier[principal P, label L]` is parameterized over the principal `P`, whose authority the caller of the helper functions should possess, and the label `L` which is the label of the declassified data structures. Thus the data is declassified from the level `{P:;L}` to the level `{L}` .

3.3.4 Assigning labels to class fields

A general rule for assigning a label of a variable is that it should be consistent with its security purpose. This basically means that it should be clear for the developer or designer of the system what variables contain sensitive information and and what the level of sensitivity of each variable is.

The only exception from this rule is the case when low-level data is operated in the high context only, and is not used anywhere in the low context.

Examples of variable declarations in the `Player` class are presented in Tables 3.1 and 3.2. Recall that the class `Player` is parameterized over the principal of the player `P` and the label of the cooperating runtime system. Therefore, in our implementation label `{P:;L}` corresponds to a high label and `{L}` to a low one.

The `ph` program variable in the Table 3.1 corresponds to the instance of the homomorphic cryptosystem `PHCrypto`. This class is instantiated with parameter `{P:;L}` so that all cryptographic operations are performed at a high security context. This implies that exceptions that may occur during cryptographic operations are also high. Thus the end-labels of the methods

Java	**Jif**
`PHCrypto ph`	`PHCrypto[{P:;L}]{P:;L} ph`
`byte[] p`	`byte{P:;L}[]{P:;L} p`
`PermutationMatrix matrix`	`PermutationMatrix[{P:;L}]{P:;L} matrix`
`KeyPair keyPair`	`KeyPair{P:} keyPair`

Table 3.1: Examples of high variable declarations

Java	**Jif**
`boolean[] available`	`boolean{L}[]{L} available`
`String name`	`String{L} name`
`DNCChain chain`	`DNCChain[L]{L} chain`

Table 3.2: Examples of low variable declarations

of this class are high too, which in turn affects the $\underline{pc}$ of the caller. Next, the `p` byte array corresponds to the player's secret permutation. That is why the first label of this array is `{P:;L}`. The permutation matrix depends on the permutation array; therefore its label follows the one of the array and is `PHCrypto`. The last example from Table 3.1 is a variable corresponding to the private and public key pair used for signing DNC links. The public component of this key pair is not secret. On the other hand its private component is sensitive data, so the entire variable `keyPair` has to be labeled as high.

Array `available` on Table 3.2 indicates what cards are available during the game. This is public information, since all players know, for example, whether the first card in the shuffled deck has been dealt of not. Another example of low data is the `name` of a player which is used for output. The `chain` variable is an instance of `DNCChain` class - distributed notarization chain used in the protocol. All communication between players in the game happens via publishing links to the chain. This is a part of the system output and should be low as any output that is observable at the low-level.

3.3.5 Assigning labels to functions and arguments

Recall that begin and end-labels in method declarations are related to side-effects in the program: in this implementation we identify side effects in Jif programs by the following signs:

- assignment to non-final member variable
- assignment to mutable data structure (array, any class instance)
- calling a method with side-effects

Listing 3.1 illustrates some methods with labels in declarations.

```
public PHInteger[L]{L} mult{L}(DABigInteger[L]{L} b):{L}
throws (IllegalArgumentException) {
    ...
}
public static CardVector[{P:;L}]{P:;L}
upgradeCardVector{P:;L} (CardVector[L]{L} x) {
    ...
}
```

Listing 3.1: Examples of begin-label and end-label declarations

The method `mult` on Listing 3.1 returns the encrypted integer, whose value is `this * b`. In this method the label of the argument `b` is restricted to be the same as the label of the current instance, so the result is of the same label. Otherwise, we do not allow multiplication of variables that have different labels. The obvious possible side effects then have the same label `L`.

The method `upgradeCardVector` on Listing 3.1 returns the copy of the same card vector that is passed to it via the argument, but upgraded to a higher level on the security lattice. This involves creation of the variable at the level `{P:;L}`, hence the begin label of the method. No exceptions are thrown by the method, thus there is no end-label.

3.3.6 Catching/throwing runtime exceptions

Because unchecked exceptions can serve as covert channels Jif requires all runtime exceptions (except `FatalError`) to be handled.

In our implementation we use one of the following approaches:

- Declare and throw: this is the most straightforward approach when all exceptions are declared in the method header and the responsibility to handle them is passed to the method caller. An advantage of this method is simplicity for the method writer, however a disadvantage is that these exceptions are still have to be handled by the method caller. Moreover their origin becomes somewhat obscured for the method caller, since there may be multiple points in the method that could throw the same exception. In addition, declaring exceptions as throwable may in its own turn affect the end-label of the method.

- Avoiding exceptions: the Jif compiler has a simple, yet useful, not-null dataflow analysis that allows it to detect if local variables are not null at particular program points. Consider the following code snippet with `BigIntPair` class declaration and `toString()` method implementation.

  ```
  public class BigIntPair[label L] {
      private final DABigInteger[L]{L} x;
      private final DABigInteger[L]{L} y;
      ...
      public String{L} toString() {
          DABigInteger[L] x= this.x;
          DABigInteger[L] y = this.y;
          if (x == null || y == null) return "";
          return "(" + x.toString() + ",␣" + y.toString() + ")";
      }
  }
  ```

  In the `toString()` method, two local variables are declared and class fields are assigned to these variables. Next, we check if either of them is null and return from the method with the empty string in that case. Otherwise, it is safe to call `x.toString()` or `y.toString()`, since `x` and `y` are already ensured to be non-null.

- Catch and ignore: there are two scenarios when a programmer wants to ignore the exception:

  - either there is a sufficient guarantee that the exception will not be thrown.
  - or, a programmer deliberately hides the presence of an exception

- Catch and handle: handling exceptions otherwise.

There is also a pattern on handling `NullPointerExceptions` for method argument, described in the patterns section.

3.3.7 Modularizing declassification

Declassification is an essential part of Jif programs. The `declassify` statement that Jif provides allows declassifying primitive data types such as `int, boolean` or references to mutable data structures.

However, declassification of mutable data structures together with elements, such as instances of arrays or classes that are parameterized over labels requires the fields of these structures to be declassified separately. Thus, in order to declassify, say, an array, one has to declassify every element of the array. If the structure in question is an instance of some class, declassification implies traversal through all fields of this class.

For this purpose our Jif implementation includes a special class `Declassifier` that contains number of static helper methods for declassification and upgrade of different data types. The upgrade methods are applied when a low data needs to be relabeled for a usage at a higher security-level. For example, in our implementation, the DNC is low. Therefore, objects that are obtained from the DNC are low as well. In order to apply a player's secret permutation to a card that is obtained from the DNC this card has to be upgraded to the level of secret permutation.

The `Declassifier` class is parameterized over principal `P` whose authority is used for declassification and label `L` to which the data is relabeled. It is assumed that the high label is `{P:;L}`. The label `L` is later substituted as a second argument of a `declassify` statement. Similarly, upgrade functions relabel values from level `L` to level `{P:;L}`.

