Preventing Illicit Information Flow in Networked Computer Games using Security Annotations

Jonas Rabbe, s991125

Kgs. Lyngby, March 2005
M.Sc. Thesis
IMM-THESIS-2005-11

Computer Science and Engineering
Informatics and Mathematical Modeling
Technical University of Denmark

Abstract

In networked computer games using a client-server structure, bugs that result in information exposure can be used to cheat.

A programming language allowing the specification of security annotations can be designed specifically for the domain of a given game. Using the classic game *Battleships* as an example, a language *gWhile* has been designed which allows annotations following the Decentralized Label Model. The *gWhile* language includes communication and cryptography for secure communications, as well as other primitives specific to Battleships.

A type system has been designed to verify the information flow of programs in *gWhile* with respect to the Decentralized Label Model. A simple analysis has also been designed, the Type Matching Communications Analysis, which attempts to match communication statements in a program.

Keywords security, language design, information flow controls, the Decentralized Label Model, declassification, complete lattice, type system.

Resumé

Fejl som resulterer i informationseksponering kan i client-server baserede computerspil, som kommunikerer over et netværk, bruges til at snyde.

Et programmeringssprog, der gør det muligt at specificere sikkerhedsannotationer, kan designes specifikt for et givet spils domæne. Udfra det klassiske spil *Sænke Slagskibe*, er sproget *gWhile* blevet udviklet. *gWhile* tillader sikkerhedsannotationer som følger den decentraliserede label model. I sproget er der inkluderet kommunikationsprimitiver og kryptografi for at tilbyde sikker kommunikation, ligesom der er inkluderet andre primitiver specifikt til Sænke Slagskibe.

Et typesystem, som kan verificere informationsstrømmen i et program i *gWhile* udfra den decentraliserede label model, er blevet designet. Ligedes er en simple analyse blevet udviklet. Type Matching Communications Analysis forsøger at matche kommunikationsprimitiver i et program.

Nøgleord sikkerhed, sprog design, informations flow kontroller, den decentraliserede label model, deklassificering, fuldstændigt gitter, type system.

Preface

This report describes and documents the M.Sc. Thesis Project of Jonas Rabbe. The project corresponds to 30 ECTS points and has been carried out in the period from October 2004 to March 2005 at the Technical University of Denmark, Department of Informatics and Mathematical Modeling, Computer Science and Engineering division under the supervision of Professor Flemming Nielson.

I would like to thank Flemming Nielson for his great interest in my work, and his constructive criticism and inspiration during the project period. I would also like to thank my parents for proof-reading and commenting on the structure of the report. And thanks to the whole crew of room 008 for good company.

Special thanks to my wife, Susan Rabbe, for support, understanding, and patience throughout this project, as well as proof-reading.

Kgs. Lyngby, March 1, 2005

Jonas Rabbe, s991125

Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

More and more computer programs communicate over an insecure network such as the Internet. These programs rely on the safe and secure communication of their data to function properly. This is for example the case with computer games where knowledge of data from the other players can be used to cheat.

In the computer games industry cheating is serious business. If it comes out that cheating takes place in a game, players will quickly abandon it in favor of games where cheating is not prevalent yet. Cheating can be likened to doping in sports. As tournaments are held with large prizes this comparison becomes more and more apt. Cheating in a tournament could potentially cost a player the prize.

Most computer games today use a client-server infrastructure, where one server runs games for a number of players. Each player has some secret information. In the case of a first person shooter this could be his location, health, ammunition, or other vital signs. In a game such as battleships, the secret information is simply the location of your ships on the playing field.

In a computer game, one category of cheats concern the leakage of secret information, for example knowing the location of your enemy's ships. This kind of *information exposure* is sometimes due to bugs in a game [Pri00]. The leakage of information can be prevented by annotating programs with information about how data may flow.

One specific set of annotations is the Decentralized Label Model which has a well defined set of rules for how data flow is restricted. The Decentralized Label Model was introduced by Myers and Liskov [ML97], and has been applied to the programming language *JIF* [Mye99].

For a specific computer game a programming language can be designed with a domain specific to the game. This language can be designed with the annotations in place to allow the program to be annotated and the annotations to be verified. I have designed such a language for the game Battleships which allows annotations following the Decentralized Label Model. For this language I have designed a type system, both to check the plain types of a program and that the annotations hold.

To facilitate the client-server architecture, the language was designed with communication statements, along with extensions to the Decentralized Label Model to accommodate the new statements. Some aspects of these statements can be checked with the type-system. However, to verify that a program does not contain a communication statement which will never be matched a simple static analysis has been designed.

Both the type system and analysis have been implemented and used to verify that the problems which could occur in the Battleships program, as mentioned above, were discovered.

1.1 Problem Specification

This project was started with initial problem specification shown in Appendix A. However, it became clear the focus of the work in this thesis, while still centered on security annotations and their use in networked computer programs, would differ from the initial objectives.

The specifics of communication attacks in particular would serve as a distraction from the focus of the project, namely the use of security annotations to restrict illegal information flow. Concentrating on the design aspects of the language and verification, as mentioned below, counteracts the necessity to discuss communication attacks beyond the most general description. Suffice to say, an implementation of the source code translation would have to this into consideration to ensure the secrecy of the communications.

The design of the programming language, and especially the introduction of the communication primitives and their associated type rules, with respect to both the plain type system and security annotations, quickly proved more time consuming than anticipated. The source code translation envisaged in the original specification, an equally large undertaking, was phased out to make sure the language and type system design received the necessary attention.

The cryptographic extensions to the language were intended for the specific nature of networked computer games, but more generally link into networked programs, and are applicable to all programs employing the type of annotations used in this thesis.

In the same fashion as the problem specification, the title of the project evolved as work progressed. The initial title was:

Preventing Cheating in Computer Games through Realization of

Security Annotated Code

This title seemed in my mind quite focussed on the realization aspect of the initial problem specification. Similarly, information leaks in games can be used to cheat as described above, but the title made this work seem like a panacea in the field of computer games. The final title is more focussed on the containment of information leaks where not allowed, putting more focus on the networked aspects of the work in this project.

1.1.1 Final Specification

Computer programs of today rely on data being protected, essentially in a strongbox, to prevent illicit information flow. Once a principal can read data, however, there are no restrictions on his distribution of it. This is especially the case with data communicated over an open medium.

In computer games, cheats are available that rely on gaining access to information and distributing it to the cheater. For example exploiting a bug in the game to read the secret information of the other players.

A programming language can be designed, with the program or game in mind, in which the legal flow of information can be specified in the source code. These annotations can be verified, using type systems and program analysis, to inspect the legality of the information flow. The language must contain cryptography and communication statements appropriate for the program it is designed for, to ensure the secrecy of the communications, and prevent information leaks. Other primitives specific to the game are also introduced.

To motivate the development and provide a domain for the programming language I study an example program, namely the game Battleships. It is an example of a game played by two players over a network. Each player hides information from the other player–the location of his ships on the battle field. It is also a game in which a player will gain a large advantage by learning the hidden information for the other player.

An extension of the While language, with parallelism, communication and various security mechanisms like access control annotations and cryptographic primitives, is studied.

1.2 Battleships

The game Battleships is used as an example program in this thesis. It is a game known to most from its days as a board game.

Battleships is played by two players that start by assigning each of their ships to a position on the playing field, while keeping these positions secret from their opponent. Once the ships have been placed each player take turns trying to shoot down the ships of their opponent.

The playing field for Battleships consists of a grid where each grid intersection can be addressed by a unique coordinate, normally in the form of a letter and a number. The player whose turn it is, announces a coordinate he is targeting; his opponent then announces if it was a *splash*, there was no ship occupying that grid intersection, or a *kaboom*, there was a ship at the target coordinates.

There are a number of different variations on the rules at this point. If the shot resulted in a hit the player who shot either gets to go again, or the two players change turns. If the player misses he always looses the turn.

The player who hits all the ships of their opponent first wins.

In most versions of the game there are a number of different ships having different sizes, for example the carrier could be five sections long, while the destroyer could be three sections in size.

The main element of the game is secrecy. Each player tries to hide his ships from the opponent, but at the same time he must be truthful if his opponent hits one of his ships.

1.3 Structure of Report

Chapter 2 discusses a specific set of security annotations, namely the Decentralized Label Model. This set of annotations are used in the programming language which is designed, and its use and properties must be explained.

Chapter 3 presents the language which is designed. Describes both the syntax and specific thoughts, especially behind the cryptography and communication primitives.

Chapter 4 introduces the type systems used for verifying programs in the programming language. Also shows a simple analysis which attempts to match communications between processes.

Chapter 5 describes the specifics of the implementation. In the course of this project a parser, a type system, and an analysis have been implemented.

Chapter 6 summarizes the results of the work in this thesis, and shows a number of future directions the results could be used on.

CHAPTER 2

The Decentralized Label Model

This chapter describes the Decentralized Label Model which is one example of security annotations in practice. First the concept of labels, the central concept in the Decentralized Label Model, is described. This is followed by a description on the operators that work on labels in Section 2.1. Special properties of the operators, specifically the restriction operator as a partial order, and the lattice property of the set of labels is also described here. Section 2.2 speaks about declassification, the notion that values can flow against restrictions. Authority is discussed in Section 2.3. Finally, *JIF* is very briefly described, along with some thoughts on implementations of the Decentralized Label Model.

The Decentralized Label Model is a model for specifying end-to-end confidentiality policies. The authors of the Decentralized Label Model, Andrew C. Myers and Barbara Liskov, saw a number of inadequacies in the way access control currently works. The most common form of access control is based on access control lists which specify who can read and write data. This works well for preventing illicit access to data, but once data has been read, there is no limitations on how it can spread. There exist some models that allow end-to-end security policies, but according to Myers and Liskov [ML97, ML00] these cannot readily be put into practice.

The Decentralized Label Model is an attempt at making an access control model which contains end-to-end security policies while enabling its use in practical systems. Myers has developed a language [Mye99] based on Java and the Java Virtual Machine which enables the use of the Decentralized Label Model. The language is introduced as *JFlow*, but has later been renamed *JIF* which is an abbreviation of *Java Information Flow*.

The Decentralized Label Model is based on the labeling of variables to specify how their values may flow. The value of a variable may flow into another variable if the flow constitutes a restriction as described below. A label is a set of owners and for each owner a set of readers. The syntax for a label has been defined as

$$\{O_1 : \overrightarrow{R}_1;\ \ldots;\ O_n : \overrightarrow{R}_n\}$$

where $\overrightarrow{R}_i$ is a comma-separated list of readers. Both the O and R of owners and readers refer to principals of the program. In literature the principals are normally named with one letter. For two principals talking to each other the letters A and B are used. If a server is involved it usually has the name S.

For a label, *owner set* means the set of all owners of the label, while *reader set* refers to the set of all readers for a given owner. For example the reader set of a label for the owner A. Another concept is the *effective reader set* of a label which is the readers all the owners can agree upon to read the data. The effective reader set is the intersection of all the reader sets for the label.

An owner should only occur once in the owner set of a label. Likewise a reader should be unique to the reader set for a given owner. For a label $\{A : B;\ B : A;\ A : C\}$ this would mean that the second instance of A would be ignored and it would be equal to the label $\{A : B;\ B : A\}$. Normally labels will be short enough that it is not a difficult task to ensure that an owner does not occur more than once.

2.1 Operators in the Decentralized Label Model

Having introduced labels and the basic concepts related to labels, the mechanisms for working on them can be introduced. In the Decentralized Label Model variables are annotated to specify their labels, but to work with the labels specified some basic operators must be defined.

In the model there are two operators that work on labels: The restriction operator, $\sqsubseteq$, and the join operator, $\sqcup$. The restriction operator is a relation on labels which is true if the first label *is less restrictive* than the second. The notation "the second label *is a restriction* on the first" is also used. The join operator is used to combine two labels.

The definitions of the restriction and join operators depend on two functions: $owners(L)$ and $readers(L, O)$, these functions are analogous to the owner set for the label and the reader set for the owner O of the label. The function $owners(L)$ returns a set containing the principals that are owners of the label L. If

$$L = \{A : A, B, C;\ C : C, B\}$$

then

$$owners(L) = \{A, C\}$$

Similarly, $readers(L, O)$ returns a set containing the principals that are readers for a given owner, O, in L. Using the same L as above:

$$readers(L, A) = \{A, B, C\}$$

An owner is implicitly a reader in his own reader set. For a label $\{O : \overrightarrow{R}\}$ this means that the set for $readers(\{O : \overrightarrow{R}\}, O)$ is $\{\overrightarrow{R}\} \cup \{O\}$. In the example with L above this is not seen since A is already a member of his reader set, but if L is $\{A : B, C\}$ the result is still the same:

$$readers(L, A) = \{B, C\} \cup \{A\} = \{A, B, C\}$$

Another case worth examining is the result for $readers(L, B)$ where $B \notin owners(L)$. Since B is not an owner of the label, he does not impose any restriction on the propagation of data, and his reader set is simply the set of all principals.

The two operators are in [ML97] defined as:

Definition of $L_1 \sqsubseteq L_2$

$$\begin{array}{rcl} owners(L_1) & \subseteq & owners(L_2) \\ \forall O \in owners(L_1), readers(L_1, O) & \supseteq & readers(L_2, O) \end{array}$$

where L_1 is less restrictive than L_2, or L_2 is a restriction on L_1. Notice that the condition on readers is the inverse of the condition for owners. And

Definition of $L_1 \sqcup L_2$

$$\begin{array}{rcl} owners(L_1 \sqcup L_2) & = & owners(L_1) \cup owners(L_2) \\ readers(L_1 \sqcup L_2, O) & = & readers(L_1, O) \cap readers(L_2, O) \end{array}$$

The join results in the least restrictive label which is at least as restrictive as both L_1 and L_2. Since the reader set for an owner, which is not in the label, is simply all the principals, the join of two labels where one contains an owner not in the other would result in the owner and his readers from the first label being inserted into the resulting label. For example:

$$\{A : B;\ B : A, C\} \sqcup \{A : B\} = \{A : B;\ B : A, C\}$$

This also means that the join of a label L and the empty label is just L, since the empty label imposes no restrictions.

In addition to the two operators described above, the notion of two labels being equal is also needed. The equality relation on two labels is defined as:

Definition of $L_1 = L_2$

$$\begin{array}{rcl} owners(L_1) & = & owners(L_2) \\ \forall O \in owners(L_1), readers(L_1, O) & = & readers(L_2, O) \end{array}$$

2.1.1 Restriction as a Partial Order

It is interesting to note that the restriction operator, $\sqsubseteq$, on the set of all labels, S_L, is a partial order. This is proven by the following three properties given the labels L_1, L_2, and L_3:

1. $L_1 \sqsubseteq L_1$, reflexivity

2. $L_1 \sqsubseteq L_2 \wedge L_2 \sqsubseteq L_3 \Rightarrow L_1 \sqsubseteq L_3$, transitivity

3. $L_1 \sqsubseteq L_2 \wedge L_2 \sqsubseteq L_1 \Rightarrow L_1 = L_2$, anti-symmetry

Reflexivity holds immediately since a set is always a subset of itself:

$$owners(L_1) \subseteq owners(L_1)$$

Similarly, for the readers

$$\forall O \in owners(L_1), readers(L_1, O) \supseteq readers(L_1, O)$$

Transitivity follows in a similar fashion, from the transitive property of the $\subseteq$ operator on sets.

$$\begin{array}{rl} & owners(L_1) \subseteq owners(L_2) \wedge owners(L_2) \subseteq owners(L_3) \\ \Rightarrow & owners(L_1) \subseteq owners(L_3) \end{array}$$

In the same fashion as the reflexivity property, the second condition of the restriction operator can be shown:

$$\begin{array}{rl} & \forall O \in owners(L_1), readers(L_1, O) \supseteq readers(L_2, O) \\ \wedge & \forall O \in owners(L_2), readers(L_2, O) \supseteq readers(L_3, O) \\ \Rightarrow & \forall O \in owners(L_1), readers(L_1, O) \supseteq readers(L_3, O) \end{array}$$

Also from the transitivity property of the $\supseteq$ operator on sets, and the knowledge from condition one.

Anti-symmetry is again shown by looking at the conditions of the restriction operator. For the first condition this is

$$\begin{array}{rl} & owners(L_1) \subseteq owners(L_2) \wedge owners(L_2) \subseteq owners(L_1) \\ \Rightarrow & owners(L_1) = owners(L_2) \end{array}$$

which holds immediately. Similarly for the second condition:

$$\begin{array}{rl} & \forall O \in owners(L_1), readers(L_1, O) \supseteq readers(L_2, O) \\ \wedge & \forall O \in owners(L_2), readers(L_2, O) \supseteq readers(L_1, O) \\ \Rightarrow & \forall O \in owners(L_1), readers(L_1, O) = readers(L_2, O) \end{array}$$

2.1.2 A Lattice of Labels

Next it is shown that the partial order $\sqsubseteq$ admits binary least upper bounds and that they are given by the formula for $\sqcup$.

It is clear that $L_x \sqcup L_y$ finds an upper bound of the set $\{L_x, L_y\}$ since the $\sqcup$ operator adds owners and removes readers when there is a coincidence of owners. If you remember the condition for the restriction operator, which is our partial order, more owners and fewer readers for the owners in the less restrictive label constitutes a restriction. The join operator yields a label which is more restrictive than each of its operands, and is therefore an upper bound. Of interest here, however, is the least upper bound, or in label terminology: The least restrictive of all labels that are more restrictive than the operands. This is proven by contradiction. Imagine that the label found by $L_x \sqcup L_y$ is not the least upper bound of the set $\{L_x, L_y\}$. That would mean that there have to exist a label, L, which is an upper bound for $\{L_x, L_y\}$, more restrictive than both while being less restrictive than $L_x \sqcup L_y$.

L would have to contain all the owners from L_x as well as all the owners from L_y in order to fulfill condition one of the restriction operator. Similarly, for each owner in L there could be no more readers than for that owner in L_x or L_y.

Since $L_x \sqcup L_y$ is found by the union of the owners, and for each owner the intersection of the readers, the label L is equivalent to $L_x \sqcup L_y$ and there cannot exist a label which is more restrictive than L_x and L_y, but is less restrictive than $L_x \sqcup L_y$. The binary join operator clearly finds the least upper bound of its operands.

In [DP90] it is shown that there are a number properties associated with a binary join operator, most notably commutativity and transitivity. This means that $\bigsqcup S$ can be used to denote the join of all the elements of the finite and non-empty set S. If S is the set $\{L_x, L_y, L_z\}$ then

$$\bigsqcup S = L_x \sqcup L_y \sqcup L_z$$

It is clear from the proof of the binary join operator above and the associated traits that $\bigsqcup S$ finds the least upper bound of S.

However, the special case $\bigsqcup \emptyset$ remains. The join of the empty set simply yields the empty label. Remember that the empty label is the least restrictive of all labels since it allows information to flow anywhere. The empty label is the least or bottom element of the lattice, and is denoted by the $\perp$ symbol.

Hence the join of a finite (and possibly empty) set $S = \{L_1, \ldots L_n\}$ (for $n \geq 0$) is given by

$$\bigsqcup S = \perp \sqcup L_1 \sqcup \ldots \sqcup L_n$$

Given a program a finite set of principals and hence a finite set S_L of labels can be arranged for.

