
Project N°: **FP7-231620**

Project Acronym: **HATS**

Project Title: **Highly Adaptable and Trustworthy Software using Formal Models**

Instrument: **Integrated Project**

Scheme: **Information & Communication Technologies**

**Future and Emerging Technologies**

# Deliverable D4.1
# Report on Security

Due date of deliverable:

Actual submission date:

Revision date:


Start date of the project: **1st March 2009** Duration: **48 months**

Organisation name of lead contractor for this deliverable: **UPM**

Revised version

| **Integrated Project supported by the 7th Framework Programme of the EC** | | |
|---|---|---|
| **Dissemination level** | | |
| PU | Public | ✓ |
| PP | Restricted to other programme participants (including Commission Services) | |
| RE | Restricted to a group specified by the consortium (including Commission Services) | |
| CO | Confidential, only for members of the consortium (including Commission Services) | |

# Executive Summary:
## Report on Security

This document summarizes deliverable D4.1 of project FP7-231620 (HATS), an Integrated Project supported by the 7th Framework Programme of the EC within the FET (Future and Emerging Technologies) scheme. Full information on this project, including the contents of this deliverable, is available online at `http://www.hats-project.eu`.

This deliverable reports on mechanisms which ensure confidentiality policies for long-lived and complex software. It consists of two parts.

The first part of the deliverable reports on the development, and on the evaluation on a common case study in ABS, of three complementary language-based methods (type systems, logics, and dynamic methods) for enforcing confidentiality policies for ABS applications, both at the ABS level and at the bytecode level. The main contributions for this part include:

- provably sound information flow type systems for core fragments of ABS and Java bytecode, and type-preserving compilation results;
- logical methods to verify (classes of) hyperproperties, including a method based on epistemic logic to verify permissive information flow policies;
- a source-to-source transformation that captures the effects of secure multi-execution, and a dynamic method to enforce data processing security policies;
- a method to generate from high-level descriptions protocol implementations in ABS.

The second part of this deliverable reports on the automatic generation of protocol implementations in ABS. The prototype has been successfully evaluated on a large set of protocols.

## List of Authors

Gilles Barthe (UPM)
Dave Clarke (KUL)
Juan Manuel Crespo (UPM)
Mads Dam (KTH)
César Kunz (UPM)
Peeter Laud (IoC)
Gurvan Le Guernic (KTH)
Andreas Lundblad (KTH)
Dimiter Milushev (KUL)
Martin Pettai (IoC)
Exequiel Rivas (UPM)

# Contents

# Chapter 1

# Introduction

Security is a fundamental concern for long-living systems: in order to gain widespread acceptance, evolvable software must not only maintain its core functionality over time; it must also ensure that assets are adequately protected over time. This deliverable reports on mechanisms which ensure confidentiality and integrity policies for long-lived and complex software. We focus on two complementary scenarios:

- passive, honest-but-curious attacker: in this scenario, concurrent computations are executed in a trusted environment, with an honest but curious attacker that aims to infer confidential information from observable states. In more detail, the attacker has no control over communication between the different components of concurrent computations, and cannot retrieve the values exchanged among threads during computations. This scenario typically corresponds to a setting in which an untrusted component executes concomittently with trusted components, on a trusted platform;

- active attacker: in this scenario, the attacker has control over the network, and can intercept, modify, replay or redirect messages. This scenario corresponds to a setting where applications are distributed over a network that is under the control of the adversary, but execute on trusted platforms. Other scenarios that consider execution over untrusted platforms require adopting solutions that are based on advanced cryptographic techniques, such as fully homomorphic encryption, and are out of scope of the HATS project.

We address the first scenario by developing language-based methods for enforcing confidentiality policies for concurrent and object-oriented programs. We address the second challenge by building evolvable implementations of cryptographic protocols using the HATS infrastructure.

## 1.1 Language-based security

While software security traditionally focuses on low-level protection mechanisms such as access control, the popularization of massively distributed systems dramatically increases the number and severity of vulnerabilities at application level. These vulnerabilities may be exploited by malicious software such as viruses, Trojan horses, etc., but also (unintentionally) by buggy software, with disastrous effects.