Helper functions in this class cover array types and classes that are used in our implementation, e.g: byte and int arrays, class `Digest`, class `Attribute`, class `DAVector`. Details about the implementation of `Declassifier` are in the patterns section.

3.3.8 Design of the cryptographic library

This paragraph describes the design issues related to the cryptographic library that has been developed for the protocol implementation.

The class `PHInteger` is a data structure for storing encrypted information. The goal of this class is to implement arithmetic operations over encrypted values: *addition* and *multiplication* of two encrypted values and *multiplication* of an encrypted integer to a *scalar*. The class `PHInteger` is parameterized over a single invariant label `L`, which stands for the sensitivity level of the information stored in the instance of the class. This label is also used to limit the operands of the arithmetic operations to the the same level.

The class `PHCrypto` provides a set of cryptographic operations, including generation of a secret encryption key pair, functions for encryption and decryption of integers and card vectors. This class is also parameterized over an invariant label `L`, which stands for the sensitivity level of all members, arguments to functions and return values of methods.

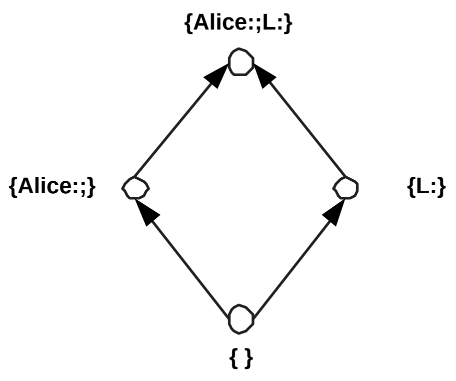

Figure 3.3: Security lattice for the Distributed Jif implementation

3.3.9 Custom exception class

In order to accommodate for protocol related exceptions, the Jif implementation includes a custom exception class `MPException`. This class is used for encapsulating some exceptions and providing textual information about the errors, as well as, hiding the internal structure of an exception if it has been thrown at a level higher than the method's return-label.

3.4 Distributed Jif implementation

Distribution of the previous implementation has been done with the purpose to see a "real-world" application of Jif. In this implementation players are running as different processes and are isolated from each other as they are running on different hosts. They exchange messages by means of a third process - binder (also referred here as a filtering process). We don't impose any security requirements on the binder process.

For the sake of simplicity we decided to use standard input/output as a communication medium. The distribution scheme works as follows: two player processes are started by the filtering process so that I/O pipes of both processes are connected to the filtering process. If a player process needs to send a message to the other process, it prepends the message with prefix ### in order for that to be distinguishable by the filtering process. When the filtering process finds this header in a message that it reads from the player's output channel, it forwards the message to the other player. Otherwise the message is considered simple console output and is printed to the screen. Figure 3.4 depicts this scheme.

Figure 3.3 displays the security lattice for one of the players (Alice). Here `L` is the label of the run-time environment. Sensitive information in the program has the label `{Alice:;L}` and it is declassified to `{L}` before being sent over the network. The lattice for the other player's process is similar to this.

Introducing distribution to the Jif implementation involves a few steps:

1. Using a class `Communicator` instead of `MPTable` — the functionality of this class is similar to the functionality of `MPTable`. While `MPTable` coordinates both players in one process, `Communicator` observes the order of the players' messages.

2. Introducing classes `SerializeWriteHelper` and `SerializeReadHelper` for data serialization. With the help of these classes we are able to convert data into strings and pass them around. Note that we are not really able to use Java serialization with Jif objects,

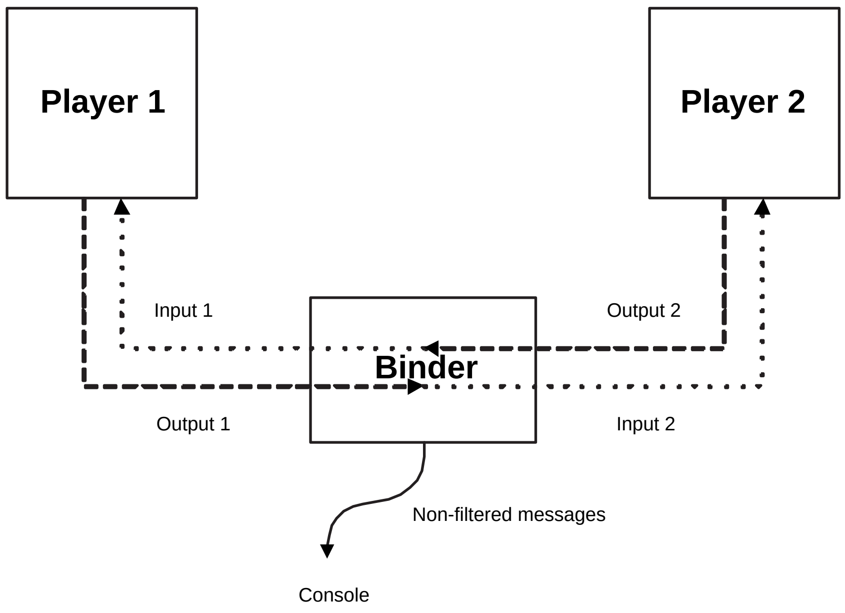

Figure 3.4: Distributed implementation architecture

since Jif forbids casting classes with different number of class parameters. With Java serialization objects of type `Object` must be casted back to their actual classes. However, in Jif it is not allowed to cast classes that have different parameters since this may lead to the laundering attack similar to the one described in Section 2.2.4.

We have to admit that this way of achieving serializability is not well scalable. One would benefit a lot from getting Jif parameterized classes to work with Java serialization.

3. `binder` - A small synchronization utility, written in Python [1]. It connects two processes via pipes, filtering out marked output and passing this data between processes.

Chapter 4

Evaluation and Discussion

4.1 Comparison of the three implementations

The previous chapter presents three different implementations. This section discusses the differences between them and how they are related to each other.

4.1.1 Comparison of the Java implementation and the Jif implementation

Lifting Java programs to Jif is not straightforward. The patterns section that follows later in the report presents a number of programming patterns that have been discovered during this work and are proposed to help Jif programmers. In our implementation the resulting Jif code is a product of a number of refactoring iterations on the initial Java version. The main impact is caused by the security label annotations of program variables. These labels propagate further into the begin and end-labels of the methods. The direct outcome of this propagation is that existing Java methods are rewritten in such a way that the new version either fixes discovered flow or declares it explicitly either via a declassification statement or via a method header. This increases general confidence in the program, in particular in protecting confidentiality of sensitive variables.

Explicit declassification helps find out exactly what data is downgraded. Thus, the intended leaks in the program are reduced to the declassification points. The discussion of the declassification points in our implementation is presented later in this chapter.