The join of the whole set, $\bigsqcup S_L$, therefore has been catered for and in fact yields the top element, the label which contains all principals of the program as owners and no specified readers for each owner. Data with this label is owned by everyone and can flow nowhere. It is the most restrictive label and has the symbol $\top$.

At this point it has been shown that the finite possibly empty set S_L is a partially ordered set with the restriction ordering, $\sqsubseteq$, and that every subset, $S_{\subseteq}$, of S_L has a least upper bound, $\bigsqcup S_{\subseteq}$.

Lemma A.2 of [NNH99, page 392] says:

> ***Lemma A.2*** For a partially ordered set $S_L = (S_L, \sqsubseteq)$ the claims
>
> 1. S_L is a complete lattice,
> 2. every subset of S_L has a least upper bound, and
> 3. every subset of S_L has a greatest lower bound
>
> are equivalent.

Therefore S_L is a complete lattice.

Some lattice properties of the set of labels have been described previously by Myers and Liskov, through reference to the work of Denning and the notion of a *security-class lattice* [Den76]. The concept of a complete lattice, however, has not been applied to the set of labels in available literature. One of the conditions of labels which should be noted again in this connection, is the prohibition of redundancy in labels. Redundancy is for example repetition of owners in a label or readers for a given owner, and would allow for labels that do not follow the anti-symmetry condition on the partial order. In the definition Myers and Liskov gives of labels there is no prohibition on redundancy [ML97].

2.2 Declassification

As mentioned above, the Decentralized Label Model allows data to flow as long as it becomes more restrictive. This works well for restricting the flow of data and preventing outsiders from learning your data. Sometimes, however, you may need to let an outsider learn something about your data. In following with the example program, Battleships, you have to let your opponent know if there is a ship at the coordinates he targeted. If you are only allowed restrictive flow you cannot do this unless he is allowed to learn all the values of your board, something which is quite undesirable. What is missing is declassification [ZM01]. Declassification allows an owner to modify the flow policy for his data. In the Decentralized Label Model this modification can either be the addition of one or more readers for the owner, or removal of the owner and his readers. Data is, in the *gWhile* language as defined in Section 3.1, declassified using the `declassify` construct. This function takes an expression and a label and attempts to put the label on the data of the expression. In Table 4.10 of Section 4.2 the type rule used for verifying the declassification expression is shown.

In its simplest form as shown in Table 2.1 a label, L_A, is constructed from the current principal, ρ in the rule. This label is used in together with the label, L, which the data should have after the declassify expression. The rule simply checks that the label of the expression, L_e, is less restrictive than L joined with L_A.

$$\frac{\rho;\ \lambda \vdash e : L_e \qquad L_A = \{\rho : \emptyset\} \qquad L_e \sqsubseteq L \sqcup L_A}{\rho;\ \lambda \vdash \texttt{declassify}(e, L) : L}$$

Table 2.1: The simple rule for verifying declassification

To follow the example above, imagine that a player, player 1, receives a pair of coordinates from an opponent, player 2, in a game of Battleships. The principal representing player 1 is denoted by A, while player 2 is represented by the principal B. The board of player 1 has the label $\{A : \emptyset\}$ and is therefore very restricted, only player 1 may read it. Player 1 has to send a reply to player 2 to show if there is a ship at the given coordinates or not. The way to do that is to declassify the value returned from accessing the board to allow the opponent to read it. This could either be done by relabeling it to $\{A : B\}$ or simply $\{\}$. Since there is only one opponent in a Battleships game there is no danger in allowing him to do what he wants with the data, and the board value is simply relabeled to the empty label using the following statement:

```
boardValue := declassify(board[x][y], {})
```

Since the current process is the process of player 1, ρ has the value A and the type rule for the declassification is verified as follows:

$$
\begin{array}{ll}
 & L_e \sqsubseteq L \sqcup L_A \\
\Leftrightarrow & \{A : \emptyset\} \sqsubseteq \{\} \sqcup \{A : \emptyset\} \\
\Leftrightarrow & \{A : \emptyset\} \sqsubseteq \{A : \emptyset\}
\end{array}
$$

In the *gWhile* language, however, it is not always as simple as using the current principal. Our data may reside on a server which under certain circumstances has the authority to act for us. If, for example, our game board is on a server which controls the logic of the game, the server must, upon receiving the target coordinates from our opponent, be allowed to declassify the board value for the coordinates. Since the current principal of the server is S it cannot declassify using the simple rule above, a more complex rule is called for, the rule that is shown in Table 2.2.

$$
\frac{\rho;\ \lambda \vdash e : L_e \qquad L_A = \{A : \emptyset | A \in \rho\} \qquad L_e \sqsubseteq L \sqcup L_A}{\rho;\ \lambda \vdash \texttt{declassify}(e, L) : L}
$$

Table 2.2: A more complex declassification type rule

This is the same rule which is shown in Table 4.10 of Section 4.2. Here ρ does not simply contain the current principal, but can be augmented using the special `actfor` construct described in Section 3.2.2 and is a set of principals the current process can act for. Since a process can always act for itself, ρ will for the server always contain S. If the server can currently act for us, ρ is extended to contain both S, and A, and the label L_A becomes $\{A : \emptyset;\ S : \emptyset\}$ since each principal in ρ is included as an owner with no readers in the label. The declassification example shown above then has the type verification:

$$
\begin{array}{ll}
 & L_e \sqsubseteq L \sqcup L_A \\
\Leftrightarrow & \{A : \emptyset\} \sqsubseteq \{\} \sqcup \{A : \emptyset;\ S : \emptyset\} \\
\Leftrightarrow & \{A : \emptyset\} \sqsubseteq \{A : \emptyset;\ S : \emptyset\}
\end{array}
$$

Outside the blocks where the server can act for us, ρ simply contains S and the declassification cannot take place

$$
\begin{array}{ll}
 & L_e \sqsubseteq L \sqcup L_A \\
\Leftrightarrow & \{A : \emptyset\} \sqsubseteq \{\} \sqcup \{S : \emptyset\} \\
\Leftrightarrow & \{A : \emptyset\} \sqsubseteq \{S : \emptyset\}
\end{array}
$$

since $\{A : \emptyset\} \sqsubseteq \{S : \emptyset\}$ does not hold.

2.3 The Principal Hierarchy

Above in Section 2.2 the hierarchy of principals was briefly touched upon through the notion of principals acting for each other.

In the *JIF* language there is a distinction between the *static principal hierarchy* and *run-time principals* [Mye99, ML00]. The static hierarchy is used in the static analysis of programs, while principals and labels can also be used as values at run-time and complicate matters further. The use of labels and principals at run-time mean that there are aspects of the principal hierarchy that may change during the execution of a program, including which principals a principal may act for. The only mention of how this dynamic hierarchy is maintained I could find is in [ML97, page 5] where it says: *"The right of one principal to act for another is recorded in a database."*

The *gWhile* language, outlined in Section 3.1, is only verified statically. The loss in versatility is outweighed by the added simplicity in the verification. Of further interest, however, is that the authority of a principal is not propagated through a convoluted hierarchy, but is entirely derived during the verification of a program.

One of the corner stones in the *gWhile* language is the notion of communication. As concluded in Section 3.2 the communication will be combined with cryptography to allow for secure communication. Both asymmetric and symmetric cryptography will be used in similar but different communication primitives. Asymmetric cryptography does not say anything about the person who encrypted and sent a message since the aptly named public key is publicly available. In symmetric cryptography, on the other hand, part of the security of the communication lies in the fact that only the two parties that communicate know the key. This has another profound impact; if you are told the key it says something about your authority. In the *gWhile* language I have let the ability to decrypt a message sent using a symmetric key be synonymous with the authority to act for the owners of the key. The owners of the key are the owners of the label for the package that is sent. In most cases this label will only have one owner and let everyone read it, there is, in other words, normally no limits to how an encrypted package may flow.

2.4 Implementations of the Decentralized Label Model

The only implementation at present of the Decentralized Label Model, with regards to a programming language, is *JIF* and the run-time system Myers has built for it.

The Decentralized Label Model is limited, as are all source code security verifications, by the quality of the runnable code, and the environment it is run in. Once the program has been compiled into machine code there is no room for annotations. There are two problems associated with this. One is the fact that a game can be altered by patches to perform illegal flow in an uncontrolled environment. The second is the situation where an end-user uses a program which breaks its promise to handle his information properly.

The second case has been the focus of most of the literature involving the Decentralized Label Model. Lately, however, the use of the Decentralized Label Model to prevent accidental information leaks has increased [ML00] compared to the earlier discussions [ML97, ML98].

In this case, the *JIF* run-time implementation allows insurance that the information flow policies of the program are enforced, making sure that the information of the user is not leaked without his knowledge. Myers has chosen to construct his own environment in which to run the compiled code. The *JIF* run-time system which is based on the Java Virtual Machine (JVM) and ensures that the information flow control of the Decentralized Label Model is observed.

Another way to ensure the enforcement of source code security verifications is through a reference monitor [SMH01]. In this case, however, the reference monitor must know the original source code and how the source code would make the program behave, or a verification result of the program saying what the program should do. If the program behaves outside the normal parameters as described by the source code or verification result, the monitor can restrict or terminate it.

The problem with both of these approaches is that it requires the user to obtain a module, the run-time system or reference monitor, which they have to trust.

Case number one is probably more applicable to the example of a computer game. Players trust the manufacturer enough to believe they will not intentionally allow people to cheat. The problem is more the imagined inaptitude of the developer. Most often, bugs which allow malicious players to hack, patch, or otherwise intercept data between the server and their own version of the game and cheat that way, are the problem. While a program running on an untrusted platform, which is what the game is on the malicious player's machine, cannot be trusted, the verification of the code that runs on the server prevents information leaks that are not consciously put in.

A further problem with regards to computer games is the issue of performance. In computer games there is an ever-present quest for getting the best graphics and fastest code on existing hardware. Using interpreted byte-code as the *JIF* run-time does, or a reference monitor which verifies each piece of object code before it is run, does not mesh well with this pursuit.

CHAPTER 3

The *gWhile* Language

To allow annotations of the example program, Battleships, as described in the previous chapter on the Decentralized Label Model, a language had to be designed. This chapter describes the design of the language and the thought behind the communication. The name of the language, *gWhile*, is short for *game While* from its use in a game, and its inheritance from the *While*Êlanguage.

First the syntax of the language is shown, along with a brief explanation of the elements. Since secure communication is a corner stone of a networked version of Battleships, Section 3.2 is devoted to the discussion of the cryptography and communication statements of the language.

3.1 Syntax of the gWhile *Language*

The syntax of the language designed for this thesis is shown in Figure 3.1, Figure 3.2, and Figure 3.3. This language is called the *gWhile* language, and is based on the *While* language introduced in [NN92]. There are, however, a number of changes in the *gWhile* language to make it more suitable for the example program, and to incorporate the annotations of the Decentralized Label Model.

The *JIF* implementation of the Decentralized Label Model uses the notion of labeling of variables. In *JIF* this is done using a syntax in which a variable is specifically initialized with a type, value, and label.

```
i:  int{A: A, B} := 0
```

The labeling of variables is inspired by the syntax of *JIF*, as is evident in the initializations of Figure 3.3. Types are in *gWhile* inferred from the initial values to simplify the initializations compared with *JIF*.

In the definition of the language some notations are used for variables, numbers, and principals which are defined as

$$\begin{array}{rcl} x, k, k^+, k^-, d & \in & \mathsf{Var} \\ n & \in & \mathsf{Num} \\ A & \in & \mathsf{Princ} \end{array}$$

where Var are variables, Num are numeric values, and Princ are principals. Similar notations for expression, statements, and so forth are defined for their relevant components below.

$$\begin{array}{lll} e & \in & \mathsf{Expr} \\ e & ::= & n \mid x \mid \texttt{this} \mid \text{'}A\text{'} \mid x[e_1][e_2] \mid \texttt{true} \mid \texttt{false} \\ & & \mid \texttt{random}(e) \mid \texttt{declassify}(e, L) \\ & & \mid e_1 + e_2 \mid e_1 = e_2 \mid e_1 < e_2 \mid \texttt{not}\ e \end{array}$$

Figure 3.1: Syntax of expression in the *gWhile* language

Expressions, as shown in Figure 3.1, have seen the addition of the `this` keyword which refers to the principal of the current process, a notation for principals, the table or two dimensional array used for the playing field of each player, and two functions: `random` and `declassify`. `random` is a weak random number generator used in the generation of the playing field for each player, it returns a number between 1 and the numerical value of the expression it is called with. `declassify` is used in the Decentralized Label Model as discussed in Section 2.2.

A more fundamental change is the combination of arithmetic and boolean expressions into the expression type. This was done to achieve greater freedom in the handling of expressions, for instance in the communication. A consequence of this is the necessity to instate a type system for the basic types of programs in *gWhile* shown in Section 4.1.

$$\begin{array}{lll} S & \in & \mathsf{Stmt} \\ S & ::= & x := e \mid x[e_1][e_2] := e_3 \mid \texttt{skip} \mid S_1;\ S_2 \\ & & \mid \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{endif} \mid \texttt{while}\ e\ \texttt{do}\ S\ \texttt{endwhile} \\ & & \mid \texttt{asend}(e_1, \ldots, e_n)\{k^+\} \mid \texttt{areceive}(e_1, \ldots, e_j; x_{j+1}, \ldots, x_n)\{k^-\} \\ & & \mid \texttt{ssend}(e_1, \ldots, e_k)\{k\} \\ & & \mid \texttt{sreceive}(e_1, \ldots, e_j;\ x_{j+1}, \ldots, x_k)\{k\} \\ & & \quad \texttt{andactfor}\ A\ \texttt{in}\ S\ \texttt{endactfor} \\ & & \mid \texttt{ssreceive}(e_1, \ldots, e_j;\ x_{j+1}, \ldots, x_k)\{k\} \\ & & \mid \texttt{instantiate}\ k \end{array}$$

Figure 3.2: Syntax of statements in the *gWhile* language

Additionally, the language also includes communication and cryptography primitives for the processes to communicate and do so securely. These are shown above–mainly in Figure 3.2–but are discussed in Section 3.2.

$$
\begin{array}{rcl}
L & \in & \textsf{Label} \\
L & ::= & A: \mid A: R_1, \ldots, R_n \mid A: \texttt{all} \mid \epsilon \mid L;\ L \\
\\
i & \in & \textsf{Init} \\
i & ::= & x\{L\} := n \mid x\{L\} := \text{`}A\text{'} \mid x\{L\} := \texttt{true} \mid x\{L\} := \texttt{false} \\
 & & \mid x[n_1][n_2]\{L\} \mid \texttt{key}\ k\{L\}\ \texttt{using}\ d \mid i, i \\
\\
AK & \in & \textsf{Asymmetric Keys} \\
AK & ::= & k(d)^+ \mid k(d)^- \mid AK, AK \\
\\
P & \in & \textsf{Proc} \\
P & ::= & A[AK] : (i)\{S\} \mid P\ P \\
\\
KD & \in & \textsf{Key Declarations} \\
KD & ::= & \texttt{declare}\ d\ \texttt{as}\ \{T_1\{L\}, \ldots, T_n\{L\}\}\{L\} \mid KD;\ KD \\
\\
Sys & \in & \textsf{System} \\
Sys & ::= & [KD]P
\end{array}
$$

Figure 3.3: Syntax of remaining elements of the *gWhile* language

The *gWhile* language has no dynamic memory allocation and all variables for each process must be declared in the initialization. The types of the variables are also determined from this initialization and follow the variables all through the program. Worth noticing is the initialization of table variables. A table can only contain integer values and is initialized with the size of the table. In the initialization all the cells of the table are automatically set to zero.

One of the initializations is the initialization of symmetric keys with the `key` k `using` d construct. The key in question is defined, but not instantiated, this means that using the key without instantiating it first would result in an error. Symmetric keys are initialized from a key declaration. A key declaration is a signature for the fields that can be sent using keys associated with it, for each field the type and label must be specified. Using, declaring, and instantiating symmetric keys is also discussed in Section 3.2.

Asymmetric keys are declared from the beginning of each process. The idea being that a client has the public keys of the server and uses these to communicate symmetric keys to the server which are then used to communicate the data. In the same fashion as the symmetric keys, an asymmetric key is defined with respect to a key declaration. This is done in the header of the process using the $k(d)^+$ syntax. The example would declare a public key with the identifier k using the key declaration d. The name of the key would be k^+.

3.2 Cryptography and Communication

In this section I will explain the cryptography and communication primitives in the *gWhile* language, as well as the reasoning behind them.

In *JIF*, communication is performed using channels. [ML00] describes channels as half-variables; they have associated labels in the same fashion as variables, but only allow *either* input or output. The rules for reading from and writing to channels are the same as for reading from or writing to variables. Channels differ further from the communication primitives normally found in network programming in that a channel is not only a way for two computers to communicate, it is also a way for a computer to communicate with its display, attached printer, even the keyboard.

While channels may work well for simple input and output of data, another layer of abstraction is desirable. It may be desired to send a message with several values over an open network.

Communication over an open network, however, has a further worry attached. Since the network is open, any data transmitted over it can be read by anyone. This opens the door for cryptography to ensure the secrecy of the communication. In communication secured by cryptography a number of conditions must be in place to ward off attacks. Cryptographic protocols are usually validated with respect to three properties:

Authenticity That each principal of the communication can be sure the other principal is who he claims to be.

Confidentiality That information communicated cannot be read by a third party.

Integrity That data cannot be altered in the process of the communication.

These properties are requirements for an implementation allowing the execution of programs in *gWhile*, but will not be considered in the analysis and verification of *gWhile* programs. It is assumed that an interpreter that implements the communication primitives would take into account the above properties to prevent attacks, allowing this discussion to focus on the primitives and their effects on the Decentralized Label Model.

The language contains both symmetric and asymmetric cryptography. The idea is that each copy of the game knows the public keys of the server, these keys are then used to communicate or negotiate one or more symmetric keys which can be used for the game specific communication.

Using symmetric cryptography alone is not feasible, since a unique set of keys for each player would be needed. While this is not a problem for the players, the number of keys that would have to be known beforehand by the server is immense. Relying solely on asymmetric cryptography, however, is not an option either. As described in Section 2.3 authority is connected with the symmetric cryptography. This authority is not readily replicable in the realm of asymmetric cryptography since the public keys of the server are available to all players of the game. The mixture of asymmetric and symmetric cryptography allow us to initiate the communication through the asymmetric cryptography, and use the symmetric cryptography to instill the notion of authority.

In the design of the language I have regarded two different approaches. The first was expression based encryption while the second was encryption built into the communication. Though the discussion in Section 3.2.1 is centered around asymmetric cryptography, much of the thought behind it is also applicable to symmetric cryptography as discussed in Section 3.2.2. The communication statements are assumed to be synchronous, this means that both the sender and receiver halts until the communication has taken place.

3.2.1 Primitives for Asymmetric Cryptography

In the discussion on asymmetric cryptography ak is used to denote an asymmetric key. This could be either a public or private key which has been shown as k^+ and k^- previously.

Expression Based Cryptography has two separate parts. The encryption takes a number of expressions and encrypts them into an encrypted package, as shown in Figure 3.4. The resulting package can be passed around in the same fashion as other expressions, provided it is typed correctly. An encrypted package can be decrypted using the `decrypt` statement. In the decryption pattern matching can take place. Pattern matching means that the encrypted package is decrypted, but its values are only assigned if its contents match the pattern of the `decrypt` statement. A pattern consists of a number of expressions to match, then a semicolon, followed by a number of variables to write the remaining values into. The first values are compared to the result of the expression, the pattern matches only if these values are equal.