Language-based security is an emerging technology that provides strong security guarantees for applications. The most distinctive feature of language-based security is that it aims to counter security threats at the application level, with the immediate benefit of countering application-level attacks at the same level at which such attacks arise. Language-based security is attractive to programmers because it allows them to express security policies and enforcement mechanisms at the level of the programming language itself. Specifically, one essential feature of language-based security is tailored to provide strong semantic properties about information flow policies. Such policies track how information is propagated throughout program execution, and are conveniently cast as properties about program executions. The focus on information flow policies, rather than say access control policies, is fundamental for long-lived software, as the guarantees they provide remain meaningful over time. This stands in sharp contrast with access control policies, which

track how information is accessed, and make implicit assumptions about the environment; while acceptable for short-lived and closed systems, implicit assumptions are incompatible with long-lived software, as the assumptions may need to evolve, or may be violated.

There are primarily three approaches to language-based security: type systems, logical methods, and dynamic methods. The approaches are complementary, in the sense that they achieve different trade-offs between conflicting requirements:

- coverage: can the method be used to verify all secure programs, or can it only verify a specific class of secure programs?
- practicality: is the method automated, or does it require significant user input, including annotations, and even proofs?
- performance: does the method incur a significant performance (or memory) penalty, or a limited overhead?
- transparency: does the method require modifying the underlying computational model, or can it be used on legacy platforms?

Additionally, the approaches are complementary in the sense that they are able to cover different language features. This complementarity w.r.t. language features is of particular importance in the context of the HATS project, because of the richness of the ABS language. Specifically, none of the existing afore-mentioned methods has yet been applied to class-based, concurrent, object-oriented languages.

In this task, we have significantly enhanced and developed these complementary lines of enforcement, in particular by integrating challenging ABS features that were not addressed by the state-of-the-art methods at the onset of the project. In the sequel, we briefly review the main characteristics of each method, and provide a summary of our contributions.

### 1.1.1 Type-based methods

Over the last decade, there has been tremendous progress towards the static enforcement of information flow policies. Starting from the seminal work of Volpano and Smith [98], type systems have become a prominent approach for a practical enforcement of information flow policies; this line of work has culminated in the design and implementation of information flow type systems for modern programming languages such as Java. One leading effort towards the development of information-flow aware programming language is Jif [77], which builds upon the decentralized label model and offers a flexible and expressive framework to define information flow policies for Java programs. The rich set of features supported by Jif has proved useful in realistic case studies such as an implementation of mental poker [7], but makes it difficult to prove that the information flow type system is sound.

To address this issue, Banerjee *et al* [11] and Barthe *et al* [18] have developed sound information flow type systems for Java and Java bytecode respectively. These type systems are simpler than Jif since they omit some language features, and do not provide mechanism for declassification. However, both type systems have been formally verified. Furthermore, Barthe *et al* [14] have established a formal relation between the type systems in the form of a type-preservation result, showing that the compiler maps typable (Java) programs to typable (bytecode) programs.

Unfortunately, the type systems are not immediately applicable to ABS, because they are restricted to sequential programs. In fact, developing flexible information flow type systems for concurrent languages is notoriously difficult, because the concurrent execution of two secure programs may be insecure [89]. Hence, existing information flow type systems for concurrent programs impose stringent restrictions; for instance, some systems do not allow threads to write on public memory after a branching statement that tests on secret variables. While sound, such type systems are not applicable. Moreover, no existing type system handles the concurrency model that underlies ABS language.

In Chapter 3, we report on three significant developments. First, we define an information flow type system for a significant fragment of Core ABS. This type system is useful for developers to ensure that their applications are secure. Second, we define an information flow type system for a concurrent fragment of the Java Virtual Machine. This type system allows users to check that deployed ABS applications respect their information flow policies. We also report on preliminary results that connect the two type systems: specifically, we show that concurrent Java programs that are deemed safe by an information flow type system that resembles [11] and handles concurrency, are compiled into bytecode programs that are typable by the bytecode type system. Extending these results for the ABS to JVM compiler developed in the HATS project is left for future work.