Insecurities

One of the interesting insecurities that has been discovered in our Jif implementation is related to exceptions that take place at different security levels. Listing 4.1 is a simplified example of such insecure code:

```
public class ExceptionLeak[label L] {
    private int{} readInput{}() {    ...     }

    public void exceptionLeak{}() throws Exception{
        while (true) {
            int{} x = readInput();
            highMethod(x);
        }
    }
10      private int{L} highDenominator;
11      private int{L} highCounter;
12      private int{L}[]{L} highArray;
13      private void highMethod{}(int{} x)
14      throws ArithmeticException, NullPointerException,
15          ArrayIndexOutOfBoundsException {
16              highArray[highCounter++] = x/highDenominator;
17      }
18  }
```

Listing 4.1: Example of leaks via exceptions

Any of `ArithemeticException`, `NullPointerException` or `ArrayIndexOutOfBoundsException` are thrown in high context and reflect problems in high variables `highDenominator`, `highArray` or `highCounter`. Therefore, the caller of `highMethod` on line 7 obtains information not only about the successful termination of the method but also detailed facts of what kind of exception occurred and, in more complicated scenarios, the stack trace back to the origin of the problem.

The patterns section describes in details how one can prevent such leaks.

4.1.2 Comparison of the Jif implementation and the distributed Jif implementation

While the second implementation reveals lots of important security issues the third one is more interesting from the practical point of view. From that perspective the second implementation may be considered as an intermediate step towards the final distributed version. The second and third implementation are both done in Jif and most of the code of the second implementation is reused without any changes in the third. Thus, the third implementation benefits from security guarantees that are achieved in the second version and in addition is simple, but yet fully working example of a distributed program written in a security-typed language.

Since the distributed Jif implementation encompasses all features of the Jif implementation, in the following discussion when referring to the Jif implementation we will assume the distributed Jif implementation, unless it is otherwise specified.

4.2 Security assurance

Recall the security goals for the poker protocol, presented in the Section 2.1.2 that are provided by the Castella-Roca *et. al* protocol.

1. Uniqueness of cards
2. Uniform distribution of cards
3. Absence of trusted third party
4. Cheating detection with very high probability
5. Complete confidentiality of cards
6. Minimal effect of coalitions

Even though the implemented protocol addresses all these goals, one should be aware that the implementation of the protocol may violate some of the properties achieved by the protocol designers. Let us review these properties and look at how the three different implementations address these goals.

The first and second properties rely on the random number generators, supported by the Java API. The third property, which is crucial to the design of the protocol, is not violated by our implementation. That is the implementation does not introduce a TTP as there are only two principals — the players. Because the sixth property is specific to the design of the protocol it is not violated by the implementation either. Cheating detection is implemented by logging the links of DNC and using it for verification as described in protocol 4 (cf. Section 2.1.4).

The Jif implementation enforces card confidentiality (property 5) by usage of high labels for sensitive information. The following list presents confidential program variables that appear in the Jif implementation.

- encryption key (instance of cryptosystem)
- signature key which is used for digital signing of messages
- player's hand—a set of cards a player obtains after shuffling and drawing
- secret permutation π_i and corresponding variables:
  - permutation matrix Π_i
  - copy of the original permutation matrix (according to the protocol, the original matrix changes during the game — thus we keep a "clean" copy that is revealed in the verification phase)
  - deck of clear text cards w
  - deck of permuted encrypted cards w'

By labeling these variables as high, we restrict flow from them to the system outputs.

Jif's type system prevents unintended flow of sensitive information unless it is otherwise specified by the `declassify` statement. It is crucial to minimize the number of declassifications in the code, so that they can be manually reviewed and judged.

4.3 Authority and declassification

Correct label assignment reduces the possible places for information flow to the points where declassification occurs in the program. Declassification is possible if and only if the running process has enough authority to relabel the data. Thus, the ability to grant a class or a method an authority is a useful but also potentially dangerous feature of security-typed languages since this authority may be misused for inappropriate declassification of confidential information.

In the Jif implementation the authority is granted in the following places.

- class `Communicator`, function `play()` - playing game.
- class `Player`, function `getPublic()` - returning public key of the player.

GR	PT	WHO	WHERE (CODE)	WHERE (PROT)	WHAT	WHEN
I	1	Anyone	`getPublic()`	—	Public key for signature	Before game start, before seal is broken
	2	Player	`drawCard()`	—	Public security parameter	Before game start, before seal is broken
	3	Player	`computeLink()`	Chaining	Computed chaining value	Any time
	4	Player	`initProtocol()`	P 1, *2 d*	Prime z_i used as a modulos for Π_i s	Before game start, before seal is broken
II	5	Player	`initProtocol()`	P 1, *2 g*	Commitment Cp_i to the perm. matrix Π_i	Before game start, before seal is broken
	6	Player	`initProtocol()`	P 1, *2 k*	Computed vector D	Before game start, before seal is broken
	7	Player	`initProtocol()`	P 1, *2 l*	Computed vector E	Before game start, before seal is broken
	8	Player	`processCardDraw()`	P 2, *2 b*	Permuted card w_i	During game, before seal is broken
	9	Player	`processSelfCardDraw()`	P 2, *4 d*	Encrypted permuted card w'_j	During game, before seal is broken
	10	Player	`processCardDraw0()`	P 2, *5 c*	Encrypted permuted card w'_j	During game, before seal is broken
III	11	Player	`finishCardDraw()`	—	Decryption flag	During game, before seal is broken, after player obtains card
	12	Player	`revealPrivateKey()`	P 4	Secret encryption key	After game end, after seal is broken
IV	13	Player	`revealPermutation()`	P 4	Secret permutation	After game end, after seal is broken
	14	Player	`revealMatrix()`	P 4	Secret permutation matrix	After game end, after seal is broken

Table 4.1: Declassification points

The need for authority in these places is explained below.

In the Jif implementation there are 14 declassification points. Table 4.1 presents these points. For each declassification it states *who* declassifies data, *where* in the code the declassification occurs, *what* exactly is declassified, and *when* this declassification may happen (the last column uses the notion of seal defined in group IV below). These columns correspond to dimensions of information release [27].

As it is shown in the table we can divide the nature of declassification points to four different groups:

1. Declassification of naturally public data (1–2) : the keys for signature are generated using Java's `java.security.KeyPairGenerator`. The obtained `java.security.KeyPair` class instance contains both sensitive (private key) and non-sensitive (public key) data. That's why, in order to return the public key, we first obtain separate high copy of the key, and then, declassify it. Similarly in the `drawCard` method the public parameter of the homomorphic cryptosystem is extracted from an instance of the `PHCrypto` class. Again we obtain the high copy of this parameter and declassify it separately. These declassifications are safe, since they do not affect the sensitive part of the key pair or of the cryptosystem.

2. Declassification related to building links in the DNC (3–10): this is the largest group of declassification points, related to the nature of the underlying protocol. There are a few points in the protocol where a player has to construct a new link.

   The first such point is related to the signature of the DNC link. Because computation of the signature involves a private key, the result gets tainted by the high label of the private key and also becomes high. Here we have to rely on the cryptographic properties of the calculated signature and assume it is safe to declassify the computed result. The obtained declassified value is used in the construction of the new link.

   The rest of the declassification points in this class is related to the different steps of the protocol. These are encrypted values created in the context with a high $\underline{pc}$-label and the motivation for their declassification is similar to the one for the signature. Table 4.1 indicates these points. The **Protocol** column shows the steps of the protocol that justify each particular declassification. We can assume that these declassifications are safe as far as we can trust the underlying crypto-protocol.