In both approaches, as shown in Figure 3.4 and Figure 3.5, the key, ak, can be either a public or a private key. Using a public key ensures confidentiality, the package is encrypted; the use of a private key ensures authenticity, the package is signed.

Since an encrypted package can be sent around, any of the expressions encrypted in the expression based encryption can be another encrypted package. This means that parts, or all, of a package can be signed while still ensuring confidentiality–a package can both be signed and encrypted.

Cryptography in Communication considers cryptography built into the communication primitives of the language, these are shown in Figure 3.5.

$$
\begin{array}{lll}
e & \in & \mathsf{Expr} \\
e & ::= & \{e_1, \ldots, e_n\}\{ak\} \\
\\
S & \in & \mathsf{Stmt} \\
S & ::= & \texttt{send}(e_1, \ldots, e_n) \mid \texttt{receive}(n_1, \ldots, n_j; x_{j+1}, \ldots, x_n) \\
 & & \mid \texttt{decrypt}\ x\ \texttt{as}\ \{n_1, \ldots, n_j;\ x_{j+1}, \ldots, x_n\}\{ak\}
\end{array}
$$

Figure 3.4: Syntax of expression based encryption

This means that all communication is encrypted and allows for matching of encrypted values in the receive statement itself. A receive statement will only accept a package if it can be decrypted and matches the specified pattern.

$$
\begin{array}{lll}
S & \in & \mathsf{Stmt} \\
S & ::= & \texttt{asend}(e_1, \ldots, e_n)\{ak\} \mid \texttt{areceive}(e_1, \ldots, e_j;\ x_{j+1}, \ldots, x_n)\{ak\}
\end{array}
$$

Figure 3.5: Syntax of cryptography in communication

Both approaches have a number of advantages and disadvantages.

Expression Based Cryptography

- Advantages
  - No more of the message than what is strictly necessary has to be encrypted.
  - Since not everything that is sent has to be encrypted there is less of a burden on the processor.
  - Messages can both be signed and encrypted to ensure both authenticity and confidentiality.
- Disadvantages
  - Which parts of the message that must be encrypted and which ones do not must be taken into consideration.
  - Compared to Cryptography built into the Communication an extra variable has to be used to hold the received package, before it can be decrypted.
  - The type system for expressions would need an additional type for encrypted packages.

Cryptography in Communication

- Advantages

  - Everything sent is either encrypted or signed, so there is no need to think about which parts to encrypt.
  - All communication using the public key of the recipient is confidential.

- Disadvantages

  - Since everything that is sent is also encrypted it can be quite processing intensive.
  - A message cannot both be encrypted and signed.

Since asymmetric cryptography is only used in the communication of symmetric keys to the server, the simplicity of the second approach made it an easy choice. This is also evident from its inclusion in the *gWhile* syntax shown in Section 3.1. The syntax included in the *gWhile* language does not provide signed messages, but only allow a public key to be used in the `asend` statement, and a private key in the `areceive` statement.

Choosing this approach leads to an augmentation to the filesystem shown in Table 4.4.

3.2.2 Primitives for Symmetric Cryptography

In the following the name sk is used to denote a symmetric key.

$$
\begin{array}{lll}
S & \in & \mathsf{Stmt} \\
S & ::= & \texttt{ssend}(e_1, \ldots, e_k)\{sk\} \\
 & & \mid \texttt{sreceive}(e_1, \ldots, e_j;\ x_{j+1}, \ldots, x_k)\{sk\} \\
 & & \quad \texttt{andactfor}\ A\ \texttt{in}\ S\ \texttt{endactfor}
\end{array}
$$

Figure 3.6: Symmetric cryptography primitives

As mentioned above, the thought process for deciding the approach to asymmetric cryptography was largely relevant for symmetric cryptography as well. The simplicity of cryptography built into the communication primitives weighs more heavily than its drawbacks. An additional thought to take into consideration for symmetric cryptography, however, is the notion that the ability to decrypt something which has been encrypted with a symmetric key says something about your authority with respect to the owner of the key. Combining the symmetric receive primitive with the notion of authority as described in Section 2.3 yields the primitives shown in Figure 3.6.

S	$\in$	Stmt
S	$::=$	`ssreceive`$(e_1, \ldots, e_j;\ x_{j+1}, \ldots, x_k)\{sk\}$

Figure 3.7: Simple symmetric receive statement

Since the case where a principal receiving a package from someone else does not want to act for that principal exists, a simple receive statement shown in Figure 3.7 is also given.

i	$\in$	Init
i	$::=$	`key` sk `using` d

Figure 3.8: Symmetric key initialization

S	$\in$	Stmt
S	$::=$	`instantiate` sk

Figure 3.9: Symmetric key instantiation

KD	$\in$	Key Declarations
KD	$::=$	`declare` d `as` $\{T_1\{L\}, \ldots, T_n\{L\}\}\{L\} \mid KD;\ KD$

Figure 3.10: Symmetric key declaration

In these constructs the key, sk, is a symmetric key. A symmetric key is defined in the initialization of the process, shown in Figure 3.8, but is not instantiated until a session key is created using the `instantiate` statement shown in Figure 3.9. If `instantiate` is called on a symmetric key which has already been instantiated, a new instance of the key is generated. This is useful to prevent some replay attacks since each message sent would be sent with a new key. Of course there may be some added problems with the distribution of the new key and the processing power used for generating it.

The symmetric key, sk, is initialized from the declared key format, d. Key formats are declared in the Key Declarations header of the program, using the `declare` block as shown in Figure 3.10. A key declaration is declared with a number of fields to be sent. For each field the type of the field and its label is specified. The label for the encrypted package sent on the network must also be specified.

A symmetric key can from such a declaration be thought of as a transformation function from a set of fields each with a label to a single encrypted field with a specific label, and vice versa.

CHAPTER 4

Type System and Analysis

With some familiarity with both the Decentralized Label Model as a model for using security annotations of a language, and the syntax of a language, the *gWhile* language, which allows annotations to be specified at the source level, the next step is to look at verifications of the model and language.

As mentioned in Section 3.1 the combination of arithmetic and boolean expressions into the Expr type meant that programs had to be typed for basic type conformance. Furthermore, the verification of the security annotations can also be performed by a type system [VSI96, VS97, ML97]. To verify both the types in programs, and the security annotations, two type systems have been designed. Section 4.1 describes the so-called *plain type system* which checks the basic types of expressions. The unique features of the *gWhile* language with respect the Decentralized Label Model, however, are discussed in the *annotation type system* of Section 4.2.

A simple analysis of the communication is shown and described in Section 4.3.

4.1 Plain Type System

Two type systems have been designed. One for checking the basic types of a program, the second for checking the program with regards to the Decentralized Label Model. This sections discusses the plain type system used for checking the basic types. The type system for the security annotations of the Decentralized Label Model is described in Section 4.2.

In the type system there is the notion of types. The basic types are given by

$$
\begin{array}{rcl}
\tau & \in & \textsf{Basic Type} \\
\tau & ::= & \mathit{int} \mid \mathit{bool} \mid \mathit{principal} \mid \mathit{int} \times \mathit{int} \rightarrow \mathit{int} \\
& & \mid \tau_1 \times \ldots \times \tau_n \mid \tau_1 \times \ldots \times \tau_n \rightarrow \mathit{crypt} \\
& & \mid \tau_1 \times \ldots \times \tau_n \rightarrow \mathit{encrypt} \mid \tau_1 \times \ldots \times \tau_n \rightarrow \mathit{decrypt}
\end{array}
$$

while the large types are given by

$$
\begin{array}{rcl}
T & \in & \textsf{Large Type} \\
T & ::= & \mathit{stm} \mid \mathit{proc} \mid \mathit{sys}
\end{array}
$$

The basic types are used by expressions, variables, keys, and key declarations while the large types are used by statements, processes, and the system.

The basic types are mostly self-explanatory with the $\mathit{int} \times \mathit{int} \rightarrow \mathit{int}$ type denoting the type for an integer table. The table can be thought of as a function which accept two expressions that evaluate to integers and return an integer.

Also worth noting is the type for a key declaration, $\tau_1 \times \ldots \times \tau_n$, where each type, τ_i, matches the i^{th} type specified for the key format.

A symmetric key is associated with a key declaration, this means that it can only be used in sending and receiving messages that are in the format specified by the key declaration it is associated with. Symmetric keys have a format which is similar to the format for the integer table, except they use the key declaration as the fields and return a *crypt* field.

An asymmetric key is also associated with a key declaration in much the same way as a symmetric key. The format is the same too, but using either the *encrypt* type for public keys, or the *decrypt* type for private keys.

The large types are returned by the type rules for statements, processes, and the system to indicate that the rules type.

Common to all the type rules is the function

$$\gamma : \textsf{Var} \mapsto \tau$$

This function is the type environment, or variable map, for the type system. It maps each variable to its type, as defined by its initialization, as well as the key declarations and the asymmetric keys declared for the process. The domain of the type environment, $\mathrm{dom}(\gamma)$, is $\{x|\gamma \text{ contains } [x \mapsto \cdots]\}$. Furthermore, $\gamma(x) = \tau$ can be written if $x \in \mathrm{dom}(\gamma)$ and the occurrence of x in γ is $[x \mapsto \tau]$.

The three type environments from the key declarations, asymmetric keys, and initialization are combined using the combination, or $\vee$, operator. This operator creates a map which for each of the inputs to the previous maps still yield the values, for example $\gamma = [x \mapsto \tau'] \vee [y \mapsto \tau'']$ would yield $\gamma(x) = \tau'$ and $\gamma(y) = \tau''$. If two maps are combined which contain the same variable name, for example $\gamma = [x \mapsto v'] \vee [x \mapsto v'']$, then it results in an error. In the type system this is modeled by

$$\mathrm{dom}([x \mapsto v']) \cap \mathrm{dom}([x \mapsto v'']) = \emptyset$$

The intersection of the domains will only be non-empty and result in an error if a variable is defined multiple times.

Using the intersection of the domains as an implicit condition on the combination operator, $\gamma' \vee \gamma''$ is sufficient for the combination $\gamma' \vee \gamma''$ where $\mathrm{dom}(\gamma') \cap \mathrm{dom}(\gamma'') = \emptyset$.

4.1.1 Expressions

(int)	$\gamma \vdash n : int$	(var)	$\gamma \vdash x : \tau$ if $\gamma(x) = \tau$
$(this)$	$\gamma \vdash \texttt{this} : principal$	$(princ)$	$\gamma \vdash \text{‘}A\text{’} : principal$
$(true)$	$\gamma \vdash \texttt{true} : bool$	$(false)$	$\gamma \vdash \texttt{false} : bool$
$(table)$	$\dfrac{\gamma \vdash e_1 : \tau_1 \qquad \gamma \vdash e_2 : \tau_2 \qquad \gamma(x) = \tau_1 \times \tau_2 \rightarrow \tau}{\gamma \vdash x[e_1][e_2] : \tau}$		

Table 4.1: Plain type rules for the basis elements of expressions

$(rand)$	$\dfrac{\gamma \vdash e : int}{\gamma \vdash \texttt{random}(e) : int}$	$(decl)$	$\dfrac{\gamma \vdash e : \tau}{\gamma \vdash \texttt{declassify}(e, L) : \tau}$
(eq)	$\dfrac{\gamma \vdash e_1 : int \qquad \gamma \vdash e_2 : int}{\gamma \vdash e_1 = e_2 : bool}$		
(lt)	$\dfrac{\gamma \vdash e_1 : int \qquad \gamma \vdash e_2 : int}{\gamma \vdash e_1 < e_2 : bool}$		
(add)	$\dfrac{\gamma \vdash e_1 : int \qquad \gamma \vdash e_2 : int}{\gamma \vdash e_1 + e_2 : int}$		
(not)	$\dfrac{\gamma \vdash e : bool}{\gamma \vdash \texttt{not}\ e : bool}$		

Table 4.2: Plain type rules for the composite elements of expressions

The type rules for expressions shown in Table 4.1 and Table 4.2 are quite straightforward. The basis elements of the language simply return their specific type. In the case of variables, (*var*), this is done by fetching the type from γ. For (*table*) the type is found in γ and compared to the type for each of the indexing expressions to return the final type. Although the type for a table is $int \times int \rightarrow int$ there is no mention of *int* in the type rule. This is because the table is initialized into γ using $int \times int \rightarrow int$ which allows us to simply check that the types are the same.

The (*decl*) rule does nothing in the plain type system, as the declassify construct only has an effect in the annotation type system and the type of the expression is simply passed on.

The constant rules (*int*), (*this*), (*princ*), (*true*), and (*false*) all simply return their appropriate types. The binary operator rules (*eq*), (*lt*), and (*add*) check that the operands have the appropriate types and return the type of the operator. The monadic operator, (*not*), and remaining function, (*rand*), check the type of the operand and return the appropriate type.

4.1.2 Statements

For statements, only one simple type is used, the large type *stm*. The coherence in the statements must be checked, but only to ensure that they type. Table 4.3 shows the type rules for the simple statements which, with the exception of the tabular assignment, are also present in the normal *while* language. The assignment rules (*ass*) and (*tass*) simply type the left side of the statement and the right hand side, and check that the types match. The (*skip*) rule always type, while the sequence rule, (*seq*), types each of the statements. Finally, the (*if*) and (*while*) rules check that the expression is a boolean expression and type their substatements.

$$(\mathit{ass}) \qquad \frac{\gamma \vdash x : \tau \qquad \gamma \vdash e : \tau}{\gamma \vdash x := e : \mathit{stm}}$$

$$(\mathit{tass}) \qquad \frac{\gamma \vdash x[e_1][e_2] : \tau \qquad \gamma \vdash e_3 : \tau}{\gamma \vdash x[e_1][e_2] := e_3 : \mathit{stm}}$$

$$(\mathit{skip}) \quad \gamma \vdash \texttt{skip} : \mathit{stm} \qquad (\mathit{seq}) \quad \frac{\gamma \vdash S_1 : \mathit{stm} \qquad \gamma \vdash S_2 : \mathit{stm}}{\gamma \vdash S_1;\ S_2 : \mathit{stm}}$$

$$(\mathit{if}) \qquad \frac{\gamma \vdash e : \mathit{bool} \qquad \gamma \vdash S_1 : \mathit{stm} \qquad \gamma \vdash S_2 : \mathit{stm}}{\gamma \vdash \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{endif} : \mathit{stm}}$$

$$(\mathit{while}) \qquad \frac{\gamma \vdash e : \mathit{bool} \qquad \gamma \vdash S : \mathit{stm}}{\gamma \vdash \texttt{while}\ e\ \texttt{do}\ S\ \texttt{endwhile} : \mathit{stm}}$$

Table 4.3: Plain type rules for simple statements

Table 4.4, however, shows the type rules for the statements that concern cryptography and communication, and have a number of points worth investigating. The asymmetric cryptographic send and receive statements, shown in the rules (*asend*) and (*arec*), check that the key is an asymmetric key and compares types of the arguments with the types specified by the key.

Symmetric cryptography and communication is typed in much the same way as asymmetric. There is the symmetric send, typed by (*ssend*), the simple symmetric receive, shown in (*ssrec*), and the symmetric receive and actfor statements. The rules differ from the rules for asymmetric cryptography in the type of the key. Furthermore, the type rule for the receive and act for statement, (*srec*), in addition to the rules which are identical to (*ssrec*), verifies that A is a principal, and checks the statement, S.

Keys in the symmetric cryptographic communication must be instantiated, from a key declaration before they can be used, using the instantiate statement. This statement is checked by the (*inst*) rule which simply verifies that the specified key is a symmetric key.

4.1.3 Initialization, Keys, Processes, Key Declarations, and System

For each initialization, (*inum*) through (*itable*) as shown in Table 4.5, the type rules create a map, mapping the variable name to the appropriate type. The only deviation is the type rule for the key initialization, (*ikey*), which looks up the key declaration in γ and uses it in the map for the key. The maps from each type rule are combined in (*icomb*) using the combination operator.

In the same fashion as the type rule for the key initialization in Table 4.5, the type rules for the asymmetric keys, (*pubk*) and (*prik*) in Table 4.6, look up the key declaration for the key and use it in the map for the key. The maps for the asymmetric keys are also combined, in (*akcomb*), with the $\vee$ operator.

The processes, shown in Table 4.7, have the large type *proc* in the same fashion as statements. The type rule for a process, (*proc*), combines the map for the initializations with the map for the keys and the global key declarations to form a map for all the variables defined for the process, this map is used when checking the statement S.

The γ received by (*proc*) is used to check both the asymmetric keys and the initialization, variables that are defined multiple times are caught in the combination of the environments. In reality the asymmetric keys and the variables from the initialization cannot interfere, the asymmetric keys must be defined with either a $+$ or $-$ while normal variables cannot contain these characters.