### 1.1.2 Logical methods

A central goal of the HATS project is to develop methods for proving properties of ABS programs. An important step towards this goal is reported in Deliverable D2.5 [46], which introduces a program logic that can be used to reason about functional behavior of ABS programs. In this task, we explore the possibility of developing logic-based methods, akin to the ABS program logic, to reason about confidentiality of applications.

It is folklore in the language-based security community that information flow policies are not safety properties, and thus cannot be formalized straightforwardly in conventional program logics, such as the ABS program logic, and Hoare logic. One can however prove that programs are information flow secure using an encoding of information flow policies in an extension of Hoare logic with purpose-specific axioms [3]. Yet, this approach is impractical, because the axioms are purpose specific; thus, its practical realization would require that the ABS tool suite is modified for each specific security policy. In a more recent work, Joshi and Leino [61] provide a characterization of non-interference using weakest precondition calculi. Their characterization is restricted to imperative languages—one noticeable feature is that their approach handles non-deterministic constructs—and it is not clear how their approach extends to the ABS language.

A better alternative is to reduce non-interference of a program $P$ to a property about executions of another program $\hat{P}$ constructed from $P$. Pottier [80] provides an early example of this approach for the pi-calculus, where the non-interference of two processes $P_1$ and $P_2$ is reduced to a property about a single process $P$ that captures the behaviours of $P_1$ and $P_2$ while keeping track of their shared sub-processes. The process $P$ is written in an extension of the pi-calculus and allows for a simple proof of non-interference using standard subject reduction techniques. In a similar vein, Barthe *et al* [13] and Darvas *et al* [43] show how Pottier's approach, which they coin self-composition, provide sound and complete methods for non-interference.

The idea of self-composition has been explored further in a series of recent works. Terauchi and Aiken [93] have proposed a type-directed transformation that performs self-composition selectively, in order to obtain verification tasks that are more amenable to formal verification. Within Task 4.3, we have recently brought significant improvements to their method; the results are published in [16] and will be reported in Deliverable 4.3. In addition, Terauchi and Aiken introduce the class of 2-safety properties, which can be reduced to safety properties by composing the program with itself, and show that non-interference is an instance of a 2-safety property. Subsequently, Clarkson and Schneider [36] have generalized this work to consider hyperproperties, that cover both liveness and safety, and that generalize 2-safety to $n$-safety. A more detailed account of this work is provided in [17], available online.

Within this task, we have developed logic-based methods for information flow. Specifically, we have further extended the foundations of self-composition, and showed how the resulting approach can be applied to verify the TaxRecord case study. Further, we have studied how approaches based on symbolic evaluation help improving the practicality of self-composition techniques; this is specially significant for the HATS project, because the ABS program logic is based on symbolic evaluation. Finally, we have been exploring more expressive logics, and in particular epistemic logics, that allow to model adversary knowledge, and provide an adequate framework for reasoning about information flow policies that go beyond non-interference, and accomodate declassification policies along the "what", "where", and "when" dimensions.

### 1.1.3 Dynamic methods

Runtime verification of properties, such as safety properties and to some extent liveness properties, is an old approach that aims at detecting errors in software during execution. It roughly consists in monitoring some sub-properties of the property to enforce during the execution of a program. The runtime aspect of dynamic methods offers some key advantages among which:

- access to the run time values in the stack and heap, which can allow to increase the precision of the analysis by easily allowing to reason about those values;
- a reduced scope for the enforcement of the property which can allow to ignore some parts of the program, as dynamic methods have to ensure that only the execution under monitoring has the desired property, not all the executions of the program monitored;
- the possibility/flexibility to "safely" run the "good" executions of a program even if some of its executions are "bad".

Naturally, the main and most common drawback of dynamic methods is the introduction (often limited) of overhead at runtime.

Even if the runtime verification of trace properties (whose truth value can be decided by observing only the trace for which a conclusion as to be given) is an old approach, the runtime verification of information flow (and not access control) related confidentiality and integrity properties is more recent and more difficult. The reason is that the truth value of information flow related properties for a given execution does not depend only on the trace of that execution, but also on the traces of some (usually not all) other traces of the program which, of course, are not available at runtime and have to be approximated.