3. Declassification of the success flag in `finishCardDraw()` (11). See the related pattern in Section 4.5.5 for motivation and details.

4. Declassification of sensitive information for verification (12–14): After the game completes, the protocol requires players to exchange their private keys and secret permutations in order to verify the fairness of each other and detect cheating. This is a common scenario for security protocols that rely on bit commitment. It is important that these functions should not be used earlier in time than they are supposed to. Language-based features of Jif are currently not powerful enough to support such complicated properties. For this reason the mixture of conventional programming techniques with language-based features is used here: a so called "seal" is a boolean flag that changes its value only once after initialization. In this implementation, the seal is initialized in the constructor of the `Player` class. Its integrity is checked in the methods that implement protocols 1–4. The seal is *opened* once the sensitive information that it protects has to be released. This is done in the methods that declassify the keys and secret permutation. The next time there is a call to the method that assumes the seal's integrity a runtime exception is thrown indicating that this method call violates the security properties of the protocol. That is one is not allowed to declassify the data and continue functioning as if the declassification had never happened before. The implementation details of this technique are presented in the patterns section.

The existence of at least four different classes for declassification opens up the question of applying different security policies for declassification. Trivially there exist much more scenarios where the declassification must be used in one way or another.

In this implementation the class `Declassifier` does not have any authority and all methods in the class require the caller to have sufficient authority in order to declassify the data. One could also imagine an alternative version of this class that would integrate the encryption with declassification, and combine these two operations transparently for the caller, so that the caller of the method could get a declassified but encrypted value without having the necessary authority for declassification. We can assume that this kind of declassification would be safe as the caller of the method receives encrypted value. However, we are not using this scenario in our implementation since it is the player who owns, encrypts and declassifies the data.

4.4 Pros & cons of programs in security-typed languages

4.4.1 General differences

This section discusses pros and cons of security-typed languages.

Pros

- The Lattice model allows a programmer to explicitly specify confidentiality level of program variables. Security type systems prevent information flow from high variables to low ones. Not only does this capture direct flows via assignments but also indirect flows such as side effects.

- Explicit declassifications statements are necessary for downwards information flow. This introduces a fine control over the points in the program where information is released. Rather than reviewing the whole program in search of the dangerous leaks, it is now only necessary to "zoom in" into the program contexts where declassifications occur. Then one can analyze the reasons for a particular declassification and justify if it is acceptable.

- Programs written in security-typed languages are composable — it is easy to reuse components for building larger programs.

- Efficiency is an important aspect of using security-typed languages for secure information flow. Program type-checking is fast, and there is little or no impact on run-time behaviour of programs.

Cons

- Practical difficulty in writing programs. Currently, there is a lack of proper tools and extensive documentation for writing programs in security-typed languages.

- Security typed languages can help programmers to identify unintended leaks in their programs, but they can not stop a malicious programmer from writing vulnerable code. For example, it is possible to declassify everything in the beginning and have a "security type checked" program that would still leak secrets.

4.4.2 Problems and vulnerabilities in Jif

Signature misuse

Providing Jif with signatures for Java written classes is a powerful feature that allows reuse of existing Java libraries in Jif programs. However, this feature has to be used with great care and is a potential place for introducing flows that are not captured by the Jif compiler. A signature is a file with Jif style method declarations. Jif programs are type checked against these headers, however during runtime execution the actual Java-compiled binary is used. The danger here is that the labels declared in the method header may not correspond the real code that used in the library. An example of such a weakness is a signature of `System.arraycopy` function from the current Jif distribution.[1]

```
public static native void arraycopy(Object{dst} src,
      int{dst} src_position, Object dst,
      int{dst} dst_position, int{dst} length)
   throws (IndexOutOfBoundsException,
           ArrayStoreException, NullPointerException);
```

As one can see, there is no begin-label in this method, that implies the absence of side-effects in this methods. However the copy of an array in the memory is an obvious side effect that should be affected in the begin-label of the method. Listing 4.2 is an example of how this weakness can be exploited.

```
public class TestLeak[label L] {
    private int{L}[] secret;
    private int{}[]  output;
```

[1]v. 1.1.1, released Oct. 2004

```
    public void leak() {
    try {
        System.arraycopy(secret, 0,
                 output, 0, secret.length);
        } catch (Exception ignored) {
        }
    }
}
```

Listing 4.2: Leakage via invalid method signature

In this example, function `leak()` calls `System.arraycopy` to copy data from the high array `secret` into the low array `output`. This is blindly accepted by Jif compiler since it trusts the method signature that has been provided.

Parameterized signatures In Section 2.2.4 we discuss how Jif prevents leakage in classes by requiring invariant labels to be the same in assignments. Consider Java class `X` on the Listing 4.3.

```
class X {
   private int p;
   public int getP() {
      return this.p;
   }
   public void setP(int n) {
     this.p = n;
   }
}
```

Listing 4.3: Java class `X`

In order to use the Java class `X` in Jif programs its signature must be written. Consider one such signature on Listing 4.4.

```
class X {
   public native int {this} getP();
   public native void setP{this}(int{this} n);
}
```

Listing 4.4: Example of malicious signature for class `X`

This code has a disadvantage — it is possible to leak information in the very same way as it is discussed in Section 2.2.4 and shown in Listing 4.5. In fact, the proper signature for this class needs to be parameterized over a label `L`. Listing 4.6 illustrates this.

```
int {high} secret;
X {high} x1
X {low} x2 = new X();
x1 = x2; // looks safe, since the flow is upwards
x1.setP(secret); // leakage: secret is now visible as x2.p!
```

Listing 4.5: Example of a leakage for class `X`

```
class X[label L]{
   public native int {L} getP();
   public native void setP{L}(int{L} n);
}
```

Listing 4.6: Correct signature for class `X`

Generally, a class should be parameterized if a variable of that class can be modified after the instantiation.

While flows of this kind are captured in pure Jif programs by the type-system, they are not prevented for class' signatures. It is the author of a signature who is responsible for its correctness.

Relabeling mutable data containers in Jif

Assume there is an array `x` of type `int{}[]{}` which we want to relabel to `int{Alice:}[]{Alice:}`. Assignment

```
int{Alice:}[]{Alice:} y = x;
```

is rejected by Jif type-checker as it may lead to laundering attack similar to the one described in Section 2.2.4. The solution to this is to create a separate copy of an array, upgrading elements one-by-one.

```
int{Alice:}[]{Alice:} x = new int[y.length];
for (int i = 0; i < y.length, i++)
    y[i] = x[i];
```

Similar code has to be written if one wants to relabel an instance of some parameterized class. Declassification is another example of relabeling, so the same argument applies when there is a need to downgrade an array or a class instance. As a consequence of this limitation, a programmer has to write code for relabeling (or declassifying) every complex data structure that is used on different security levels.

Section 3.3.7 discusses how we approach this problem in our Jif implementation, and Section 4.5.2 provides more details as well drawbacks of this approach.