$(asend)$	$$\frac{\gamma \vdash e_1 : \tau_1 \ \ldots \ \gamma \vdash e_n : \tau_n \qquad \gamma(k^+) = \tau_1 \times \ldots \times \tau_n \to \mathit{encrypt}}{\gamma \vdash \texttt{asend}(e_1, \ldots, e_n)\{k^+\} : \mathit{stm}}$$
$(arec)$	$$\frac{\gamma \vdash e_1 : \tau_1 \ \ldots \ \gamma \vdash e_j : \tau_j \qquad \gamma \vdash x_{j+1} : \tau_{j+1} \ \ldots \ \gamma \vdash x_n : \tau_n \qquad \gamma(k^-) = \tau_1 \times \ldots \times \tau_n \to \mathit{decrypt}}{\gamma \vdash \texttt{areceive}(e_1, \ldots, e_j;\ x_{j+1}, \ldots, x_n)\{k^-\} : \mathit{stm}}$$
$(ssend)$	$$\frac{\gamma \vdash e_1 : \tau_1 \ \ldots \ \gamma \vdash e_n : \tau_n \qquad \gamma(k) = \tau_1 \times \ldots \times \tau_n \to \mathit{crypt}}{\gamma \vdash \texttt{ssend}(e_1, \ldots, e_n)\{k\} : \mathit{stm}}$$
$(srec)$	$$\frac{\gamma \vdash e_1 : \tau_1 \ \ldots \ \gamma \vdash e_j : \tau_j \qquad \gamma \vdash x_{j+1} : \tau_{j+1} \ \ldots \ \gamma \vdash x_n : \tau_n \qquad \gamma(k) = \tau_1 \times \ldots \times \tau_n \to \mathit{crypt} \qquad \gamma \vdash A : \mathit{principal} \qquad \gamma \vdash S : \mathit{stm}}{\gamma \vdash \texttt{sreceive}(e_1, \ldots, e_j;\ x_{j+1}, \ldots, x_n)\{k\}\ \texttt{andactfor}\ A\ \texttt{in}\ S\ \texttt{endactfor} : \mathit{stm}}$$
$(ssrec)$	$$\frac{\gamma \vdash e_1 : \tau \ \ldots \ \gamma \vdash e_j : \tau \qquad \gamma \vdash x_{j+1} : \tau_{j+1} \ \ldots \ \gamma \vdash x_n : \tau_n \qquad \gamma(k) = \tau_1 \times \ldots \times \tau_n \to \mathit{crypt}}{\gamma \vdash \texttt{ssreceive}(e_1, \ldots, e_j;\ x_{j+1}, \ldots, x_n)\{k\} : \mathit{stm}}$$
$(inst)$	$$\frac{\gamma(k) = \tau_1 \times \ldots \times \tau_n \to \mathit{crypt}}{\gamma \vdash \texttt{instantiate}\ k : \mathit{stm}}$$

Table 4.4: Plain type rules for cryptographic and communicative statements

$(inum)$	$\gamma \vdash x\{L\} := n : [x \mapsto int]$
$(iprinc)$	$\gamma \vdash x\{L\} :=$ *'A'* $: [x \mapsto principal]$
$(itrue)$	$\gamma \vdash x\{L\} := \texttt{true} : [x \mapsto bool]$
$(ifalse)$	$\gamma \vdash x\{L\} := \texttt{false} : [x \mapsto bool]$
$(itable)$	$\gamma \vdash x[n_1][n_2]\{L\} : [x \mapsto int \times int \to int]$
$(ikey)$	$\dfrac{\gamma(d) = \tau_1 \times \ldots \times \tau_n}{\gamma \vdash \texttt{key}\ k\{L\}\ \texttt{using}\ d : [k \mapsto \tau_1 \times \ldots \times \tau_n \to crypt]}$
$(icomb)$	$\dfrac{\gamma \vdash i_1 : \gamma' \qquad \gamma \vdash i_2 : \gamma''}{\gamma \vdash i_1, i_2 : \gamma' \vee \gamma''}$

Table 4.5: Plain type rules for the initialization

$(pubk)$	$\dfrac{\gamma(d) = \tau_1 \times \ldots \times \tau_n}{\gamma \vdash k(d)^+ : [k^+ \mapsto \tau_1 \times \ldots \times \tau_n \to encrypt]}$
$(prik)$	$\dfrac{\gamma(d) = \tau_1 \times \ldots \times \tau_n}{\gamma \vdash k(d)^- : [k^- \mapsto \tau_1 \times \ldots \times \tau_n \to decrypt]}$
$(akcomb)$	$\dfrac{\gamma \vdash AK_1 : \gamma' \qquad \gamma \vdash AK_2 : \gamma''}{\gamma \vdash AK_1, AK_2 : \gamma' \vee \gamma''}$

Table 4.6: Plain type rules for the Asymmetric Keys

$(proc)$	$\dfrac{\gamma \vdash AK : \gamma' \qquad \gamma \vdash i : \gamma'' \qquad \gamma \vee \gamma' \vee \gamma'' \vdash S : stm}{\gamma \vdash A[AK] : (i)\{S\} : proc}$
$(plist)$	$\dfrac{\gamma \vdash P_1 : proc \qquad \gamma \vdash P_2 : proc}{\gamma \vdash P_1\ P_2 : proc}$

Table 4.7: Plain type rules for the Processes

(kd)	$\texttt{declare}\ d\ \texttt{as}\ \{\tau_1\{L\}, \ldots, \tau_n\{L\}\}\{L\} : [d \mapsto \tau_1 \times \ldots \times \tau_n]$
$(kdcomb)$	$\dfrac{KD_1 : \gamma' \qquad KD_2 : \gamma''}{KD_1;\ KD_2 : \gamma' \vee \gamma''}$

Table 4.8: Plain type rules for the Key Declarations

Table 4.8 show the rules for the key declarations. A key declaration, (*decl*), declares a key format which is used by either an asymmetric or a symmetric key as shown above, the names of each key declaration is mapped to the composite type shown in the table. For several key declarations, (*kdcomb*), the maps for each are combined.

$$(sys) \quad \frac{KD : \gamma \qquad \gamma \vdash P : proc}{[KD]P : sys}$$

Table 4.9: Plain type rules for the System

The type rule for the system, (*sys*) in Table 4.9, simply types the key declarations, KD, and passes the resulting map to the processes, P.

4.2 Type System for Security Annotations

The security annotations of the *gWhile* language is based on the Decentralized Label Model discussed in Chapter 2. The use of a type system to statically evaluate information flow in a program is not new. In [VSI96] Volpano, Smith, and Irvine show a type system for checking a simple programming language with respect to information flow. Myers does check the Decentralized Label Model using a type system, [ML97], but uses the block label to prevent implicit flow as described in Section 4.2.1. This functionality was in [VS97] achieved using subtypes which is considerably more unwieldy.

The type system described in this section is used to check the Decentralized Label Model as implemented in the *gWhile* language. It is structured much in the same fashion as the plain type system.

Just as for the plain type system, the annotation type system has a type environment. However, the type environment for the annotation type system, λ, is the function

$$\lambda : \mathsf{Var} \mapsto \mathsf{Label}$$

The definitions of $\mathrm{dom}(\lambda)$ and $\lambda(x) = L$ are the same as $\mathrm{dom}(\gamma)$ and $\gamma(x) = \tau$ in the plain type system, with the difference that $\lambda(x)$ returns a label while $\gamma(x)$ returns a basic type. In addition to the type environment the annotation type system carries two variables. The first is the set, ρ, which is used in declassification as shown in Table 4.10. ρ contains the current principal as well as any principals the current principal can act for at present, and is also referred to as *the set of current principals*. The second is the block label B which is described further in Section 4.2.1 below.

The combination, or $\vee$, operator uses the same definition as described in Section 4.1, including the domain intersection condition.

The rules for statements, processes, and the system use the large types defined for the plain system to indicate that they type, in the same fashion as those rules in the plain type system.

4.2.1 The Block Label

Given a simple program segment with the variables `h` which is a high-security variable, and `l` which is low-security:

```
if h = 0 then
    l := 0
else
    l := 1
```

Depending on the value of `l` something is known about `h` after the if statement has been executed. This is referred to as *implicit information flow*. To solve this problem an assignment inside an if statement can only take place if the variable which is being assigned to is more restrictive than both the value which is being assigned, *and* the expression which is branched on. For the statement `l := 0` this would be:

$$L_0 \sqsubseteq L_l \wedge L_{h=0} \sqsubseteq L_l$$

which amounts to

$$L_0 \sqcup L_{h=0} \sqsubseteq L_l$$

where $L_{h=0}$ is the label for the branching expression.

If an assignment is nested inside several if statements, while loops, or similar then $L_{h=0}$ would of course have to be augmented with the labels for those expressions as well.

The idea with the block label, which is the role $L_{h=0}$ played in the example above, is to initialize it with the $\perp$ element. Each time a branch or loop statement is encountered the block label is augmented, with the label for the expression, using the $\sqcup$ operator. The augmented block label is then used to check the blocks of the branch or loop. Once the blocks have been checked the original block label is restored.

The symmetric receive and act for statement both augments the block label and the set of current principals, ρ, before checking the statement S. Both these variables are restored when the statement of the act for statement has been checked.

The type rules for the annotation type system have names in the same fashion as the plain type rules. They are, however, subscripted with L to signify the rules deal with labels, for examples int_L.

4.2.2 Expressions

All expressions carry the label map, λ, and the set of current principals, ρ. The label map is used for finding labels for simple variables or tables. The current principals are used in the declassify expression as described in Section 2.2.

(int_L)	$\rho;\ \lambda \vdash n : \bot$	(var_L)	$\rho;\ \lambda \vdash x : L$ if $\lambda(x) = L$
$(this_L)$	$\rho;\ \lambda \vdash \texttt{this} : \bot$	$(princ_L)$	$\rho;\ \lambda \vdash \texttt{'A'} : \bot$
$(true_L)$	$\rho;\ \lambda \vdash \texttt{true} : \bot$	$(false_L)$	$\rho;\ \lambda \vdash \texttt{false} : \bot$

$$(table_L)\quad \frac{\rho;\ \lambda \vdash e_1 : L_1 \qquad \rho;\ \lambda \vdash e_2 : L_2 \qquad \lambda(x) = L_x}{\rho;\ \lambda \vdash x[e_1][e_2] : L_x \sqcup L_1 \sqcup L_2}$$

$$(bop_L)\quad \frac{\rho;\ \lambda \vdash e_1 : L_1 \qquad \rho;\ \lambda \vdash e_2 : L_2}{\rho;\ \lambda \vdash e_1\ bop\ e_2 : L_1 \sqcup L_2}$$

$$(mop_L)\quad \frac{\rho;\ \lambda \vdash e : L}{\rho;\ \lambda \vdash mop\ e : L}$$

$$(decl_L)\quad \frac{\rho;\ \lambda \vdash e : L_e \qquad L_A = \{A : \emptyset | A \in \rho\} \qquad L_e \sqsubseteq L \sqcup L_A}{\rho;\ \lambda \vdash \texttt{declassify}(e, L) : L}$$

Table 4.10: Annotation type rules for expressions

Apart from the type rule $(decl_L)$ the type rules for expressions are pretty straightforward. The constant rules, (int_L), $(this_L)$, $(princ_L)$, $(true_L)$, and $(false_L)$, all return the empty label shown as the $\bot$ element. For variables, (var_L), the corresponding label is fetched from λ.

The rule for tables, $(table_L)$, is somewhat similar to variables, in that the label for the table is fetched from λ here, too. The label for the value returned from the table, however, depends not only on the label for the table, but also on the labels for the two indexing expressions. The label returned from $(table_L)$ is therefore the join of the three labels.

Binary operators typed by (bop_L), the $+$, $=$, and $<$ operators in the *gWhile* language, return the join of the label of each expression. Monadic operators, (mop_L), for example the not operator, simply return the label of the expression. While the random function is not as such an operator, its label is handled by the type rule for monadic operators.

The $(decl_L)$ rule for the declassification expression is, in contrast to the other type rules, a bit convoluted. In the Decentralized Label Model assignment of values to variables can only take place if the assignment constitutes a restriction. This means that eventually information can no longer flow.

Following the restriction operator it is not possible to let another principal read data unless he is already in the effective reader set of the label. As described in Section 2.2, the declassify function allows the removal of an owner from a label, or the addition of reader to his reader set, provided the owner is in the set of current principals, ρ.

4.2.3 Statements

In addition to λ and ρ the annotation type rules for statements carry the block label, B.

(ass_L)	$\dfrac{\rho;\ \lambda \vdash e : L_e \qquad \rho;\ \lambda \vdash x : L_x \qquad B \sqcup L_e \sqsubseteq L_x}{B;\ \rho;\ \lambda \vdash x := e : stm}$
$(tass_L)$	$\dfrac{\begin{array}{c}\rho;\ \lambda \vdash e : L_e \qquad \rho;\ \lambda \vdash e_1 : L_1 \qquad \rho;\ \lambda \vdash e : L_1 \\ \rho;\ \lambda \vdash x : L_x \qquad B \sqcup L_e \sqcup L_1 \sqcup L_2 \sqsubseteq L_x\end{array}}{B;\ \rho;\ \lambda \vdash x[e1][e2] := e : stm}$
$(skip_L)$	$B;\ \rho;\ \lambda \vdash \texttt{skip} : stm$
(seq_L)	$\dfrac{B;\ \rho;\ \lambda \vdash S_1 : stm \qquad B;\ \rho;\ \lambda \vdash S_2 : stm}{B;\ \rho;\ \lambda \vdash S_1;\ S_2 : stm}$
(if_L)	$\dfrac{\begin{array}{c}\rho;\ \lambda \vdash e : L_e \\ B \sqcup L_e;\ \rho;\ \lambda \vdash S_1 : stm \qquad B \sqcup L_e;\ \rho;\ \lambda \vdash S_2 : stm\end{array}}{B;\ \rho;\ \lambda \vdash \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{endif} : stm}$
$(while_L)$	$\dfrac{\rho;\ \lambda \vdash e : L_e \qquad B \sqcup L_e;\ \rho;\ \lambda \vdash S : stm}{B;\ \rho;\ \lambda \vdash \texttt{while}\ e\ \texttt{do}\ S\ \texttt{endwhile} : stm}$

Table 4.11: Annotation type rules for basic statements

Some of the statements, the aforementioned "basic" statements, shown in Table 4.11 simply follow the Decentralized Label Model. The assignment, (ass_L), allows assignments if they constitute a restriction on the label for the expression and the block label as described in Section 4.2.1 above.

The $(skip_L)$ always types, while the (seq_L) rule types each of the statements and then returns the large type stm.

The rules for branch statements, (if_L) and $(while_L)$, augment the block label, B, by joining it with the label of the expression to prevent implicit information flow. The augmented block label is then used to check the block of the if or while statement.

Worth noticing is the rule for the table assignment, $(tass_L)$, compared to the expression type rule for the table, $(table_L)$ in Figure 4.10. Imagine a table, `t`, which everyone can read where the contents are known, a variable `l` with a low security level, and a variable `h` with a high level. Given the assignment

```
l := t[1][h]
```

It is clear that something can be learned about `h` from the value of `l`, an example of implicit flow as discussed before. The label for the expression `t[1][h]` is therefore dependent on the labels for both `t`, `1`, and `h` as shown in the ($table_L$) rule of Table 4.10. Unfortunately, the inverse problem still exists, illustrated by the code

```
t[1][h] := l
```

If the rule described above is simply followed then `t[1][h]` is more restrictive than `l` and the assignment is valid. A search in the table `t` for the value of `l` afterwards, however, yields information about the value of `h`. This problem is solved by letting the labels for the indexing expressions add to the label of the assigned expression, `l`. In the current example this means that the label for the table must be more restrictive than the label for `h` joined with the label for `l`. This rule is shown in ($tass_L$) in Table 4.11 with the additional use of the block label to prevent implicit flow.

As described in Section 3.2 the *gWhile* language contains both asymmetric and symmetric cryptographic primitives.

$$(asend_L) \qquad \frac{\begin{array}{c}\rho;\ \lambda \vdash e_1 : L_1\ \ldots\ \rho;\ \lambda \vdash e_n : L_n \\ \lambda(k^+) = L_{k1} \times \ldots \times L_{kn} \mapsto L_k \\ (\forall i \in [1,n])(B \sqcup L_i \sqsubseteq L_{ki})\end{array}}{B;\ \rho;\ \lambda \vdash \texttt{asend}(e_1, \ldots, e_n)\{k^+\} : stm}$$

$$(arec_L) \qquad \frac{\begin{array}{c}\rho;\ \lambda \vdash e_1 : L_1\ \ldots\ \rho;\ \lambda \vdash e_j : L_j \\ \rho;\ \lambda \vdash x_{j+1} : L_{j+1}\ \ldots\ \rho;\ \lambda \vdash x_n : L_n \\ \lambda(k^-) = L_{k1} \times \ldots \times L_{kn} \mapsto L_k \\ B' = B \sqcup L_1 \sqcup L_{k1} \sqcup \ldots \sqcup L_j \sqcup L_{kj} \\ (\forall i \in [j+1,n])(B' \sqcup L_{ki} \sqsubseteq L_i)\end{array}}{B;\ \rho;\ \lambda \vdash \texttt{areceive}(e_1, \ldots, e_j;\ x_{j+1}, \ldots, x_n)\{k^-\} : stm}$$

Table 4.12: Annotation type rules for asymmetric cryptographic and communication statements

The type rules for asymmetric and symmetric cryptographic communication statements, as shown in Table 4.12 and Table 4.13 respectively, are quite similar. The type rule for `asend`, ($asend_L$), is analogous to the one

$$(ssend_L)\quad \frac{\begin{array}{c}\rho;\ \lambda \vdash e_1 : L_1\ \ldots\ \rho;\ \lambda \vdash e_n : L_n \\ \lambda(k) = L_{k1} \times \ldots \times L_{kn} \mapsto L_k \\ (\forall i \in [1,n])(B \sqcup L_i \sqsubseteq L_{ki})\end{array}}{B;\ \rho;\ \lambda \vdash \texttt{ssend}(e_1,\ldots,e_n)\{k\} : stm}$$

$$(srec_L)\quad \frac{\begin{array}{c}\rho;\ \lambda \vdash e_1 : L_1\ \ldots\ \rho;\ \lambda \vdash e_j : L_j \\ \rho;\ \lambda \vdash x_{j+1} : L_{j+1}\ \ldots\ \rho;\ \lambda \vdash x_n : L_n \\ \lambda(k) = L_{k1} \times \ldots \times L_{kn} \mapsto L_k \\ B' = B \sqcup L_1 \sqcup L_{k1} \sqcup \ldots \sqcup L_j \sqcup L_{kj} \\ (\forall i \in [j+1,n])(B' \sqcup L_{ki} \sqsubseteq L_i) \\ A \in owners(L_k) \qquad B'; \rho \cup \{A\}; \lambda \vdash S : stm\end{array}}{\begin{array}{c}B;\ \rho;\ \lambda \vdash \texttt{sreceive}(e_1,\ldots,e_j;\ x_{j+1},\ldots,x_n)\{k\} \\ \texttt{andactfor}\ A\ \texttt{in}\ S\ \texttt{endactfor} : stm\end{array}}$$

$$(ssrec_L)\quad \frac{\begin{array}{c}\rho;\ \lambda \vdash e_1 : L_1\ \ldots\ \rho;\ \lambda \vdash e_j : L_j \\ \rho;\ \lambda \vdash x_{j+1} : L_{j+1}\ \ldots\ \rho;\ \lambda \vdash x_n : L_n \\ \lambda(k) = L_{k1} \times \ldots \times L_{kn} \mapsto L_k \\ B' = B \sqcup L_1 \sqcup L_{k1} \sqcup \ldots \sqcup L_j \sqcup L_{kj} \\ (\forall i \in [j+1,n])(B' \sqcup L_{ki} \sqsubseteq L_i)\end{array}}{B;\ \rho;\ \lambda \vdash \texttt{ssreceive}(e_1,\ldots,e_j;\ x_{j+1},\ldots,x_n)\{k\} : stm}$$

$$(inst_L)\quad B;\ \rho;\ \lambda \vdash \texttt{instantiate}\ k : stm$$

Table 4.13: Annotation type rules for symmetric cryptographic and communication statements

for `ssend`, $(ssend_L)$, and likewise for `areceive` and `ssreceive`, with the rules $(arec_L)$ and $(ssrec_L)$. Most of the type rule for the receive and act for statement, $(srec_L)$, is the same as $(ssrec_L)$, the difference being in the augmentation of ρ.

When a number of values are sent it is useful to think of a series of assignments taking place, from the values to the fields of the send statement. The typical rule for an assignment is $B \sqcup L_e \sqsubseteq L_x$ where L_e is the label for the expression and L_x is the label for the assignee. For each field in the key format, $i \in [1, n]$, a similar rule, $B \sqcup L_i \sqsubseteq L_{ki}$, is given. In the rule each field has an associated label, L_{ki}, and each value has a label, L_i.

For the receive statements pattern matching is used which makes the type rules a bit different. The first j expressions are used for matching the pattern, while the variables specified for the remaining $n - j$ fields are assigned to. The assignment to these variables use a rule much the same as for send statements and normal assignments, $B' \sqcup L_{ki} \sqsubseteq L_i$. Worth noticing, however, is that the assignment is to the variables from the fields, hence the reversal of L_{ki} and L_i. Furthermore, the block label, B', is an augmentation of the normal block label B. The receive statement is only executed if the first j fields match, these fields are in a way conditions on the statement. The block label is therefore enlarged the same way it would have been for a series of nested if statements, each containing an equality condition corresponding to the fields and values. The augmented block label, B', is also used in the verification of the statement, S, in the receive and act for statement of $(srec_L)$. Additionally, the receive and act for statement adds the principal A to ρ before checking S. Both the block label and ρ are restored before the statement following the receive statements are checked.