Among these dynamic enforcement techniques, a novel approach coined Secure Multi-Execution (SME) has emerged. SME has appealing theoretical properties: it preserves the behaviour of non-interfering programs while preventing ilicit flows from insecure ones. This is achieved by running concurrently one copy of the program per security level and by modifying the semantics of input/output statements in a way that the insecure flows are forbidden and that proper synchronisation between threads is established.

Within this task, we have conducted a survey of dynamic methods for the verification of information flow related security policies and developed an alternative to secure multi-execution, that achieves similar effects without requiring to modify the underlying runtime infrastructure (Sect. 5.1). The alternative is based on a source-to-source transformation that is easy to implement and shown equivalent to SME in terms of the induced security.

In a reversed approach, we have studied how to inline monitoring code relying on an existing information flow related tracking mechanism in order to enforce application programming interface (API) usage policies with a finer level of precision than is usually possible (Sect. 5.2).

## 1.2 Automated generation of cryptographic protocols

Cryptography plays an essential role in the development of secure software solutions; yet it has proved extremely difficult to design correct cryptographic implementations and to enforce their correct usage: the history of cryptography is fraught with examples of widely used yet insecure primitives and protocols. Moreover, the difficulty in achieving correct cryptographic implementations is exacerbated further in long-lived software, as security must be guaranteed alongside with evolvability and variability: for instance, evolvable software is subject to roll-back attacks, where malicious attackers are able to trigger otherwise secure systems to switch to older, less secure implementations with known vulnerabilities—the widely used TLS protocol is vulnerable to one such famous attack. It is therefore a pressing challenge to develop cryptographic implementations that remain secure in long-lived systems.

We address the second challenge by building evolvable implementations cryptographic protocols using the HATS infrastructure. Specifically, we implement a domain-specific compiler that automatically gener-

ates ABS implementations from high-level specification of cryptographic protocols. The compiler has been successfully applied to many protocols from the Clark-Jacob library.

## 1.3 List of papers comprising Deliverable D4.1

This section lists all the papers that comprise this deliverable, indicates where they were published, and explains how each paper is related to the main text of this deliverable. As requested by the reviewers, the papers are not directly attached to Deliverable D4.1, but are made available on the HATS web site at the following url: `http://www.hats-project.eu/sites/default/files/D4.1`. Direct links are also provided for each paper listed below.

**Paper 1: Securing the Future — an Information Flow Analysis of a Distributed OO Language**

This paper [79] develops an information flow type system for a slightly modified version of the object-oriented fragment of Core ABS.

This paper was written by Martin Pettai and Peeter Laud and is to appear in the proceedings of International Conference on Current Trends in Theory and Practice of Computer Science 2012.

Download Paper 1.

**Paper 2: A Certified Lightweight Non-Interference Java Bytecode Verifier**

This paper [19] develops an information flow type system that covers a significant subset of (sequential) JVM programs including objects, arrays, methods, and exceptions. Since JVM poses a natrual target for compiling ABS programs, it is therefore important to devise methods that are able to detect illegal information flows in JVM programs.

This paper was written by Gilles Barthe, David Pichardie, and Tamara Rezk and is accpeted for publication in the journal Mathematical Structures in Computer Science.

Download Paper 2.

**Paper 3: Security of Multithreaded Programs By Compilation**

This paper [20] develops a modular method for extending an information flow type system for a sequential low-level language into an information flow type system for a concurrent low-level language.

This paper was written by Gilles Barthe, Tamara Rezk Alejandro Russo and Andrei Sabelfeld and it was published in ACM Transactions on Information and System Security.

Download Paper 3.

**Paper 4: Static enforcement of information flow policies for a concurrent JVM-like language**

This paper [21] develops an information flow type system for JAVA bytecode and formalizes a result linking a type system for high level JAVA like language with the low level type system presented.

This paper was written by Gilles Barthe and Exquiel Rivas and it was published in the proceeedings of Trustworthy Global Computing 2011.

Download Paper 4.

**Paper 5: Secure Information Flow by Self-Composition**

This paper [17] presents a method to reduce the task of establishing non-interference of programs (and any 2-safety propery in general) to a program verification task.

This paper was written by Gilles Barthe, Pedro D'Argenio, and Tamara Rezk and it was published in the journal Mathematical Structures in Computer Science.

Download Paper 5.