Signature for key generation In the Jif implementation, Java's `java.security.KeyPair` class is used for storing information about keys pair used for digitally signing messages. The signature of this class looks as follows:

```
public final class KeyPair{
  public KeyPair(PublicKey{this} publicKey, {this} privateKey) {}
  public native PublicKey{this} getPublic();
  public native PrivateKey{this} getPrivate();
}
```

As it is described in Section 4.3 an instance of this class should be high, because it contains the sensitive private key. Declassification is applied when information about the public key is needed. This kind of declassification is safe because released information is naturally public.

By means of class signatures, it is possible to avoid this explicit declassification by parameterizing a signature over two labels — one for the private key, and the other for the public one. The signature then might look as follows:

```
public final class KeyPair[covariant label L, covariant label H]{
  public KeyPair(PublicKey{L} publicKey, {L;H} privateKey) {}
  public native PublicKey{L} getPublic();
  public native PrivateKey{L;H} getPrivate();
}
```

Here, `L` stands for the low label of the public key, and `H` stands for the high label of a private key. With such a signature explicit declassification is avoided — not because this signature is safer than the previous. The flow still exists but it is rather invisible to Jif now.

Runtime system

In the current version of Jif the runtime system provided by the compiler is somewhat inflexible. One of the problems is that the data obtained from the I/O channels is labeled with the label of a special runtime principal—the current user running the program. However, because the authority of this user can not be obtained by Jif programs, this label propagates further in the parts of the program that depend on runtime input.

Missing Java features

Although Jif extends a large part of Java, some useful features of Java are missing in it. The incomplete list of these is

- inner classes
- serializability for parameterized classes.
- `super` calls

The lack of inner classes and `super` calls have not proven a major difficulty. Most important in the context of this case study is the missing support of serializability for parameterized classes. As a result, serialization routines used in the distributed implementation need to be written manually for every class.

4.5 Programming patterns

Here we will show some programming patterns that one may find useful when programming in Jif, or lifting Java programs to Jif.

4.5.1 Checking arguments in Jif functions

Handling runtime exception in Jif can make code look messy and less readable. Consider the example of Java code in Listings 4.7

```
public boolean validate (byte [] oPerm,
PermutationMatrix oMatrix) {
    if (!oMatrix.validate(oPerm, z_i))
        return false;
}
```

Listing 4.7: Java code

Straightforwarding porting of this code to Jif may lead to the code as it appears on Listing 4.8

```
public boolean validate{L}(byte{L}[]{L} oPerm,
PermutationMatrix[L]{L} oMatrix):{L}
throws NullPointerException {
    if (!oMatrix.validate(oPerm, z_i))
        return false;
}
```

Listing 4.8: Jif code 1

That is, since we are using the `oMatrix.validate()` method, we have to either catch the `NullPointerException` or declare it as thrown. A `NullPointerException` exception may be thrown if an argument that is passed to the function is invalid. We can avoid this if we check whether the argument is null right away in the beginning of the function before we actually use it, and raise a more specific exception in that case. Listing 4.9 illustrates how the above code can be rewritten so that an `IllegalArgumentException` is thrown whenever an argument is `null`. Note that because of Jif's `NullPointerException` analysis it is not necessary to handle `NullPointerException` anymore for variable `oMatrix`.

```
public boolean validate{L}(byte{L}[]{L} oPerm,
PermutationMatrix[L]{L} oMatrix):{L}
throws IllegalArgumentException {
    if (oMatrix == null)
        throw new IllegalArgumentException
            ("validate:␣oMatrix␣null");

    if (!oMatrix.validate(oPerm, z_i)) return false;
}
```

Listing 4.9: Jif code 2

This pattern achieves cleaner code as all arguments are checked in advance before they are used, and the type of an exception that is thrown from the function is more specific rather than the generally ambiguous `NullPointerException`.

4.5.2 Declassification of large data structures

Declassification of big data structures may not be that easy as it seems at first sight. Jif provides a method `declassify` that allows one to declassify a variable. This however does not declassify the whole structure, if it is parameterized over some label. Consider class on Listing 4.10

```
public class IntPair[label L] {
    private int x;
    private int y;

    public IntPair(int{L} _x, int{L} _y) {
        this.x = _x;
        this.y = _y;
    }
}
```

Listing 4.10: Sample class

Declassifying variable `a` of type `IntPair[L]{L}` to type `IntPair[{}]{}`, provided that the caller has sufficient authority leads to cloning the object `a`. This is required in order to avoid the kind of laundering attack that is described in Section 2.2.4. Listing 4.11 presents the code for such declassification.

```
IntPair[L] b = declassify(a, {});
IntPair[{}] c = new IntPair[{}] (
    declassify(b.x, {}) , declassify(b.y, {}));
```

Listing 4.11: Declassification of IntPair class

That is, we are declassifying fields of `a` and basically, constructing a new object consisting of declassified elements.

```
IntPair[L]{L} b = new IntPair[L](a.x, a.y);
```

Listing 4.12: Upgrade of the label

The similar pattern applies if we want to upgrade the label of the variable from low to high, in order to process it on the higher level. Listing 4.12 illustrates this. This technique, even though it works for small classes, may become quite cumbersome when using larger classes. To overcome the problem, we have created a special class `Declassifier` which contains declassification routines for required data structures. `Declassifier` is parameterized over a principal whose authority is used for declassification and a label to which the data is declassified. Listing 4.13 is an example of how such class can be implemented.

```
1  public class Declassifier[principal P, label L] {
2      // let no one instantiate this class.
3      private Declassifier() {}
4
5      public static byte{L}[]{L}
6      declassifyByteArray{L}(byte{P:;L}[]{P:;L} x1):{L}
7      where caller (P) {
8          byte{P:;L}[]{L} x = declassify (x1, {L});
9          if (x == null) return null;
10         int t = x.length;
11         byte{L}[] y = new byte[t];
12         try {
13             for (int i = 0; i < t; i++)
14                 y[i] = declassify (x[i], {L});
15             return y;
16         } catch (ArrayIndexOutOfBoundsException ignored) {
17             return null;
18         }
19     }
20
21     public static BigIntPair[L] {L}
22     declassifyBigIntPair(BigIntPair[{P:;L}]{P:;L} x1)
23     where caller(P) {
24         BigIntPair[{P:;L}] x = declassify (x1, {L});
25         if (x == null) return null;
26         BigIntPair[L] y = new BigIntPair [L](
27             declassifyBigInt(x.getX()),
28             declassifyBigInt(x.getY()));
29         return y;
30     }
31     ...
32 }
```

Listing 4.13: Declassifier

The function `declassifyByteArray` accepts a high array as an argument and returns a low one. The method's begin label is restricted to {L} since we create a data structure (e.g. declare a new array) at low memory. Note that on line 6 we require the caller of this method have the authority of principal `P`.

The first declassification on line 8 downgrades the array reference itself. This declassification prevents the following operations to raise $\underline{pc}$. Now information about `x` is low, while the elements themselves are still high. Line 11 declares a new array into which declassified elements will be copied. The loop in lines 13– 14 declassifies the elements of array `x` and copies their values to array `y`, which we return in line 15.