4.2.4 Initialization, Keys, Processes, Key Declarations, and System

Table 4.14 shows the type rules for the initialization part of a process. It is very similar to the type rules for the plain type system shown in Section 4.1, except that variable names map to labels instead of types. Worth noticing is that the map for a symmetric key, as shown in $(ikey_L)$, is the same as the map for its key declaration. The same is the case for the asymmetric keys shown in Table 4.15.

A process, typed by $(proc_L)$, receives a type environment from the key declarations through the system, types the asymmetric keys and the initialization with respect to λ, and use the combination of the three environments together with the $\perp$ element for the block label and ρ as a singleton set of the current principal, for checking the statement. The list of processes, $(plist_L)$, are simply checked one at the time with the environment λ as generated from the key declarations.

The key declarations, Table 4.17, define the formats that can be used for

$(inum_L)$	$\lambda \vdash x\{L\} := n : [x \mapsto L]$
$(iprinc_L)$	$\lambda \vdash x\{L\} := \text{‘}A\text{’} : [x \mapsto L]$
$(itrue_L)$	$\lambda \vdash x\{L\} := \texttt{true} : [x \mapsto L]$
$(ifalse_L)$	$\lambda \vdash x\{L\} := \texttt{false} : [x \mapsto L]$
$(itable_L)$	$\lambda \vdash x[n_1][n_2]\{L\} : [x \mapsto L]$
$(ikey_L)$	$\dfrac{\lambda(d) = L_1 \times \ldots \times L_n \mapsto L}{\lambda \vdash \texttt{key}\ k\ \texttt{using}\ d : [k \mapsto L_1 \times \ldots \times L_n \to L]}$
$(icomb_L)$	$\dfrac{\lambda \vdash i_1 : \lambda' \qquad \lambda \vdash i_2 : \lambda''}{\lambda \vdash i_1, i_2 : \lambda' \vee \lambda''}$

Table 4.14: Annotation type rules for the initialization

$(pubk_L)$	$\dfrac{\lambda(d) = L_1 \times \ldots \times L_n \mapsto L}{\lambda \vdash k(d)^+ : [k^+ \mapsto L_1 \times \ldots \times L_n \to L]}$
$(prik_L)$	$\dfrac{\lambda(d) = L_1 \times \ldots \times L_n \mapsto L}{\lambda \vdash k(d)^- : [k^- \mapsto L_1 \times \ldots \times L_n \to L]}$
$(akcomb_L)$	$\dfrac{\lambda \vdash AK_1 : \lambda' \qquad \lambda \vdash AK_2 : \lambda''}{\lambda \vdash AK_1, AK_2 : \lambda' \vee \lambda''}$

Table 4.15: Annotation type rules for the Asymmetric Keys

$(proc_L)$	$\dfrac{\begin{array}{c}\lambda \vdash i : \lambda' \qquad \lambda \vdash AK : \lambda'' \\ \perp;\ \{A\};\ \lambda \vee \lambda' \vee \lambda'' \vdash S : stm\end{array}}{\lambda \vdash A[AK] : (i)\{S\} : proc}$
$(plist_L)$	$\dfrac{\lambda \vdash P_1 : proc \qquad \lambda \vdash P_2 : proc}{\lambda \vdash P_1\ P_2 : proc}$

Table 4.16: Annotation type rule for the processes

(kd_L)	`declare` d `as` $\{T_1\{L_1\},\ldots,T_n\{L_n\}\}\{L\}$: $[d \mapsto L_1 \times \ldots \times L_n \rightarrow L]$	
$(kdcomb_L)$	$\dfrac{KD_1 : \lambda' \qquad KD_2 : \lambda''}{KD_1;\ KD_2 : \lambda' \vee \lambda''}$	

Table 4.17: Annotation type rule for the key declarations

the keys of the asymmetric and symmetric cryptography and communication statements. Each declaration is entered into the variable map, λ, with a composite type similar to the table type in the normal type system. A key declaration has, in the annotation type system, the type $L_1 \times \ldots \times L_n \rightarrow L$. The labels for each of the n fields together yield the label of the encrypted package. Table 4.13 and Table 4.12 shows how this is used in the cryptographic statements.

$$(sys_L) \quad \frac{KD : \lambda \qquad \lambda \vdash P : proc}{[KD]P : sys}$$

Table 4.18: Annotation type rule for the system

The system, typed by (sys_L) in Table 4.18, simply types the key declarations to get λ which is used in checking the list of processes.

4.3 Type Matching Communications Analysis

The Type Matching Communications Analysis, TMCA, is a simple analysis which notes occurrences of communication statements in a program, and attempts to find out if they are matched. A communication statement is matched if there is another communication statement such that the communication can be carried out.

The analysis begins by traversing the program recording a some of information concerning each communication statement. The following pieces of information are gathered:

- The type of operation, send or receive
- Asymmetric or symmetric cryptography
- The key declaration for the key used in the communication
- Iteration, is it inside a branch, a loop, or in the normal flow
- The principal for the current process

After this information has been collected, for each communication statement, it is matched to verify that everything that is communicated is matched. The nature of the analysis, however, is that it is an over-approximation. This means that there may be communications that is matched by the rules but does not have a match if the program was executed. However, if a communication statement is not matched it really cannot be matched.

The assertion that the analysis is an over-approximation is connected to the matching rules below. In itself, the gathering of information is mute towards the nature of the analysis, but the rules for iteration are constructed in a way that, if at all possible, it will try to match a statement. In contrast, an under-approximation would match only those statements that it could, with absolute certainty, be sure would actually communicate. In this case there might be statements that still matched, but if the analysis said two statements match, they would match in the execution of the program. The data gathered is insufficient to allow the matching rules to make this kind of distinction.

4.3.1 Matching Rules

The tuples of information recorded for each communication statement must be matched after a rigid number of criteria:

1. A send statement must be matched to a receive statement, and vice versa
2. Communication statements can only match statements of the same type, asymmetric matches asymmetric, symmetric matches symmetric
3. The key declarations must be the same
4. Since communication is synchronous a statement cannot be matched to other statements from the same process

These rules leave the iteration information. When this information is gathered loops have higher precedence than branches which again have higher precedence than the normal flow. In other words, it does not matter if a while loop is inside an if statement, or if an if statement is inside a while loop, a communication statement on the inner most level may be performed zero or more times.

In matching the iteration communication statements in the normal flow take highest precedence, since it is not known whether a statement inside a branch or loop will even be executed it is more important to see if those statements in the normal flow can be matched. Given a program with

```
ssend(x, b){k}
```

for process A and

```
while b do
    ssreceive(x; b){k}
endwhile;
ssreceive(x; b){k}
```

for process B, then the loop in process B may never be executed and the analysis should approve this program.

The above leads to a matching algorithm where the occurrence of a statement in the normal flow opposite a branch or loop will first try and match other communication statements in the normal flow first. Only if the statement does not match other statements in the normal flow will it match the branch or loop opposite it.

If a branch matches another statement (after checking as described above) it is removed in the same fashion as a statement in the normal flow. A loop, however, is pressed onto the list of statements to match again, so it is matched against all the other statements because a loop may be executed an arbitrary number of times.

If a statement cannot be matched, an appropriate error message is output.

CHAPTER 5

Implementation

To test the design of the language, type systems, and simple analysis, an implementation of each was carried out.

The implementation is based on the *Moscow ML* implementation of Standard ML [Sesff]. The strong pattern matching, high order functions, and type features of SML made it an obvious choice. The *Moscow ML* implementation was chosen for its availability on a large number of platforms, and use in previous work.

In this chapter some knowledge of SML is assumed. For further information see *Introduction to Programming using SML* [HR99] or similar. In the implementation of the typechecker the modules Set and Table are used [HR99, Appendix E].

After the discussion of the parser, Section 5.1, parse tree for the *gWhile* language, Section 5.2, and the implementation of the type systems, Section 5.3, Section 5.4 describes the Type Matching Communications Analysis. The testing procedure for the implementation is described in Section 5.5. Finally, the implementation of Battleships is discussed in Section 5.6.

5.1 *Parsing the* gWhile *Language*

The parser for the *gWhile* language is implemented in variants of *Lex* and *Yacc* for SML. For all intents and purposes the Lex and Yacc versions for SML are the same as those for the C programming language. The parser functions are inspired by earlier implementations [TH03].

In the implementation of the parser there have been a few changes to prevent shift/reduce and reduce/reduce conflicts. These changes have carried over to the abstract syntax shown in Section 3.1, and it is possible to directly implement the abstract syntax using Lex and Yacc.

There have been other changes, however, that have been made to ease the traversal of the parse tree. The abstract syntax for the key declarations, processes, asymmetric keys, initializations, and labels imply that they have a tree structure when there are several of them. For example, two processes are shown in the abstract syntax as a process tree which has two branches, each containing a process. In the implementation these trees are represented as lists as shown in Section 5.2.

At times it is useful to allow the else clause of an if statement to be omitted, this has been implemented in the parser. The returned syntax tree for such an if statement is equivalent to an if statement where the else branch contains a skip statement and nothing else.

Lastly, there is the matter of type specification in the key declarations. It is quite difficult to write $int \times int \rightarrow int$ using only ascii characters, and similarly for the $\tau_1 \times \ldots \times \tau_n \rightarrow crypt$ type for symmetric keys. In the implemented syntax the two-dimensional array, or table, is represented with the keyword `table`. The symmetric keys, however, are a bit different. Since a symmetric key is based upon a key declaration, and the key declarations are global and not used elsewhere in the declarations themselves. A symmetric key is, in a key declaration, denoted by the name of the key declaration it corresponds to. For example, the following key declaration can be given

```
declare d1 as {int{A:}, bool{B: A, C}}{A: all}
```

A key declaration for communicating this key as a session key would then have the form

```
declare d2 as {table{B: A}, d1{B: C}}{B: all}
```

This syntax means that the key declaration `d1` must be declared before `d2`, since it is referenced there.

The remaining available types in a key declaration are: `int`, `bool`, and `principal` for numbers, boolean values, and principals respectively.

It is also difficult to write subscript as used in the asymmetric keys. A public key in use is therefore simply written as `k+`, and declared as `k(d2)+`, if it should use the key declaration `d2`.

5.2 The gWhile *Parse Tree*

As mentioned above, the implementation of the *gWhile* language differs in some respects from the abstract syntax shown in Section 3.1. This section describes the datatypes for the implemented parse tree of the *gWhile* language.

```
type label = (string * string list) list;
                (* owner: reader1, reader2, ...; owner2: ... *)
```

Figure 5.1: Datatype for labels

Figure 5.1 show the datatype for labels. A label

$$\{A : B;\ B : A, C\}$$

is in the parse tree written as

$$[(\text{"}A\text{"}, [\text{"}B\text{"}]), (\text{"}B\text{"}, [\text{"}A\text{"}, \text{"}C\text{"}])]$$

Labels are discussed further in Section 5.3.2.

```
datatype expr =
      NUM of int
    | VAR of string
    | THIS
    | PRINC of string
    | TABLE of string * expr * expr
    | TRUE
    | FALSE
    | RAND of expr
    | NOT of expr
    | ADD of expr * expr
    | EQ of expr * expr
    | LT of expr * expr
    | DECL of expr * label
;
```

Figure 5.2: Datatype for expressions

The datatype for expressions, shown in Figure 5.2, closely follow the abstract syntax. The variables and table have their name which is used in the type system to look up their values and label. This, true, and false denote their values directly, while the remaining expressions either have a value or associated expressions which can be followed in a tree structure.

The statements in Figure 5.3 also follow the abstract syntax. Note the use of lists for the expressions and variables in the communication statements. The variables of the receive statements are denoted as strings to ensure that they are simply variable names. An assignment assigns to the first expression from the second, it is the job of the type checker to ensure that the first expression is a variable or table index. The `SRECETC` type is the

```
datatype stmt =
      ASS of expr * expr
    | SKIP
    | SEQ of stmt * stmt
    | IF of expr * stmt * stmt
    | WH of expr * stmt
    | ASEND of expr list * string
    | AREC of expr list * string list * string
    | SSEND of expr list * string
    | SRECETC of expr list * string list * string * string *
        stmt
    | SSREC of expr list * string list * string
    | INST of string
;
```

Figure 5.3: Datatype for statements

```
datatype init =
      INUM of string * int * label
    | IPRI of string * string * label
    | ITRU of string * label
    | IFAL of string * label
    | ITAB of string * int * int * label
    | IKEY of string * string * label
;
```

Figure 5.4: Datatype for the initialization

symmetric receive and act for statement, the final string of that statement is the principal which the current process wants to act for.

For the initialization in Figure 5.4 the first string for all the types is the name of the variable, and furthermore, each variable also has a label. A Number, simply has its numeric value. A principal variable has a string containing the name of the principal. Boolean variables are initialized specifically with a truth value. The table has its dimensions as numbers. And the key has the name of its associated key declaration.

```
datatype akey =
      PUBK of string * string
    | PRIK of string * string
;
```

Figure 5.5: Datatype for the asymmetric keys

Figure 5.5 shows the two types for the asymmetric keys. Both the public and private key has a type in the datatype. The first string is the identifier of the key, which is combined with a + for a public key and a − for a private key, to create the name of the key. The second is the key declaration for the key.

```
datatype process =
      PROC of string * akey list * init list * stmt
;

datatype keydecl =
      KD of string * keylabel list * label
;

datatype sys =
      SYS of keydecl list * process list
;
```

Figure 5.6: Datatype for processes, key declarations, and system

The final three datatypes, shown in Figure 5.6, are the processes, key declarations, and the system. A process has an identifier or principal, a list of asymmetric keys, a list of initializations, and a statement tree. The use of lists differ from the tree structure of the abstract syntax as described in Section 5.1 above. SML has a large number of function that work on lists and make list traversal more elegant than tree traversal.

A key declarations is comprised of a name, a list of type names and labels, and the label of the encrypted package. The list in the key declaration is the format of the associated keys and is simply list with fields of the format

$$\text{string} * \text{label}$$

The system contains a list of key declarations and a list of processes.

5.3 Type System Implementation

In the implementation of the type systems the two systems, as described Section 4.1 and Section 4.2, were combined and implemented as one. While the block label, B, and the current principals, ρ, were unchanged the variable maps, λ and γ, were combined to one map. This new map, also called γ, has the format

$$\gamma : \mathsf{Var} \mapsto \tau \times \mathsf{Label}$$

for each variable x with the type τ and the label L. The value of τ would reside in the old map γ, while the label would be stored in λ. The new γ is represented by the SML type

$$\text{(string, basic\_type} * \text{label)table}$$

5.3.1 Data Types

The implementation is built around the `basic_type` data-type which is shown in Figure 5.7. These types are analogous to the basic types in Section 4.1.

```
datatype basic_type =
      T_INT
    | T_BOOL
    | T_PRINCIPAL
    | T_TABLE of basic_type * basic_type * basic_type
    | T_KEYDECL of (basic_type * label) list
    | T_KEY of (basic_type * label) list * (basic_type * label)
    | T_CRYPT
    | T_ENCRYPT
    | T_DECRYPT
    | T_ERROR
;
```

Figure 5.7: The basic type datatype

While the types for integers, booleans, and principals are quite straightforward, the remaining require a little bit of explanation. The type for a table is initialized as

T_TABLE(T_INT, T_INT, T_INT)

The key declaration is initialized from the written declaration, the declaration

```
declare d as {int{A: B}, bool{}}{A: all}
```

would in the type system implementation be initialized as

T_KEYDECL [(T_INT, {A: B}), (T_BOOL, {})]

when this key declaration is put into the variable map γ it is as a tuple, with the key declaration as the first element and the label, $\{A : all\}$ as the second.

The differences between symmetric and asymmetric keys in the implementation lie not in the type for the key, but in the type for the result. In the two type-systems the type for a symmetric key was represented as

$$T_1 \times \ldots \times T_n \rightarrow crypt$$
$$L_1 \times \ldots \times L_n \rightarrow L$$

in the combined map, γ, this becomes

$$(T_1 \times L_1) \times \ldots \times (T_n \times L_n) \rightarrow crypt \times L$$

The asymmetric keys are initialized in the same way as the symmetric keys with the difference that they use *encrypt* and *decrypt* for public and private keys respectively.

A symmetric key initialized using the key declaration `d` from above would result in the following `basic_type`:

T_KEY([(T_INT, {A: B}), (T_BOOL, {})], (T_CRYPT, {A: all}))

This key type would then be put into γ together with the label for the key.

The `T_ERROR` type is used in the error handling. If an expression cannot be checked it will output an error message and return the error type. This will, of course, result in some follow-on effects, but will allow the checking of the program to continue to catch as many errors as possible.

5.3.2 Label Implementation

In the previous, I have used the syntax $\{A : B\}$ for labels. In the type checker, however, they are represented as

(string ∗ string list)list

The label $\{A : B, C;\ B : A\}$ would then become

$$[("A", ["B", "C"]), ("B", ["A"])]$$

In addition to the labels themselves, there are also the label functions: Restriction operator, $\sqsubseteq$; least upper bound operator, $\sqcup$; $owners(L)$; and $readers(O, L)$. Both $\sqsubseteq$ and $\sqcup$ are dependent on the $owners$ and $readers$ functions so they will be discussed first.

The owners function simply traverses the label as a list and inserts each owner into the set of owners. The empty label becomes the empty set. The set of owners is represented by the Set module mentioned above, which makes sure that there are no repetitions in the set, thus enforcing the condition that an owner is unique in a label.

The readers function takes two arguments, a label and a string, containing the name of an owner. The label is traversed and if the owner is found, the list of principals–the readers–is converted to a set. Before the set is returned the owner is inserted into the set since he is an implicit reader. If the owner is not in the label, the empty set is returned, this is in opposition to the definition of the readers function, but the only place this is a problem is in the $\sqcup$ function where it is handled.

The least upper bound operator is a function which takes two labels. It fetches the union of the owners of the two labels using the $owners$ and the Set.union functions. The union of owners is traversed to create the intersection of readers for each previous label. Remember that join of two labels, for example $\{A : B\} \sqcup \{A : B, C;\ B : A\}$ should result in the label $\{A : B;\ B : A\}$. In other words, the absence of the owner B from the first label should not affect the result. However, the readers function returns the empty set for an owner not in the label which would remove all readers for the given owner, as per the definition of intersection on sets. Checking whether an owner was in the original label allows the emulation of the wanted behavior, and simply return the readers of the other label for the owner.

As discussed in Section 2.1.2 there is, in addition to the binary join operator, the join of finite sets of labels, $\bigsqcup S$. This is emulated by a function which accepts a list of labels and join all the elements using the binary join on two labels at the time.

The restriction operator is a function which uses some of the high order functions trickery available in SML. The function checks the first condition of the restriction operator using the Set.subset function. If that is fulfilled it uses the fold functionality to check, for each owner of the first label, whether the second condition holds as shown in Figure 5.8.

```
Set.subset(owners(L1), owners(L2)) andalso (List.foldr ssfr true
    L1)

where ssfr is defined as

val ssfr = fn ((ow, _), b) =>
    Set.subset(
        readers(L2, ow),
        readers(L1, ow)
    ) andalso b
```

Figure 5.8: Using the fold functionality in the restriction operator

5.3.3 Type Checking

Expressions

The type system implementation with respect to expressions is quite simple, and follow the type systems as laid out in Section 4.1 and Section 4.2. Worth remembering is that the combination of the two type systems mean that the type function for expressions has the format shown in Figure 5.9 where the string set is ρ and the table is γ.

```
val CheckExpr = fn : string set * (string, basic_type * label)
    table * expr -> basic_type * label
```

Figure 5.9: Format for the type function to check expressions

For the constant expressions this is quite simply the type, as seen in Figure 5.7, and the empty label. Variables are looked up in the type environment, γ, since a table lookup returns an option type, using `NONE` if the variable was not in the table, the value returned from the table must be rudimentarily checked to return the tuple. Looking up a table also involves checking the indexing expressions, and verifying that it really is a table. The label returned from the type rule is the least upper bound of the label for the label of the table, L, and each of the indexing expressions, L_1 and L_2.

Composite expressions simply check the expressions they are made from and return the appropriate type and label as per the type rules.

Statements

Statements are checked using a type function with the format of Figure 5.10. In addition to ρ and γ as for expressions above, the first label in the function definition is the block label. The function returns `NONE` if the statement does not type, and `SOME T_STM` otherwise.

```
val CheckStmt = fn : label * string set * (string, basic_type *
      label) table * stmt -> large_type option
```

Figure 5.10: Format for the type function to check expressions

The first statement checked by the type function is the assignment. Both assignments to variables and table cells is handled in the same function. The assignee is checked with a function `GetAssType` with the format shown in Figure 5.11.

```
val GetAssType = fn : string set * (string, basic_type * label)
      table * expr -> basic_type * label * label
```

Figure 5.11: Format for the function which gets the type and label of an assignee

The reason for using a special function to fetch the type and label of an assignee is the case where

```
t[h][1] := l
```

as discussed in Section 4.2.3. The first label is simply the label found from looking up in γ. For a variable the second is just the empty label, for a table, however, the second returned label is the join of the label for each of the two indexing expressions.

The skip statement always type, the sequence statement checks each statement, and the if and while statements simply check that the expression is boolean and checks each of the attached statements with the augmentation of the block label as described in Section 4.2.1.

The next statements that should be examined are the communication statements. The type rules for the two send statements are almost identical, and likewise for the asymmetric and the simple symmetric receive. The receive and act for statement have the same rules as the the other receive statements, but also checks the principal and connected statement.

```
val (TTe, LLe) = ListPair.unzip(map (fn e =>
      CheckExpr(rho, gamma, e)) ee);
```

Figure 5.12: Using unzip and map to get the types and labels as separate lists

The send statements look up the key in γ and gets a list of types and a list of labels for the expressions, that should be sent, as shown in Figure 5.12. The type of the key is verified, and using `ListPair.all` the types are compared. The labels are checked using the function `LabelsAreLegal` with the format in Figure 5.13.

```
val LabelsAreLegal = fn : label * label list * label list * label
    list -> bool
```

Figure 5.13: Format for the function which checks that labels are legal

The function can be used to verify both that labels for send statements are legal, as well as the labels on the variables of receive statements. For all uses the first label is the block label. The remaining three parameters depend on the use. For send statements the first list is the empty list, the second is the field labels from the key, and the last are the labels for the expressions. For receive statements the first list is the labels for the expressions, LL_e, the second is the labels for the variables, LL_v, and the third are all the field labels from the key, LL_k.

First `LabelsAreLegal` computes B'. If the first list is empty, as is the case for send statements, B' is simply B. Otherwise, the first j fields are taken from the LL_k and joined with LL_e and B. Since the first list is empty for send statements $j = 0$ and no labels are taken from LL_k. The first j fields from LL_k are then dropped to make a list of labels corresponding to LL_v.

Using `ListPair.foldl` the restriction condition of the statements are checked as shown in Figure 5.14

```
ListPair.foldl (fn (Lv, Lk, b) => restr(lub(Bprime, Lk), Lv)
    andalso b) true (LLv, LLk)
```

Figure 5.14: Checking the restriction condition of communication statements

For send statements LL_e is the empty list, LL_k is the list of expressions, and LL_v is the list of labels from the key. This means that the condition is reversed compared to the receive statements as should be the case.

The receive and act for statement then checks that the principal A is an owner of the label of the encrypted package. If that is the case the statement S is checked using B' for the block label, $\rho \cup \{A\}$ for ρ and γ.

The `instantiate` type rule simply looks up the key in γ and checks that it is a symmetric key.

Initialization, Asymmetric Keys, and Key Declarations

The initialization, asymmetric keys, and key declarations are all type rules that return a version of γ. In the type system it seems that each returns subsection of γ and each of these are combined. In the implementation the key declarations return a γ, this is used in checking the asymmetric keys which returns a combination of the γ form the key declarations and the maps for each of the keys, and similarly for the initializations. This is done through three functions: `GammaAfterDecl` for the key declarations, `GammaAfterAkey` for the asymmetric keys, and `GammaAfterInit` for the initializations with the formats shown in Figure 5.15.

```
val GammaAfterDecl = fn : keydecl list -> (string, basic_type
      * label) table

val GammaAfterAkey = fn : (string, basic_type * label) table *
      akey list -> (string, basic_type * label) table

val GammaAfterInit = fn : (string, basic_type * label) table *
     init list -> (string, basic_type * label) table
```

Figure 5.15: Formats for the functions that alter γ

Each initialization simply returns the type and specified label as a tuple for the variable identifier which is then put into γ by `GammaAfterInit`. The only deviation from this behavior is the key initialization. For a key the associated key declaration is looked up in γ and returns the type-label tuple as

(T_KEY(TL, (T_CRYPT, fL)), L)

for the key declaration

(T_KEYDECL TL, fL)

The type function for asymmetric keys is almost identical to the initialization of symmetric keys. The only differences are the use of `T_ENCRYPT` and `T_DECRYPT` instead of `T_CRYPT`, and that the asymmetric keys do not have a label (or rather, they have the empty label).

The key declarations simply return the type-label list of the fields, `TL`, and the label of the encrypted field, `fL`, as

T_KEYDECL(TL, fL)

Processes and System

The system type function gets γ from the key declarations and pass it onto each of the process type functions which check the specified process. The processes augment γ through the `GammaAfterAkey` and `GammaAfterInit` functions and check the statements of the process with the empty label as the block label, the singleton set of the process principal as ρ, and γ from the `GammaAfterInit` and `GammaAfterAkey` functions.

5.3.4 Error Handling

Errors that occur while type checking a program are handled by a number of `PrintError` functions. These functions print an error and either return an appropriate value or, in the case of a fatal error, raise a `FatalError` exception. If a value is returned the type checking continues and try to catch more errors. There are two basic versions of the `PrintError` functions, the normal `PrintError` and the `PrintFatalError`. The `PrintError` function accepts an error message and a value it will return after printing the message, the fatal error function raises the `FatalError` exception after printing the message. Each of these error functions come in one more version, `PrintErrorIn` and `PrintFatalErrorIn`, which accept an additional argument indicating the troublesome part of the program, this could for example be the statement where an error occurred.

5.4 Type Matching Communications Analysis

The implementation of the TMCA is also done in SML. The analysis attempts to match communication statements that allow a communication to be carried out. This is achieved by first gathering a set of information for each communication statement, and then matching the statements according to the rules in Section 4.3.1.

The analysis starts by using the asymmetric key definitions and key initialization to build a table of key names pointing to their respective key declarations. In essence a

(string, string)table

This table will for a key name find its associated key declaration.

The statements of the program are then traversed to gain information about communication statements. For each of the communication statement a tuple is populated with the information of the statement which is passed up the parse tree and combined to form a list of tuples.

The tuples from the statements are populated using the data-types shown in Figure 5.16 and a couple of standard SML data-types to form tuples with the following fields: type of operation, type of cryptography, the key

```
datatype operation = SEND | RECEIVE;
datatype crypto = ASYMMETRIC | SYMMETRIC;
datatype iteration = ONE | BRANCH | LOOP;
```

Figure 5.16: Data types for fields in the TMCA tuple

declaration, the iteration, and the current principal. These fields correspond to the format

$$\text{operation} * \text{crypt} * \text{string} * \text{iteration} * \text{string}$$

The key declaration is entered into the tuple simply with its name. For two communication statements to be able to communicate they must use the same key, and therefore the same key format, this was the easiest way to ensure that is obeyed. Using the key name instead of the name of the declaration is not safe since it may be different in each process, the declarations are global and therefore their names are too.

After the list of tuples has been generated the TMCA will attempt to match all the tuples using the rules in Section 4.3.1.

The matching is carried out by two functions: `AnalyseResult` and `MatchTuple`. `AnalyseResult` will remove the first tuple from the list of tuples, and attempt to match it in the rest of the list using `MatchTuple`. `AnalyseResult` returns a boolean value reflecting whether all the tuples have been matched or not. After getting a result back from `MatchTuple`, `AnalyseResult` will recursively check the remainder of the list to see if the tuples there match. If the false value is returned from `MatchTuple`, an error is output.

The `MatchTuple` function returns a boolean value saying whether a tuple was matched, and the list of tuples as it looks after the matching procedure has been carried out. The returned list is in reality what is used in `AnalyseResult` to check the remainder of the list. With the matching rules set forth in Section 4.3.1 a branch or loop that has not been matched–by reaching the end of the list–simply return true to signify that they match, refer to the discussion in Section 6.2 on branches in TMCA for an idea of how this could be made more accurate. A tuple in the normal flow reaching the end of the list without being matched result in the boolean value false.

The first step in matching a tuple with another tuple inside the list, is to extract the contents of each tuple. These contents are then checked as follows: The operation type is not equal–since there are only two states for the operation, send or receive, checking that they are not equal is sufficient. The cryptography type is equal–asymmetric cannot match symmetric and vice versa. The key declarations must match. And finally, due to the synchronous nature of the communication statements, the principal for the process of the statements must not be equal. If the contents do not match after these rules, the current tuple is checked against the remainder of the list. If the tuples do match for these conditions, however, an examination of the iteration field of the tuples is needed.

Matching the iteration is the most complex part of the analysis. If both statements are in the normal flow, they are simply matched and the boolean value true and the list with both removed is returned. If only one of the statements is in the normal flow, however, the remainder of the list is recursively checked to see whether two statements in the normal flow can be matched. After attempting this match for the remainder the return value is checked to see whether the statement in the normal flow was matched. If that was the case, the true value and list as returned from the recursive call are simply returned. Otherwise, current tuples are matched, and the function returns true and depending on the iteration of the statement not in the normal flow, the list with or without it–a branch statement which is matched is removed from the list, same as a statement in the normal flow, while a statement inside a loop is checked again since it can match more than once.

5.5 Test

The parser was tested continuously during the implementation of the type system and TMCA. The test was performed by matching each part of a test file with the parse tree output from the parser. Further testing of the parser was not done.

The test for the implementation of the type system and the TMCA were mostly complete functional tests. Each test case was programmed in the file `test.sml`. To allow for some simplifications the test cases have been entered simply as parse trees, and have not been parsed from files. A file `test.w` has been included in the `examples` folder, see Appendix C.1, which is comprised of all the non-fatal test cases. Examples of fatal test cases are doubly defined variables or key declarations.

The results of each test case is shown in Appendix D, where the conventions for the test cases are also described.

The tables in Appendix D do not show all the possible test cases. There are some cases that are equal in structure. This is most notable with the asymmetric and symmetric communication statements. The implementation of the asymmetric send is the same as the symmetric send with the only difference in the type the key leads to, and similarly for the simple receives.

Some of the more convoluted expressions possible within the Decentralized Label Model are not specifically tested, but are tested through the simplification of the test cases.

In testing only minor errors were found. In other words, there were no major problems with the implementations or theory of either the type system or TMCA. One notable error was the reversal of the annotation conditions on the asymmetric and symmetric send statement. From Section 4.2 the condition

$$B \sqcup L_i \sqsubseteq L_{ki}$$

is known, which is the inverse of the receive statements. The error meant the condition was checked as

$$B \sqcup L_{ki} \sqsubseteq L_i$$

This error had the implication that variables could be sent which should have been flagged.

The final tests of the type system and TMCA found no errors.

5.6 Battleships

This section describes the differences in the Battleships implementation compared to the prevalent modus operandi as outlined in Section 1.2.

The first change was of course the conversion to a computer game. Since the game is designed around two players an obvious choice was to have to two players communicate over a network such as the Internet. Most computer games today based exclusively on networking use a client server architecture where the server has access to the world of the game, and the server is the only one who makes changes to the world based on the actions of the players. This is the implementation used for this example.

To simplify the problem, a number of further changes were made. The players still place their ships on the board, but the players themselves are modeled as automatic processes, A and B, and randomly put ships with a size of one onto the board. The server "tosses a coin" using the random function of the *gWhile* language to decide who starts. The players also randomly decide on a set of coordinates to target and send off to the server. After checking the coordinates the server sends a message to each player saying if it was a hit, if the game is over, and the coordinates.

The data sent to the players are in the implementation of the game simply ignored. In further implementation an extension to the *gWhile* language could allow this data to be written to the screen. In *JIF* as described in [ML00] this is handled through channels which handle any input and output from a program.

Key Declarations

Battleships has a large number of key declarations since both symmetric keys for communicating data and asymmetric keys for communicating the symmetric keys are needed. Using parameterization of the key declarations, as discussed in Section 6.2, would halve the number of key declarations needed since each key declaration could be used for transmitting similar data to each of the two players. Furthermore, *gWhile* is a strongly typed language and necessitates a specific key declaration for each format of data that needs communicating.

```
# key declarations for the symmetric keys of A
declare Appp as {principal{}, principal{},
    principal{}}{A: all};
declare Appt as {principal{}, principal{}, table{A:}}{A:
    all};
declare Appb as {principal{}, principal{}, bool{}}{A:
    all};
declare Appii as {principal{}, principal{}, int{},
    int{}}{A: all};
declare Appbbii as {principal{}, principal{}, bool{},
    bool{}, int{}, int{}}{A: all};
```

Figure 5.17: Key declarations for the symmetric keys used in communication between A and the server

Figure 5.17 shows the key declaration for the symmetric keys used in communication between the principal A and the server. The keys based on these declarations have been referred to as session keys. Five further key declarations are declared for communication from A to the server. They are used for the asymmetric keys necessary to communicate the session keys.

Key declarations for the corresponding communication between B and the server have also been declared.

The Players

The two player processes are carbon copies of each other with different principals. The discussion of one is therefore the same as the discussion of the other. In this description, only the process for A will be regarded.

The process starts with the definition of the five asymmetric keys. This is followed by the initialization, Figure 5.18. All the variables for the process have very restrictive labels, which is the most conservative configuration for data, and is used to prevent any illegal data flow. Among the initializations are of course the five symmetric keys, at the bottom of Figure 5.18, corresponding to the key declarations of Figure 5.17. In the description below, I do not simply note each time a key is instantiated and sent to the server, as this is done before every symmetric communication. There are a few cases which are a little different, these will be described briefly. Each session key is reused throughout the game after it is initially communicated.

```
server{A:} := 'S',
opponent{A:} := '',

boardSizeH{A:} := 10,
boardSizeW{A:} := 10,

board[10][10]{A:},
numShips{A:} := 10,

i{A:} := 0,
x{A:} := 0,
y{A:} := 0,

myTurn{A:} := false,
done{A:} := false,
result{A:} := false,

key kAppp{A: S} using Appp,
key kAppt{A: S} using Appt,
key kAppb{A: S} using Appb,
key kAppii{A: S} using Appii,
key kAppbbii{A: S} using Appbbii,
```

Figure 5.18: Initializations for process A

The process for a player starts by instantiating a key, the key `kAppp`. This key is sent to the server which stores the principal for the player and the key. The server then uses the key to tell the player who his opponent is, the principal he is playing against. The first asymmetric communication serves both to send the first session key, but also to say that the principal is ready to play a game and who he is.

After receiving his opponent from the server, the player places his ships on the playing field, using the random function to find the coordinates where each ship is placed, and an if statement to check that two ships are not placed at the same location. The board is sent to the server which sends back a boolean true value once both boards have been received.

At this point the game itself starts and the session keys used within the game loop are communicated before the loop is entered. The loop uses the value of the variable `done` which is initialized to `false` to iterate over, a new value of `done` is received after each player has had a turn to see if the game has concluded.

```
if myTurn then
    x := random(boardSizeW);
    y := random(boardSizeH);

    ssend(server, this, declassify(x, {}), declassify(y,
        {})){kAppii}
else
    ...
```

Figure 5.19: Targeting a pair of coordinates

Each turn the player receives a boolean value from the server, indicating whether it is his turn or not. If the player has his turn he generates a set of coordinates, Figure 5.19, and sends them to the server. In the player processes the coordinates have quite restrictive labels, and they have to be declassified for the server to be able to read them and send them to the opponent. If it is not his turn he waits to receive the coordinates from the server that his opponent is targetting. After receiving the coordinates, the player sends a boolean true value back to indicate that the server may declassify the board value at those coordinates, as described in Section 2.2.

Finally, regardless of turn, the player receives a message back from the server with the result of the shot, the new value of `done`, and the coordinates that were targeted.

As discussed in further detail in Section 6.2 there are several places in each player process, where user interaction could be injected.

The Server

The server has defined asymmetric keys for communicating with both players. However, while the players have the public keys for communicating with the server, the server only have the private keys defined. The initializations for the server are shown in Figure 5.20. Notice in particular that `board1` is the board for player 1, while `hit1` is the table showing where player 2 has shot, and `hitShips1` contains the number of ships player 2 has hit. The very restrictive labels for the boards are used so that S cannot access these variables outside the act for blocks.

The server process starts by receiving a session key from each principal. The session key is used for sending each player as an opponent to the other. Then the server receives the board from each player, and sends back the boolean value `true` to indicate that the game is about to start.

Using the random function, Figure 5.21, the server selects which player starts, and enters the game loop. Just inside the loop a boolean value for each player indicating if the current turn is his, is sent as shown in Figure 5.22.

The game logic has been unfolded into two parts, each specific to one player. The two parts of the unfolding are equal except for the principals. The description will be of the part which is for player 1, but it is equally applicable for player 2.

First the coordinates are received from player 1 and the server will act for that player for the rest of the turn. The coordinates are forwarded to player 2, who sends back the boolean value `true`, to indicate that the server may act for him, Figure 5.23, and declassify the value for the board at the target coordinates. The declassified value is assigned to the variable `boardValue`.

`boardValue` is checked to see if there is a ship there, if a ship is present the table of shots from player 1 is checked. The shot is only a hit if there is a ship at the coordinates and player 1 has not shot there before. Remember that the coordinates are generated randomly so there is a chance that the same coordinates are generated several times in a game. In the board game version of Battleships you normally loose your turn if you shoot at coordinates you have shot at before, this is also the case in this implementation.

If there is a ship at the coordinates and it is the first time player 1 has targeted those coordinates then it is noted as a hit, the table of shots by player 1 is updated, and the number of ships hit by player 1 is incremented. The value for `done` is set to true if the number of ships that have been hit by player 1 is equal to the total number of ships.

The server sends a message to each player with the result of the shot, the value of `done`, and the coordinates.