The function `declassifyBigIntPair` declassifies an instance of the `BigIntPair` class that it accepts as an argument. Similarly to the previous function, on line 24 we declassify the argument. We return `null` if the argument is null. This also triggers Jif's `NullPointerException` analysis. A new container for declassified values is created on line 27. Note that the label of the parameter is low this time. The fields of type `BigIntPair` are then declassified in lines 27–28 and passed as the arguments to the constructor.

Listing 4.14 illustrates how this class can be used in a program.

```
byte[] chainingValue =
   Declassifier[P,L].declassifyByteArray(chainingValueP);

DABigInteger[L] pkey =
   Declassifier[P,L].declassifyBigInt(ph.getPublicParameter());

CardVector[{P:;L}]{P:;L} w_0_0 =
   Declassifier[P,L].upgradeCardVector(w_0);
```

Listing 4.14: Usage of Declassifier

A disadvantage of this approach is that it requires the constructors of relabeled classes to have all class fields as arguments. This is not always desirable since values of private variables, (e.g. internal state of an object), are not supposed to be instantiated in the constructors.

4.5.3 Ordering side-effects

Side effects are traced very rigidly across Jif programs in the compilation phase. This often leads to $\underline{pc}$ being too restrictive, and as a consequence it is necessary to declassify the current $\underline{pc}$ of the program. Sometimes this can be avoided, if a programmer can rearrange the code in the original Java program in such a way that all low operations precede the high operations.

A particular example of this is reading some input and later processing it at a higher level. Java programs may be written in such a way that low-level input reading and high-level processing instructions are interleaved. When lifting such code to Jif, declassifications must be introduced in between these inputs and the high processing. Often, it is possible to order these two parts and have input processed before.

As an illustration of this, consider function `processCardDraw0()` from the `Player` class.

```
public void processCardDraw0{L}(): {L}
throws MPException, SecurityException where caller (P) {
    // initialization
    DABigInteger[L] z_i = null;
    // public level...
    try {
        // input from DNC
    } catch (IndexOutOfBoundsException ex) {
        throw new MPException("processCardDrawo:␣IOB{}");
    }
    if (z_i == null) throw new
        MPException("processCardDraw0:␣z_i␣==␣null");
    ...
    // end of public level computation
    EncryptedCardVector[{P:;L}] ew = null;
    // entering classified level
    try {
        DABigInteger[{P:;L}] z_i_p
            = Declassifier[P,L].upgradeBigInt (z_i);
```

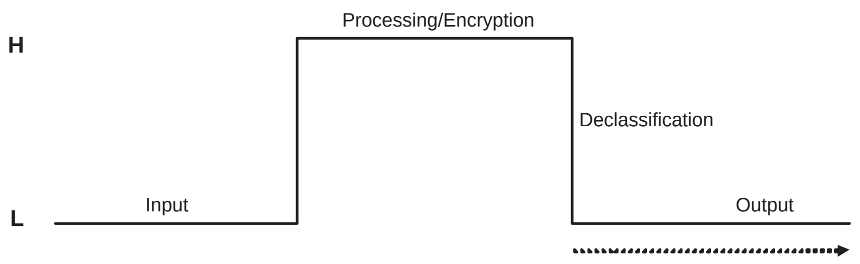

Figure 4.1: Program Flow

```
20            ...
21            ew = ... //
22        } catch (Exception ex) {
23            // catching high exceptions
24        }
25        try {
26            buildLink("en␣wi",
27             Declassifier[P,L].declassifyEncCardVector(ew));
28        } catch (IllegalArgumentException ex) {
29             throw new MPException("processCardDraw0␣failed");
30        }
31    }
```

Listing 4.15: Example of side-effect separation

In this example side effects are separated into three different parts. The first part, on lines 4–14, corresponds to *low input from DNC*. The second part, on lines 15–24 corresponds to processing input data on high level, such as encryption and applying secret permutation. In the third part, on lines 25–30, the result of the function is declassified. Note that we are obtaining a high copy of object `z_i` on lines 18–19: the variable `z_i_p` refers to a different object in the memory that has the same value as `z_i`.

Ordering side-effects in such a way that low-level input goes before high processing reduces the number of declassifications in high context. The presented code follows the pattern illustrated on Figure 4.1.

4.5.4 Requiring authority vs. granting it

Consider the two examples presented on Listing 4.16 and Listing 4.17. Both functions reveal the permutation matrix as a return value, declassifying it to a lower label.

```
public PermutationMatrix[L]{L} revealMatrix{L}()
where authority(P){
    return Declassifier[P,L].declassifyMatrix(matrixCopy);
}
```

Listing 4.16: Granting authority when revealing a secret

```
public PermutationMatrix[L]{L} revealMatrix{L}()
where caller(P){
    return Declassifier[P,L].declassifyMatrix(matrixCopy);
}
```

Listing 4.17: Checking for sufficient privileges

The code on Listing 4.16 grants the method the authority of the player. That is, it can be called from any context. This is a dangerous declaration as in this way secret information can be leaked very easily. The code on Listing 4.17, in contrast, does not grant anything, but rather requires the caller to have the necessary authority for declassification. Thus, a call of this method from a context where a process does not have authority of the player will be rejected statically by the compiler. Generally, as it has been stated before, granting authority to a piece of code should be done with care.

4.5.5 Avoiding restrictive end-labels

Listing 4.1 in Section 4.1.1 presents an example of the code that needs to release information about the termination of the method, but also not leak detailed data on the reason why the method failed to terminate normally.

Assuming that `foo()` is another such function, Listing 4.18 shows how one can use a boolean flag variable to track termination path of high methods.

```
 1  public void foo{L}():{L}
 2  throws MPException where caller(P) {
 3      boolean ok = false;
 4      try {
 5         ... // high code that can throw an exception
 6         ok = true;
 7      } catch (Exception ex) {
 8         ... // exception handling code here
 9      }
10      if (declassify (!ok, {L})) {
11          throw new MPException();
12      }
13  }
```

Listing 4.18: Declassification of success flag

In this example, the boolean flag `ok` is initialized on line 3. High code that can generate exceptions is enclosed by `try ... catch` statement, so that possible exceptions are caught and handled on lines 7–9.

Note that the assignment `ok=true` on line 6 can not generate exception and is the last one within the `try` block. Line 10 declassifies the value of this variable and depending on that value it may generate low exception that will propagate to the caller.

Ignoring exceptions

Ignoring exceptions is traditionally believed to be a sign of bad programming style. With Jif, sometimes it is safe to ignore some of the exceptions provided that there is sufficient evidence in the code that the exception will not be thrown. The most common exceptions to be ignored are

- `NullPointerException`. This exception may be ignored when there is no way that the object can be null. Partially, this is handled by Jif's `NullPointerException` analysis, mentioned in Section 3.3.6.
- `IndexOutOfBoundsException`. One can ignore this exception type if there is a check on the bounds of the array, that can generate this exception.

- `ClassCastException`. The operator `instanceof` can be used to check if the object is of the class that is used for casting.