Finally, the `turn` variable is updated to reflect that it is the other player's turn.

```
player1{} := '',
player2{} := '',

key kAppp{A: S} using Appp,
key kAppt{A: S} using Appt,
key kAppb{A: S} using Appb,
key kAppii{A: S} using Appii,
key kAppbbii{A: S} using Appbbii,

key kBppp{B: S} using Bppp,
key kBppt{B: S} using Bppt,
key kBppb{B: S} using Bppb,
key kBppii{B: S} using Bppii,
key kBppbbii{B: S} using Bppbbii,

numShips{} := 10,

# board for A and where B has shot
board1[10][10]{A:},
hit1[10][10]{B: A},
hitShips1{B: A} := 0,

# board for B and where A has shot
board2[10][10]{B:},
hit2[10][10]{A: B},
hitShips2{A: B} := 0,

done{} := false,
hit{} := false,

boardValue{} := 0,

turn{} := 0,

x{} := 0,
y{} := 0
```

Figure 5.20: Initializations for the server

```
# select the starting player
turn := random(2);
```

Figure 5.21: Using random to select the starting player

```
ssend(player1, this, turn = 1){kAppb};
```

Figure 5.22: Indicating turn of player with a boolean value

```
sreceive(this, player2, true;){kBppb} andactfor B in
    boardValue := declassify(board2[x][y], {})
endactfor;
```

Figure 5.23: Acting for player 2 to declassify board value at the target coordinates

5.6.1 Verification of Battleships

After the battleships program had been implemented it was checked using the implementation of the type checker. There were quite a few problems with respect to the annotation type system, not so much in the server, but in the player processes.

First and foremost there were a number of restrictions due to implicit flow. Since all the variable are defined with the very restrictive $\{A :\}$ label, for principal A, all assignments inside branches or loops were restricted with this label in the block label. One such example is a simple communication statement as shown in Figure 5.24.

```
ssend(server, this, true){kAppb}
```

Figure 5.24: Simple communication statement which is affected by implicit information flow control

Since the block label within the game loop is $\{A :\}$ the conditions for this statement, which from the type rule for the symmetric send statement is defined as

$$B \sqcup L_i \sqsubseteq L_{ki}$$

which for each of the three fields becomes

$$\begin{array}{rrcl} & B \sqcup L_1 & \sqsubseteq & L_{k1} \\ \Rightarrow & \{A:\} \sqcup \{A:\} & \sqsubseteq & \{\} \\ \Rightarrow & \{A:\} & \sqsubseteq & \{\} \\ \\ & B \sqcup L_2 & \sqsubseteq & L_{k2} \\ \Rightarrow & \{A:\} \sqcup \{\} & \sqsubseteq & \{\} \\ \Rightarrow & \{A:\} & \sqsubseteq & \{\} \\ \\ & B \sqcup L_3 & \sqsubseteq & L_{k3} \\ \Rightarrow & \{A:\} \sqcup \{\} & \sqsubseteq & \{\} \\ \Rightarrow & \{A:\} & \sqsubseteq & \{\} \end{array}$$

A solution to this problem is to declassify the value of `done` in the condition for the game loop. The branch on the current turn has the same problem, due to the label of the `myTurn` variable. Since the statement in Figure 5.24 is inside both the game loop and the branch both cases must be declassified.

Declassifying both the condition of the game loop and of the turn branch solved the problems for the two last fields of the statement. However, the first field sends the value of `server` to address the server. The variable has the restrictive label $\{A:\}$, but the field in the key declaration asks for the empty label. There are three possible solutions to this problem

1. Correct the key declaration to use the restrictive, $\{A:\}$, label
2. Declassify the value of `server` each time it is used
3. Set the label of `server` to the empty label

Option one is not viable, the implicit flow problems associated with the more restrictive label would be destructive to other fields in the key formats, for example in the declassification of the board value shown in Figure 5.23 of the server. The second option would work fine, and so would option three. However, the dangers of letting the name of the server flow is not very great, and this option was chosen.

```
if hitShips2 = numShips then
    done := true
endif
```

Figure 5.25: An example of illegal implicit flow in the server

In the server there was only one real problem found by the verification. When the server checks if a player has hit all the ships of his opponent as shown in Figure 5.25.

As shown in Figure 5.20 `hitShips2` has the label $\{A : B\}$. From the implicit flow condition `done` cannot be assigned a value inside the branch since it has the empty label. The solution is to declassify the value of `hitShips2` to the empty label and accept the minute information leak associated with this; after all, both players have to know that the game is over.

5.6.2 Introducing Leaks

One of the purposes of the security annotations, as in the Decentralized Label Model, is to catch and disallow information leaks. To see some examples of this in action, some leaks are introduced into the example program.

Since there is no communication directly between the two players, only leaks from the server are regarded. The server is trusted to not behave maliciously, this means that the only point of interest is unintentional information leaks from the server.

One error that can be introduced which will result in a leak is on line 138 of Appendix C.2. The if condition

```
if declassify(hit2[x][y], {}) then
```

inside the block where the server acts for player 1, can be changed to

```
if declassify(hit1[x][y], {}) then
```

The only change is that `hit2` has been changed to `hit1`. This change would result in player 1 being able to shoot the same ship of player 2 several times, since the place where his shots is recorded is not the same that is checked. This is an example of a place where a bug can be used to cheat. This bug will be caught. Since the owner of `hit1` is simply the principal B, or player 2, and the server is acting for A, player 1, the owner B cannot be removed from the label.

On the other hand, the obvious bug which would result in an information leak, on line 135 of Appendix C.2, cannot be caught. The bug is reproduced by changing `board2` to `board1` in

```
boardValue = declassify(board2[x][y], {})
```

At this point in the program the server is acting for both player 1 *and* player 2. To be able to catch this leak, the block where he acts for player 1 must be ended before he can start acting for player 2. This could, for example, be achieved with the `donotactfor` statement suggested in Section 6.2. With the constructs present in the language, the act for block of player 1 could end before the server begins acting for player 2. After declassifying the board value, the server would stop acting for player 2, and could get a new communication from player 1 to start acting for him again. This would result in at least one extra communication per turn, but would catch this bug. The possible leaks shown in this section have been put into the file `server-based-battleships-error.w` which is available with the rest of the program as described in Appendix C.1.

CHAPTER 6

Discussion

6.1 Results

In this thesis I have achieved several interesting results. First and foremost I have studied the Decentralized Label Model in some detail to be able to include communication statements in a language which could be verified within the model.

Although *JIF* is mostly-statically checked using a type system, the type systems in this thesis was created from scratch, albeit inspired by prior work.

The labels of the Decentralized Label Model has some interesting properties, especially interesting is that the set of labels form a complete lattice ordered with the restriction operator, with the join operator finding the binary least upper bound. There are some conditions on a label which allow for the lattice of labels, specifically the condition that an owner is unique in the label, and that each reader is unique for a given owner. As mentioned before, allowing redundancy, which is what a reoccurrence of owners is, would mean that labels could go against the anti-symmetry condition of the partial order. The proof of the set of labels as a complete lattice is a step forward from the rather informal approach around the Decentralized Label Model in previous literature.

An important basis for this project was the notion of networked programs. To allow the verification of communication over an open medium, the concept of what correct communication is and how it can be verified must be decided. The decision to use a specific format for communication, through the definition and use of key declarations, mean that communication can be verified statically, something which is often a problem. To let the authority be propagated through the communication also allowed for further static verification of authority. The model for authority in *JIF* puts more verification on the run-time system. In addition, the use of tables had some interesting problems in comparison to normal assignments.

The simple analysis, TMCA, was the first step into what could be a number of useful analysis on programs in the *gWhile* language. The analysis is quite basic, and mentioned below is one approach which could make the result more accurate.

The implementations of the parser, type systems, and TMCA were merely a tool to see that the theory would work. The bulk of the project was in the design of the *gWhile* language and the type systems, especially with regard to the Decentralized Label Model and communication.

The example program, used to motivate the design of the language and type systems, was the game Battleships. Most of the errors that were found by the type system, were by-products of implicit information flow. The flow conditions from the Decentralized Label Model make for a particular mindset, to see why specifically labeling a value can cause a whole host of errors.

6.2 Future Work

There are many venues the results of this project may be developed on top of or used in further projects. This section contains a discussion of the elements that can be added to the work on the Decentralized Label Model or where the work in this thesis may be used in the future.

The original problem specification, shown in Appendix A, called for a source translation to allow the program to be executed. The security annotations would have to be translated into dynamic checks that could verify the continued adherence to the Decentralized Label Model. Although the language and verification were completed, a lot of work remains to allow for this translation. Most specifically, the analysis to attain the necessity of specific annotations, and therefore the use of specific dynamic checks in the translated code, had to be designed and implemented.

The early concepts for communication used pseudo principals $p1$ and $p2$ for player 1 and player 2. In the design of the verification rules the real principals were bound to specific key declarations in the definition of asymmetric or symmetric keys. This binding means that player 1 will always be A and player 2 will always be B in the implementation of Battleships.

One design idea which could be further developed is the use of parameterization in the key declarations. A key declaration could be declared as

```
declare d[p, q] as {principal{p:  q}, int{p:}, bool{q:  p}}{p:
                              all}
```

In the definition of this key the principals could be specified

```
key k{A: B} using d[A, B]
```

Allowing for this kind of parameterization, the pseudo principals $p1$ or $p2$ could be specified in the use of one process, and the analysis and type system could choose appropriate values for the pseudo principals in the verification. The pseudo principals would allow the players to be specified without tying them to specific principals. A game could therefore be played in which player 1 was B and player 2 was A or even a third or fourth principal, in contrast to the normal binding of the principals that has been used in the example program.

If two unique random principals could be allowed to play a game, the next logical extension is to allow a multitude of games on the server. You, as a player, could connect to the server which would see if there were any other players waiting for a game. You would then be put into a game with that principal.

Allowing a multitude of games would mean that the server had to have a multitude of boards, for example in a list of boards. The list could be addressed as

```
boards[playerid][x][y]
```

where `playerid` is a unique index for the principal of a given game. A player could play several games with a unique id in each, where the server would make sure that principal and game did not get mixed up.

The use of `playerid` to index the correct board means that the central if statement could be folded up again, using the index to attain the correct board and table of shots for the players. This would also necessitate a number of changes to the language and type system. First of all to allow for the list of tables, or 3-dimensional table; secondly to allow each index in the list to have a label specific to the player, of course using pseudo principals as mentioned above to avoid tying it to a specific principal.

If labeling of indexed values is allowed, the next step could be a different label on the value of each table cell than on the table itself. For a board for player 1 the table could be quite open with a label such as $\{p1 : S, p2\}$, while the cell values could be very restricted with the label $\{p1 :\}$.

One change which could be made to Battleships which would not necessitate any changes to the language nor the type system is in the initialization of the game boards. The idea is that the players do not store a single board themselves, but during the board initialization, they send the coordinates for each ship to the server which then places a ships at the specified coordinates. The server would send back a value depending on whether the ship was placed successfully or not, this value would tell the player to either find new coordinates for the ship or to send the coordinates for the next ship. This scenario is quite plausible in the use of networked computer programs and games, and would mean extra diligence on the part of the server. Especially if the placement of ships for each player was interlaced, as opposed to one player placing all his ships before the other got to place his.

The example program was based on a client-server structure. A peer-to-peer structure, where each player communicate directly with his opponent without a server, could also have been chosen. A peer-to-peer architecture would present some different problems from those discussed in this project. First and foremost, due to the problems of integrity of the client programs, the truthfulness of the each client had to be verified in one form or another. In the program in this thesis, the server is very authoritative and the only handled by each client during a game is the target. If each client was potentially compromised, and had the task of checking whether a target resulted in a hit on his own ships, the problem would be quite different from the problem addressed in this project, where the server is not willfully malicious.

User interaction is non-existent in the implementation of Battleships. To allow for some user interaction channels, as seen in *JIF*, could be employed. This would call for an additional *string* type, to allow for feedback to the user, which again would result in an augmentation of the type system.

Sometimes a process would, inside an act for block, temporarily not act for a give principal. In Battleships there is no need to act for A when getting the board value for B. The only way to do this in the *gWhile* language is to end the act for block and start a new one to act for the principal again. One way this could also be achieved is using a *do not act for* statement as shown in Figure 6.1.

$$\begin{array}{lcl} S & \in & \mathsf{Stmt} \\ S & ::= & \texttt{donotactfor}\ A\ \texttt{in}\ S\ \texttt{enddonotactfor} \end{array}$$

Figure 6.1: Syntax of do not act for statement

This statement would remove the principal A from ρ in the verification of S as

$$B;\ \rho \setminus \{A\};\ \lambda \vdash S : stm$$

and restore it before checking the rest of the program.

With regards to TMCA, the matching rules are somewhat basic. The matching rule, and information gathered, for if statements in particular could be extended. The current rules assume that statement within a branch of an if statement will either be run, or it will not. The nature of if statements, however, mean that one of the branches will be run. If there is only a communication statement in one of the branches the case laid out in the current matching rules is sufficient. If, on the other hand, there is a communication statement in each branch only one of them *will* be run, and this connection could be extended upon. There would have to be a coupling of the tuples for the communication statements, this association would, in the matching rules, be followed to ensure that at least one of the two was matched.

Finally, and perhaps most interestingly, the work of this thesis could be carried over to the use of the Decentralized Label Model in *JIF*. The current implementation of *JIF* only contains communication through channels. The rules for communication over an open medium could be merged into *JIF* to allow this kind of communication. This inclusion would by no means be trivial, but is one solution to the problems of secure communications. One thing to keep in mind about *JIF* is that there is no notion of multi-threaded programs, since two programs communicating would not be in the same file, extra diligence must be employed to ensure that the same key declarations are used.

6.3 Conclusion

This project was started with quite lofty goals which is also evident by the initial problem specification in Appendix A. The goal quickly became one of investigating the Decentralized Label Model with regards to networked programs, and specifically networked computer games.

To examine the application of annotations, the *gWhile* language was designed, with built-in labeling as defined in the Decentralized Label Model.

Communication and cryptography was added to the Decentralized Label Model for use in the *gWhile* language as required for Battleships. The use of key declarations allow for a complete static verification to ensure that there is no illicit information flow. Similarly, allowing the authority of principals to follow the ability to decrypt a package means the act for hierarchy can be checked statically.

The Type Matching Communications Analysis is the first attempt at analyzing programs in the *gWhile* language. The initial problem specification set the scene for a number of analyses to find out which annotations were necessary in a source code translation. The confines of this project, however, meant that only the TMCA was designed and implemented.

The example program, Battleships, was chosen for this project for its clear use of secrecy and networked nature. The client-server structure was selected due to its likeness with modern networked computer game architecture.

With regards to the example program, some problems could be caught as discussed in Section 5.6.2. However, the ramifications are not as great as I initially thought (or hoped) due to the intricacies of the Decentralized Label Model. To make sure that the labeling in a program makes sense, the development process should include a step specifically to ascertain the *correct* labels to use. Whether the time spent on assessing the labels could be equally well spent debugging a program outside the Decentralized Label Model is, however, an open question. Some of the limitations of the Decentralized Label Model with regards to the problem discussed in this project are probably due to the design of the Decentralized Label Model. Initially the model was made to restrict data flow from a program running on a computer, in this case the code–or the programmer–could be thought of as the malicious party. The idea to use the Decentralized Label Model to catch errors, resulting in illicit flow, has been introduced later.

Section 6.2 discusses further work that could be done using this project as a basis. Of greatest interest in my mind would be the marriage of the rules for communication within the Decentralized Label Model and *JIF*.

APPENDIX A

Initial Problem Specification

Programs working together over a communication medium, have a number of security requirements that must be met for them to function properly.

The communication must be secure with respect to attacks, for example man-in-the-middle or replay attacks. Furthermore, the programs must be safe from each other, in other words, they must not be able to cheat one another.

A programming language can be designed where security requirements can be specified directly in the source code. Programs in this language can be analyzed, using program analysis, to learn information about the security. In some cases it would be necessary to translate these programs into a lower level language, this is done to run them or enhance their performance. The information learned from the program analysis is used in the translation to ensure that the security requirements are still met.

To motivate the development I plan to study an example program, namely the game Battleships. It is an example of a game played by two players. Each player hides information from the other player - the location of his ships on the battle field. It is also a game in which a player will gain a large advantage by cheating.

The plan is to study an extension of the While language with parallelism, communication, and various security mechanisms like access control annotations and cryptographic primitives. The program analysis is likely to be a variant of a reachability analysis combined with security information.

APPENDIX B

User Guide

B.1 Installation Instructions

The installation of the *gWhile* tool requires the source code for the tool, as mentioned in Appendix C.1. Included with the source code is all the source files used by the tool, including the Set and Table data-types from [HR99].

In addition to the source code, the tool requires an installation of Moscow ML, available from

`http://www.dina.dk/~sestoft/mosml.html`

Furthermore, the `make` utility is needed.

To install the *gWhile* tool, the `Makefile` must be edited to reflect the location of the Moscow ML installation. The tool can then be built with the `make` command. This will build and start the tool. After it has been build once, the `make` command will just start the tool.

Using the command `make clean` will remove all the compiled files.

B.2 *Verifying* gWhile *Programs*

The tool is started using the `make` command from the terminal when in the `gwhile_tool/` folder. This will start the tool and prompt for a file name. To check the included `test.w` file enter

`../examples/test.w`

at the prompt. The tool will guide you through the rest of the process.

APPENDIX C

Source Code

C.1 Parser, Typechecker, and Analysis

The parser and typechecker are included as source files on the attached CD-ROM, alternatively they can be downloaded from

`http://www.jonas.rabbe.com/thesis/source-code.zip`

To build and run the tool for the *gWhile* language, see the installation instructions and user guide in Appendix B.