4.5.6 Seal class

The seal class is used in the Jif implementation to ensure that the sensitive keys and the secret permutation matrix are not revealed before the game ends. Also, the methods that are used during the game use the “seal” to capture possible key misdeclassifications that could happen before the method is invoked. Unlike other patterns this one is a combination of conventional programming techniques and use of security features of Jif. Thus, the actual validation takes place at run-time. Also this pattern in no way prevents the programmer from using declassification without any prior checks. We believe that the idea of this pattern can be embedded as a possible extension to Jif, so that it would allow static analysis of declassification for this type of condition.

The structure of the `Seal` class is presented on Listing 4.19. The class is parameterized over two labels — owning principal `P` and the label `L`. This label stands for the lowest security level on which this seal is visible. The value of the seal is stored in the `boolean` variable `open`. The label of this variable means that it is only accessible to the owner `P`. Note the `caller` constraint in the class constructor on line 7. This constraint requires that the constructor should be called within the context that possesses the authority of the principal `P`.

Initially, the value of the variable `open` is `false`. It may only change to `true` in the method `unseal`. Similarly to the constructor call, this method has a `caller` constraint. This prevents calling this method from program contexts that do not have the authority of the seal’s owner.

The method `isOpen()` grants the authority of the owning principal to the process in order to declassify the current value to the visible level and return it. The method `assertIntegrity()` is similar to `isOpen()` and is a suggested way of checking if the seal has been opened or not.

```
1  /* Seal belongs to a principal P, and she allows it to be */
2  /* visible on the level L */
3  public class Seal[principal P, label L]  authority(P) {
4      /*actual value of the seal */
5      private boolean{P:;L} open;
6      /* require the principal to create this */
7      public Seal{P:;L}() where caller(P) {
8          this.open = false;
9      }
10     /* require the principal to unseal it */
11     public void unseal{P:;L}() where caller (P) {
12         this.open = true;
13     }
14     /* anyone on the level L can check it */
15     public boolean{this;L} isOpen():{L} where authority (P) {
16         return declassify (open, {this;L});
17     }
18     /* similar to previous */
19     public void assertIntegrity():{L} throws SecurityException {
20         if (this.isOpen())
21             throw new SecurityException();
22     }
23 }
```

Listing 4.19: Seal class

```
...
private Seal[Alice,{}] seal; // declaration
...
public void init() // initialization
where caller(Alice) {
  ...
  this.seal = new Seal[Alice, {}]();
  ...
}
...
public void work()
throws SecurityException, NullPointerException{
  // check the integrity in the beginning of the method
  this.seal.assertIntegrity();
  ...
}
...
public void revealSecret() where caller (Alice)
throws NullPointerException{
  this.seal.unseal();
  // declassification goes next
  ...
}
...
```

Listing 4.20: Usage of the Seal class

Chapter 5

Related work

A recent classification of declassification by Sabelfeld and Sands [27] considers current information release policies. It also stresses need for more expressive policies that would combine “what”, “who”, “where”, and “when” dimensions of information release.

A type system for web programming [20] developed by Li and Zdancewic implements a practical instance of *relaxed noninterference* [19]. However, to what extent this language addresses challenges for practical security has not so far been reported.

Giambiagi’s and Dam’s work on *admissible flows* [12] studies security conditions under which an implementation is guaranteed to reveal no more information than the specification of a protocol. The authors are considering *dependency specification* to show how much the underlying security protocol allows to specify. Next, they consider a simple imperative language, and proof that implementations in that language do not violate the requirements. This implementation language, however, is rather distant from a realistic language like Jif.

Recently, Chong and Myers have considered temporal release policies in the context of an ML-like language with the noninterference “until” policies. This policies guarantee that secrets are released after a certain statically-enforceable condition becomes true. This approach, however, abstracts away from how the release conditions are enforced.

Gutmann in [14] considers the errors that programmers tend to do when implementing crypto software. He also provides a series of design guidelines which may help to minimize damage due to misuse by inexperienced users. However, this work is orthogonal to ours.

Heldal *et al.* [16] considers how UML can be incorporated with Jif. This line of work is promising for specifying confidentiality labels and declassification points earlier in the design process.

Among so far reported implementations in Jif, the two are remarkable - battleship game protocol implemented with jif/split [33, 35] and an evaluation of an earlier version of Jif on a library of cryptographic primitives [31]. However, these implementations are relatively light (500 and 800 lines of code, respectively, vs. 4500 lines in this study).

Chapter 6

Conclusion

This report presents the largest code written in a security-typed language up to date. As a proof of concept a non-trivial cryptographic protocol has been implemented in a security-typed language.

6.1 Lessons learned

We have found security types useful for protecting confidentiality of sensitive variables. In this work security types have proven useful in preventing some explicit and non-trivial implicit flows in the baseline implementation, such as flows via exceptions or leaks via mutable data structures.

Although writing security-typed programs is quite laborious, the study shows that it is possible to write practical applications in such languages. When lifting Java programs to Jif the resulting security-typed code does not radically differ from the original. Moreover, we have developed patterns for security-typed programming to make programming in security-typed languages more convenient.

On the practical side, the lack of documentation and programming tools such as debuggers is an obstacle for widespread usage of such languages.

Jif helps identify points in the program where information release occurs. We refer to these points as declassification points. This study shows that declassification has a multi-faced nature. There are different reasons for information release in programs. Some declassifications are safer than the others and may be used in more contexts, and some may violate the security properties if applied inappropriately. When justifying every declassification point it is important to consider all dimensions of information release [27] — i.e. what is declassified, who performs this declassification, and where and when it exactly occurs.

Being the most ambitious security-typed language, Jif is not currently expressive enough to cover flexible security policies such as imposing temporary conditions on declassification. Furthermore, while manual inspections of declassification points is acceptable at some time, there is a need for expressing security policies in high-level languages.

6.2 Future work

- In this work, we are using a simple flat principal hierarchy with only two principals. Considering more sophisticated hierarchies with `actsFor` relations is one of the interesting directions that may be explored in future studies.

- Different kinds of declassifications that are identified in this work may be treated differently.
  - declassification+encryption: one can rely on the cryptographic properties of the encryption function and assume that it is safe to declassify the result of encryption.

    One way of doing it is by writing a cryptographic library that would give an authority of principals for declassification of encrypted data.
  - declassification+temporal policies: the seal approach presented in this work gives assurance that can be enforced at run-time only. We believe that introducing a seal-like data primitive into a language as a first-class citizen would lead to the possibility of enforcing stronger security guarantees statically.
- Improving Jif's shortcomings. One of these is the relabeling of mutable data structures. We believe this can be improved by introducing operations that would combine declassification and object cloning, so that a relabeled and separate copy of an object would be created. This would prevent laundering attacks as well as make programming in Jif easier since this is one of the most commonly used ways of declassification.

  One way of doing this is by defining Jif interface `Declassifiable` that would allow an operation `declassifyAndClone` to be performed on the class instances that implement the interface.

Bibliography

[1] Python programming language. Software release.

[2] Wikipedia, the free encyclopedia. Poker. `http://en.wikipedia.org/wiki/Poker`.

[3] Jif source code for the mental poker protocol, March 2005. `http://www.cs.chalmers.se/~aaskarov/jifpoker`.

[4] A. Barnett and N. P. Smart. Mental poker revisited. In *Proc. Cryptography and Coding IMA International Conference*, volume 2898 of *LNCS*, pages 370–383. Springer-Verlag, December 2003.

[5] J. Castellà-Roca and J. Domingo-Ferrer. On the security of an efficient ttp-free mental poker protocol. In *International Conference on Information Technology: Coding and Computing (ITCC'04), Volume 2, April 5-7, 2004, Las Vegas, Nevada, USA*, pages 781–. IEEE Computer Society Press, 2004.

[6] J. Castellà-Roca, J. Domingo-Ferrer, A. Riera, and J. Borrell. Practical mental poker without a TTP based on homomorphic encryption. In *Progress in Cryptology-Indocrypt*, volume 2904 of *LNCS*, pages 280–294. Springer-Verlag, December 2003.

[7] D. Coppersmith. Cheating at mental poker. In *CRYPTO '85: Advances in Cryptology*, pages 104–107. Springer-Verlag, 1986.

[8] C. Crépeau. A secure poker protocol that minimizes the effect of players coalitions. In *Advances in Cryptology: Crypto'85*, volume 218 of *LNCS*, pages 73–86. Springer-Verlag, 1986.

[9] C. Crépeau. A zero-knowledge poker protocol that achieves confidentiality of the players' strategy or how to achieve an electronic poker face. In *Advances in Cryptology: Crypto'86*, volume 263 of *LNCS*, pages 239–247. Springer-Verlag, 1987.

[10] J. Domingo-Ferrer. A new privacy homomorphism and applications. *Information Processing Letters*, 60(5):277–282, 1996.

[11] J. Edwards. Implementing electronic poker: A practical exercise in zero-knowledge interactive proofs. Master's thesis, Department of Computer Science, University of Kentucky, 1994.

[12] P. Giambiagi and M. Dam. On the secure implementation of security protocols. In *Proc. European Symp. on Programming*, volume 2618 of *LNCS*, pages 144–158. Springer-Verlag, April 2003.

[13] S. Goldwasser and S. Micali. Probabilistic encryption and how to play mental poker keeping secret all partial information. In *Proc. ACM Symp. Theory of Computing*, pages 365–377, 1982.

[14] P. Gutmann. Lessons learned in implementing and deploying crypto software. In *Proc. USENIX Security Symp.*, pages 315–325, August 2002.

[15] R. Hanna, A. Rideout, and D. Ziegler. Secure peer-to-peer texas hold'em. Course project, MIT. `http://web.mit.edu/ardonite/6.857/`, 2003.

[16] R. Heldal, S. Schlager, and J. Bende. Supporting confidentiality in UML: A profile for the Decentralized Label Model. In *Proc. International Workshop on Critical Systems Development with UML*, pages 56–70, 2004.

[17] K. Kurosawa, K. Katayama, and W. Ogata. Reshufflable and laziness tolerant mental card game protocol. *IEICE Transactions*, E80-A(1):72–78, 1997.

[18] K. Kurosawa, Y. Katayama, W. Ogata, and S. Tsujii. General public key residue cryptosystems and mental poker protocols. In *Advances in Cryptology: EuroCrypto'90*, volume 473 of *LNCS*, pages 374–388. Springer-Verlag, 1991.

[19] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 158–170, January 2005.

[20] P. Li and S. Zdancewic. Practical information-flow control in web-based information systems. In *Proc. IEEE Computer Security Foundations Workshop*, June 2005. To appear.

[21] R. Lipton. How to cheat at mental poker. In *Proc. AMS short course on Cryptology*, January 1981.

[22] R. Lipton. How to cheat at mental poker. In *Proceeding of the AMS short course on Cryptology*, January 1981.

[23] A. C. Myers. *Mostly-Static Decentralised Information Flow Control*. PhD thesis, MIT, 1999.

[24] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. ACM Symp. on Operating System Principles*, pages 129–142, October 1997.

[25] F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 319–330, January 2002.

[26] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.

[27] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proc. IEEE Computer Security Foundations Workshop*, June 2005. To appear.

[28] C. Schindelhauer. A toolbox for mental card games, 1998. `http://citeseer.ist.psu.edu/schindelhauer98toolbox.html`.

[29] A. Shamir, R. Rivest, and L. Adleman. Mental poker. *Mathematical Gardner*, pages 37–43, 1981.

[30] V. Simonet. The Flow Caml system. Software release. Located at `http://cristal.inria.fr/~simonet/soft/flowcaml/`, July 2003.

[31] S. Tse and G. Washburn. Cryptographic programming in Jif. Course project, 2003. `http://www.cis.upenn.edu/~stse/bank/main.pdf`.

[32] S. Zdancewic. Challenges for information-flow security. In *Proc. Programming Language Interference and Dependence (PLID)*, August 2004.

[33] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proc. ACM Symp. on Operating System Principles*, pages 1–14, October 2001.

[34] W. Zhao, V. Varadharajan, and Y. Mu. A secure mental poker protocol over the internet. In *CRPITS '21: Proceedings of the Australasian information security workshop conference on ACSW frontiers 2003*, pages 105–109. Australian Computer Society, Inc., 2003.

[35] L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proc. IEEE Symp. on Security and Privacy*, pages 236–250, May 2003.

Appendix A

Class and signature descriptions

A.1 Class descriptions for the Java implementation

- `BigIntPair` - pair of two big integers
- `BigIntVector` - vector of big integers
- `MPTable` - game coordinator
- `CRT` - Chinese remainder theorem
- `CardVector` - vector representation of a card, according to the game protocol
- `DABigInteger` - encapsulation of java.math.BigInteger
- `DAVector` - vector of DataFieldAttributes
- `DNCChain` - distributed notarization chain
- `DNCLink` - link, which is element of the chain
- `DataField` - field element of the chain
- `Digest` - byte digest of the permutation matrix
- `EncryptedCardVector` - representation of an encrypted card vector
- `PHCrypto` - implementation of the PH cryptosystem
- `PHEPermutationMatrix` - encrypted permutation matrix
- `PHIntVector` - vector of PHIntegers
- `PHInteger` - base class for operations over the encrypted integers - dual to BigInteger
- `PermutationMatrix` - permutation matrix
- `Player` - main class, which contains all the logic of the Player

A.2 Signatures written for the Jif implementation

- `java.lang.Character`
- `java.lang.Math`
- `java.math.BigInteger`
- `java.security.DigestException`
- `java.security.InvalidKeyException`
- `java.security.KeyPair`
- `java.security.KeyPairGenerator`
- `java.security.MessageDigest`
- `java.security.NoSuchAlgorithmException`
- `java.security.PrivateKey`
- `java.security.PublicKey`
- `java.security.SecureRandom`
- `java.security.Signature`
- `java.security.SignatureException`
- `java.util.Random`