C.1.1 List of Files

The CD-ROM or zip file contains the directory structure and files shown below. The file `Readme` contains the text of Appendix B, while `Report.pdf` is a pdf version of this report.

```
Readme

Report.pdf

examples/

    server-based-battleships-error.w
    server-based-battleships.w
    test.w

gwhile_tool/
```

```
Makefile
Set.sig
Set.sml
Table.sig
Table.sml
dlm_operators.sml
gwhile_parser.sml
gwhile_prettyprint.sml
gwhile_tool.sml
gwhile_tree.sml
gwhilelex.lex
gwhilepar.grm
parseloader
test.sml
tmca.sml
tool.sml
type_checker.sml
type_functions.sml
type_types.sml
typeloader
```

C.2 server-based-battleships.w

```
# start with the key declarations
[
    # key declarations for the symmetric keys of A
    declare Appp as {principal{}, principal{}, principal{}}{A: all};
    declare Appt as {principal{}, principal{}, table{A:}}{A: all};
    declare Appb as {principal{}, principal{}, bool{}}{A: all};
    declare Appii as {principal{}, principal{}, int{}, int{}}{A: all};
    declare Appbbii as {principal{}, principal{}, bool{}, bool{}, int{},
        int{}}{A: all};

    # key declarations for the asymmetric keys of A
    declare aAppp as {principal{}, principal{}, Appp{A: S}}{A: all};
    declare aAppt as {principal{}, principal{}, Appt{A: S}}{A: all};
    declare aAppb as {principal{}, principal{}, Appb{A: S}}{A: all};
    declare aAppii as {principal{}, principal{}, Appii{A: S}}{A: all};
    declare aAppbbii as {principal{}, principal{}, Appbbii{A: S}}{A: all};
```

```
    # key declarations for the symmetric keys of B
    declare Bppp as {principal{}, principal{}, principal{}}{B: all};
    declare Bppt as {principal{}, principal{}, table{B:}}{B: all};
    declare Bppb as {principal{}, principal{}, bool{}}{B: all};
    declare Bppii as {principal{}, principal{}, int{}, int{}}{B: all};
    declare Bppbbii as {principal{}, principal{}, bool{}, bool{}, int{},
        int{}}{B: all};

    # key declarations for the asymmetric keys of B
    declare aBppp as {principal{}, principal{}, Bppp{B: S}}{B: all};
    declare aBppt as {principal{}, principal{}, Bppt{B: S}}{B: all};
    declare aBppb as {principal{}, principal{}, Bppb{B: S}}{B: all};
    declare aBppii as {principal{}, principal{}, Bppii{B: S}}{B: all};
    declare aBppbbii as {principal{}, principal{}, Bppbbii{B: S}}{B: all};
]

# then come the processes
S [
    SAppp(aAppp)−,
    SAppt(aAppt)−,
    SAppb(aAppb)−,
    SAppii(aAppii)−,
    SAppbbii(aAppbbii)−,

    SBppp(aBppp)−,
    SBppt(aBppt)−,
    SBppb(aBppb)−,
    SBppii(aBppii)−,
    SBppbbii(aBppbbii)−,
] :
(
    player1{} := '',
    player2{} := '',

    key kAppp{A: S} using Appp,
    key kAppt{A: S} using Appt,
    key kAppb{A: S} using Appb,
    key kAppii{A: S} using Appii,
    key kAppbbii{A: S} using Appbbii,

    key kBppp{B: S} using Bppp,
    key kBppt{B: S} using Bppt,
    key kBppb{B: S} using Bppb,
    key kBppii{B: S} using Bppii,
    key kBppbbii{B: S} using Bppbbii,

    numShips{} := 10,

```

```
    # board for A and where B has shot
    board1[10][10]{A:},
    hit1[10][10]{B: A},
    hitShips1{B: A} := 0,

    # board for B and where A has shot
    board2[10][10]{B:},
    hit2[10][10]{A: B},
    hitShips2{A: B} := 0,

    done{} := false,
    hit{} := false,

    boardValue{} := 0,

    turn{} := 0,

    x{} := 0,
    y{} := 0
)
{
    # recieve a starting key from each player
    areceive(this; player1, kAppp){SAppp−};
    areceive(this; player2, kBppp){SBppp−};

    # send opponent to each player
    ssend(player1, this, player2){kAppp};
    ssend(player2, this, player1){kBppp};

    # get keys that are needed for the communication of the board
    areceive(this; player1, kAppt){SAppt−};
    areceive(this; player2, kBppt){SBppt−};

    areceive(this; player1, kAppb){SAppb−};
    areceive(this; player2, kBppb){SBppb−};

    # get playing field from each
    ssreceive(this, player1; board1){kAppt};
    ssreceive(this, player2; board2){kBppt};

    # send values to indicate that we received boards
    # and are ready to start
    ssend(player1, this, true){kAppb};
    ssend(player2, this, true){kBppb};

    # get keys that are needed in the game loop
    areceive(this; player1, kAppii){SAppii−};
    areceive(this; player2, kBppii){SBppii−};

```

```
areceive(this; player1, kAppbbii){SAppbbii−};
areceive(this; player2, kBppbbii){SBppbbii−};

#select the starting player
turn := random(2);

#start the game itself
while not done do
    #send boolean values to indicate if it's that player's turn
    ssend(player1, this, turn = 1){kAppb};
    ssend(player2, this, turn = 2){kBppb};

    if turn = 1 then
        # then A has a turn

        # get target coordinates
        sreceive(this, player1; x, y){kAppii} andactfor A in
            hit := false;

            # send coordinates to B and recieve 'true' back
            ssend(player2, this, x, y){kBppii};
            sreceive(this, player2, true;){kBppb} andactfor B in
                boardValue := declassify(board2[x][y], {})
            endactfor;
            if boardValue = 1 then
                if declassify(hit2[x][y], {}) = 0 then
                    hit2[x][y] := 1;
                    hit := true;
                    hitShips2 := hitShips2 + 1;
                    if declassify(hitShips2, {}) = numShips then
                        done := true
                    endif
                endif
            endif;

            # send result to both A and B
            ssend(player1, this, hit, done, x, y){kAppbbii};
            ssend(player2, this, hit, done, x, y){kBppbbii}

        endactfor;

        turn := 2
    else
        # then B has a turn

        # get target coordinates
        sreceive(this, player2; x, y){kBppii} andactfor B in
            hit := false;

```

```
                # send coordinates to A and recieve 'true' back
                ssend(player1, this, x, y){kAppii};
                sreceive(this, player1, true;){kAppb} andactfor A in
                    boardValue := declassify(board1[x][y], {})
                endactfor;
                if boardValue = 1 then
                    if declassify(hit1[x][y], {}) = 0 then
                        hit1[x][y] := 1;
                        hit := true;
                        hitShips1 := hitShips1 + 1;
                        if declassify(hitShips1, {}) = numShips then
                            done := true
                        endif
                    endif
                endif;

                # send result to both B and A
                ssend(player2, this, hit, done, x, y){kBppbbii};
                ssend(player1, this, hit, done, x, y){kAppbbii}

            endactfor;

            turn := 1
        endif
    endwhile

}

A [
    SAppp(aAppp)+,
    SAppt(aAppt)+,
    SAppb(aAppb)+,
    SAppii(aAppii)+,
    SAppbbii(aAppbbii)+
] :
(
    server{} := 'S',
    opponent{A:} := '',

    boardSizeH{A:} := 10,
    boardSizeW{A:} := 10,

    board[10][10]{A:},
    numShips{A:} := 10,

    i{A:} := 0,
    x{A:} := 0,
    y{A:} := 0,

```

```
    myTurn{A:} := false,
    done{A:} := false,
    result{A:} := false,

    key kAppp{A: S} using Appp,
    key kAppt{A: S} using Appt,
    key kAppb{A: S} using Appb,
    key kAppii{A: S} using Appii,
    key kAppbbii{A: S} using Appbbii,
)
{
    instantiate kAppp;
    asend(server, this, kAppp){SAppp+};
    ssreceive(this, server; opponent){kAppp};

    while i < numShips do
        x := random(boardSizeW);
        y := random(boardSizeH);

        if board[x][y] = 0 then
            board[x][y] := 1;
            i := i + 1
        endif
    endwhile;

    instantiate kAppt;
    asend(server, this, kAppt){SAppt+};
    instantiate kAppb;
    asend(server, this, kAppb){SAppb+};

    # send board to server and receive true back
    ssend(server, this, board){kAppt};
    ssreceive(this, server, true;){kAppb};

    instantiate kAppii;
    asend(server, this, kAppii){SAppii+};

    instantiate kAppbbii;
    asend(server, this, kAppbbii){SAppbbii+};

    while not declassify(done, {}) do
        ssreceive(this, server; myTurn){kAppb};
        if declassify(myTurn, {}) then
            x := random(boardSizeW);
            y := random(boardSizeH);

            ssend(server, this, declassify(x, {}), declassify(y, {})){kAppii}
        else
            ssreceive(this, server; x, y){kAppii};
```

```
            ssend(server, this, true){kAppb}
        endif;

        ssreceive(this, server; result, done, x, y){kAppbbii}
    endwhile
}

B [
    SBppp(aBppp)+,
    SBppt(aBppt)+,
    SBppb(aBppb)+,
    SBppii(aBppii)+,
    SBppbbii(aBppbbii)+
] :
(
    server{} := 'S',
    opponent{B:} := '',

    boardSizeH{B:} := 10,
    boardSizeW{B:} := 10,

    board[10][10]{B:},
    numShips{B:} := 10,

    i{B:} := 0,
    x{B:} := 0,
    y{B:} := 0,

    myTurn{B:} := false,
    done{B:} := false,
    result{B:} := false,

    key kBppp{B: S} using Bppp,
    key kBppt{B: S} using Bppt,
    key kBppb{B: S} using Bppb,
    key kBppii{B: S} using Bppii,
    key kBppbbii{B: S} using Bppbbii,
)
{
    instantiate kBppp;
    asend(server, this, kBppp){SBppp+};
    ssreceive(this, server; opponent){kBppp};

    while i < numShips do
        x := random(boardSizeW);
        y := random(boardSizeH);

        if board[x][y] = 0 then
            board[x][y] := 1;
```

```
            i := i + 1
        endif
    endwhile;

    instantiate kBppt;
    asend(server, this, kBppt){SBppt+};
    instantiate kBppb;
    asend(server, this, kBppb){SBppb+};

    # send board to server and receive true back
    ssend(server, this, board){kBppt};
    ssreceive(this, server, true;){kBppb};

    instantiate kBppii;
    asend(server, this, kBppii){SBppii+};

    instantiate kBppbbii;
    asend(server, this, kBppbbii){SBppbbii+};

    while not declassify(done, {}) do
        ssreceive(this, server; myTurn){kBppb};
        if declassify(myTurn, {}) then
            x := random(boardSizeW);
            y := random(boardSizeH);

            ssend(server, this, declassify(x, {}), declassify(y, {})){kBppii}
        else
            ssreceive(this, server; x, y){kBppii};
            ssend(server, this, true){kBppb}
        endif;

        ssreceive(this, server; result, done, x, y){kBppbbii}
    endwhile
}
```

C.3 test.w

```
[
    declare d as {int{A: B}, bool{A: }}{A: all}
]

A[] :
(
    x{} := 0,
    y{} := 0,
    b{A: } := true,
    b2{A: B} := true,
    l{} := 0,
    h{A: } := 0,
```

```
    h2{B: } := 0,
    key k{} using d,
    t[5][5]{},
)
{
    ### Assignment ###

    x := y;
    x := b;
    l := h;

    l := t[1][h];
    t[1][h] := l;

    if x + y then
        skip
    endif;

    if h < 0 then
        x := y
    endif;

    ### Communication ###

    ssend(x, b){k};
    ssend(h, b){k};
    ssend(h, x){k};

    ssreceive(h; b){k};
    ssreceive(h; b2){k};

    sreceive(h; b){k} andactfor A in
        h := 1
    endactfor;
    sreceive(h; b){k} andactfor B in
        h := 1
    endactfor;

    instantiate h;

    ssreceive(h; b){h};
    ssreceive(h; b){u};

    ### Declassification ###

    l := declassify(h, {});
    l := declassify(h2, {});
    if b2 then
        l := declassify(h, {})
```

```
    endif;

    ### Types in Expressions ###

    h := l + b;
    b := l < b;
    b := l = b;
    b := not l;
    l := random(b);
    l := t[l][b];
    l[3][4] := u;

    skip
}
```

APPENDIX D

Test Results

Running the tests from `test.sml` is done using the command

```
make test
```

from the terminal in the `gwhile_tool` folder.

D.1 Type System

In the test of the type system the following conventions are used:

- `x` and `y` are numeric variables
- `h` and `h2` are numeric variables with the restrictive labels $\{A :\}$ and $\{B :\}$
- `l` is a numeric variable, used in tests together with h
- `t` is a table
- `b` and `b2` are boolean variables, `b` has the label $\{A :\}$ while `b2` has the label $\{A : B\}$
- `d` is a key declaration with the format `declare d as {int{A: B}, bool{A: }}{A: all}`
- `k` is a symmetric key using the key declaration `d`
- `u` is an undefined variable

unless something else is noted the variables are labeled with the empty, or least restrictive, label.

Follow-on errors are not shown except for the communication statements where they are relevant.

A test program, `test.w`, is included in Appendix C.3, which contains all the test cases for the type system, except those leading to fatal errors.

Test Case	Expected Result	OK
`x := y`	No errors occurred	$\surd$
`x := b`	Error in `x := b` : The types int and bool do not match	$\surd$
`l := h`	Error in `l := h` : Label {} of `l` is not a restriction on the label {A: A}	$\surd$
`l := t[1][h]` *	Error in `l := t[1][h]` : Label {} of `l` is not a restriction on the label {A: A}	$\surd$
`t[1][h] := l` *	Error in `t[1][h] := l` : Label {} of `t[1][h]` is not a restriction on the label {A: A}	$\surd$
`if x + y then skip endif`	Error in `if x + y then skip else skip endif` : Type int of expression `x + y` did not match expected bool	$\surd$
`if h < 0 then x := y endif`	Error in `x := y` : Label {} of `x` is not a restriction on the label {A: A}	$\surd$
`ssend(x, b){k}`	No errors occurred	$\surd$
`ssend(h, b){k}`	Error: The label {A: B} is not a restriction on {} ⊔{A: } = {A: A} Error in `ssend(h, b){k}` : Label error	$\surd$
`ssend(h, x){k}`	Error in `ssend(h, x){k}` : Type mismatch between key and fields	$\surd$
`ssreceive(h; b){k}`	No errors occurred	$\surd$
`ssreceive(h; b2){k}`	Error: The label {A: B} is not a restriction on {A: A} ⊔{A: } = {A: A} Error in `ssreceive(h; b2){k}` : Label error	$\surd$
`ssreceive(h; b2){k}` **	Error: The label A: B is not a restriction on {A: A} ⊔{A: B} = {A: A} Error in `ssreceive(h; b2){k}` : Label error	$\surd$
`sreceive(h; b){k} andactfor A in h := 1 endactfor`	No errors occurred	$\surd$

continued on next page

continued from previous page

Test Case	Expected Result	OK
`sreceive(h; b){k} andactfor B in h := 1 endactfor`	Error in `sreceive(h; b){k} andactfor B in h := 1 endactfor` : Principal B is not authenticated	√
`instantiate h`	Error in `instantiate h` : The variable `h` is not a symmetric key	√
`ssreceive(h; b){h}`	Error in `ssreceive(h; b){h}` : The variable `h` is not a symmetric key	√
`ssreceive(h; b){u}`	Error in `ssreceive(h; b){u}` : The key `u` is not defined	√
`declassify(h, {})`	No errors occurred	√
`declassify(h2, {})`	Error: The label {} lub {A: } = {A: A} is not a restriction on {B: } in `declassify(h, {})` Error in `l := declassify(h, {})` : Label {} of `l` is not a restriction on the label {B: B}	√
`if b2 then l := declassify(h, {}) endif`	Error in `l := declassify(h, {})` : Label {} of `l` is not a restriction on the label {A: B, A}	√
`l + b`	Error: Expected types int and int did not match types int of expression `l` or bool of expression `b` in `l + b`	√
`l < b`	Error: Expected types int and int did not match types int of expression `l` or bool of expression `b` in `l < b`	√
`l = b`	Error: Expected types int and int did not match types int of expression `l` or bool of expression `b` in `l = b`	√
`not l`	Error: Type int of expression `l` did not match expected bool in `not l`	√
`random(b)`	Error: Type bool of expression `b` did not match expected int in `random(b)`	√
`t[l][b]`	Error: Expected types int and int did not match types int of expression `l` or bool of expression `b` in `t[l][b]`	√
`l[3][4] := u`	Error: The variable `l` is not a table. Error: Variable `u` is not defined.	√
`key k{} using h`	Fatal Error in `key k{} using h` : The variable `h` is not a key declaration	√

continued on next page

continued from previous page

Test Case	Expected Result	OK
`key k{} using u`	Fatal Error in `key k{} using u` : The key declaration `u` is not defined	√
`dA:  := true`	Fatal Error in `dA:  := true` : The variable `d` is defined more than once	√
`A(A+)-`	Fatal Error in `A(A+)-` : The variable `A+` is not a key declaration	√
`A(u)-`	Fatal Error in `A(u)-` : The key declaration `u` is not defined	√
`declare d as {int{A: B}, bool{A: }}{A: all}`	Fatal Error in `declare d as {int{A: B}, bool{A: }}{A: all}` : The key declaration `d` is defined more than once	√

Table D.1: Test cases and results for type checker

* Tested in accordance with Section 4.2.3

** `d` is declared as `declare d as {int{A: B}, bool{A: B}}{A: all}` to test effects of implicit flow in receive statements.

D.2 TMCA

The test of the TMCA uses the same key declarations and variables as the test of the type system with a few changes and additions.

- `d2` is a key declaration with the same format as `d`
- `k2` is a symmetric key using the key declaration `d2`
- `A+` is a public key using `d`
- `x` has the label $\{A : B\}$

Test Cases			
In A	**In** B	**Expected Result**	**OK**
`ssend(x, b){k}`	`ssreceive(x; b){k}`	No errors occurred	√
`ssend(x, b){k};`		Error: Could not match symmetric receive with key declaration d communicated once from process A	√

continued on next page

continued from previous page

In *A*	**In** *B*	**Expected Result**	**OK**
`ssreceive(x; b){k}`		Error: Could not match symmetric send with key declaration d communicated once from process A	
`ssend(x, b){k}`	`ssreceive(x; b){k2}`	Error: Could not match symmetric receive with key declaration d2 communicated once from process B Error: Could not match symmetric send with key declaration d communicated once from process A	√
`asend(x, b){A+}`	`ssreceive(x; b){k}`	Error: Could not match symmetric receive with key declaration d2 communicated once from process B Error: Could not match asymmetric send with key declaration d communicated once from process A	√
`ssend(x, b){k}`	`while b do ssreceive(x; b){k} endwhile`	No errors occurred	√
`if b then ssend(x, b){k} else ssend(x, b){k} endif`	`while b do ssreceive(x; b){k} endwhile`	No errors occurred	√
`if b then ssend(x, b){k} else ssend(x, b){k} endif`	`if b then ssreceive(x; b){k} endif`	No errors occurred	√
`if b then ssend(x, b){k} else ssend(x, b){k} endif`	`ssreceive(x; b){k}`	No errors occurred	√
`ssend(x, b){k}`	`if b then ssreceive(x; b){k} else ssreceive(x; b){k} endif`	No errors occurred	√

continued on next page

continued from previous page

In A	In B	Expected Result	OK
`while b do ssend(x, b){k} endwhile`	`while b do ssreceive(x; b){k} endwhile; ssreceive(x; b){k}`	No errors occurred	$\surd$
`while b do ssend(x, b){k} endwhile; ssend(x, b){k}`	`while b do ssreceive(x; b){k} endwhile`	No errors occurred	$\surd$
`ssend(x, b){k}`	`while b do ssreceive(x; b){k} endwhile; ssreceive(x; b){k}`	No errors occurred	$\surd$

Table D.2: Test cases and results for TMCA

Bibliography

[Den76] Dorethy E. Denning. A lattice model for secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[DP90] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[HR99] Michael R. Hansen and Hans Rischel. *Introduction to Programming using SML*. Addison-Wesley, Harlow, England, 1999.

[ML97] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[ML98] Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1998.

[ML00] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, October 2000.

[Mye99] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, January 1999.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications : A Formal Introduction*. Wiley, 1992.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, Berlin, 1999.

[Pri00] Matt Pritchard. How to hurt the hackers: The scoop on internet cheating and how you can combat it. *Gamasutra*, 2000. Available from `http://www.gamasutra.com/features/20000724/pritchard_01.htm`.

[Sesff] Peter Sestoft. Moscow ml home page
. `http://www.dina.dk/~sestoft/mosml.html`, 1995ff.

[SMH01] Fred B. Schneider, Greg Morrisett, and Robert Harper. A language-based approach to security. *Lecture Notes in Computer Science*, 2000, 2001.

[TH03] Terkel Tolstrup and Michael R. Hansen. Source code from dtu course 02240 for a simple While parser. `http://imm.dtu.dk/courses/02240/`, 2003.

[VS97] Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In *TAPSOFT '97, Colloqium on Formal Approaches in Software Engineering*, April 1997.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[ZM01] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proceedings of the 2001 IEEE Computer Security Foundations Workshop*, pages 15–23, June 2001.