**Paper 6: Towards Incrementalization of Holistic Hyperproperties**

This paper [76] defines the notions of holistic and incremental hyperproperties and present a method, coined incrementalization, to convert holistic specifications into incremental ones.

This paper was written by Dimiter Milushev and Dave Clarke and it is to appear in the proceedings of First Conference on Principles of Security and Trust.

Download Paper 6.

**Paper 7: Noninterference via Symbolic Execution**

This paper [75] proposes a novel, alternative approach to non-interference utilizing symbolic execution in combination with ideas from program logics in an attempt to increase the precision of analyses and automate noninterference testing.

This paper was written by Dimiter Milushev, Wim Beck, and Dave Clarke and is currently under review.

Download Paper 7.

**Paper 8: Epistemic Temporal Logic for Information Flow Security**

This paper [9] presents a computational model and an epistemic temporal logic used to reason about knowledge acquired by observing program outputs. This approach is shown to elegantly capture standard notions of noninterference and declassification in the literature as well as information flow properties where sensitive and public data intermingle in delicate ways.

This paper was written by Mussard Balliu, Mads Dam and Gurvan Le Guernic and was published in the proceedings of the Workshop in Programming Languages and Analysis for Security 2011.

Download Paper 8.

**Paper 9: ENCOVER: Symbolic Exploration for Information Flow Security**

This paper [10] addresses the problem of automatic program verification for information flow policies (expressed as epistemic formulas as presented in [9]) by means of symbolic execution and satisfiability modulo theory. These ideas are realised through the implementation of a tool.

This paper was written by Mussard Balliu, Mads Dam and Gurvan Le Guernic. Unpublished draft.

Download Paper 9.

**Paper 10: Introductory Survey to Dynamic Information Flow Security**

This paper [66] introduces the main notions related to information flow security and reviews the main concepts followed in order to dynamically enforce information flow policies.

This paper was written by Gurvan Le Guernic. Unpublished draft.

Download Paper 10.

**Paper 11: Secure multi-execution through static program transformation**

This paper [15] presents a static transformation technique that achieves the effect of Secure Multi-execution, a dynamic non-interference technique, hence avoiding the burden of reimplementong the underlying computational infraestructure, while retaining its appealing theoretical properties.

This paper was written by Gilles Barthe, Juan Manuel Crespo, Dominique Devriese, Frank Piessens, Exequiel Rivas and it is currently under review.

Download Paper 11.

# Chapter 2

# Case study

In order to assess the security analysis methods proposed in HATS, we have devised a case study with several variants. The case study concerns a hypothetical tax-paying scenario where several entities, specified in ABS, are involved in computing and verifying the amount of taxes to be payed.

We will regard certain inputs to this computation as sensitive (have the security level $H$). We test how well our methods can separate sensitive from non-sensitive, and whether they can be used to establish the security of ABS programs with complex control structure.

The different variants aim at showing different features and how and if each method can cope with them, e.g. looping on a guard that depends on a secret value.

## 2.1 Basic notions

As customary in works on secrecy and confidentiality, we assume a security lattice containing the values $L$ and $H$, standing for low (or public) and high (or secret) respectively.

In the following set of examples, the main goal is to establish that the programs conform to a set of basic security policies. For example, it is desirable that the checker, that represents the entity that will verify that the taxpayer has properly payed the taxes, does not gain information regarding its salary or the amount of money it has donated to charity. Similarly, the entity that is in charge of computing the total amount donated to charity, while granted access to the amount donated by individuals, should not obtain information regarding their salaries or whether an individual has made effective the payment of its taxes.

The way in which such policies are specified depends heavily on the underlying method used to establish the property. For example, such specifications boil down to a simple mapping of objects and fields to security levels when using a security type system, while when using logic based methods they generally amount to a conjunction of equalities as pre and postconditions for procedures. The formalised policies will be presented along with the application of each method for selected snippets extracted from the examples.

## 2.2 Baseline example

The source code of our example is given below.

```
/*
 * -- TaxRecord.abs --
 *
 * This file is part of TaxRecord. TaxRecord is the case study for task 4.1 of
 * the \HATS project. It simulates a simplified tax paying process.
 *
 */

module TaxRecord;

interface TaxServer4charity {
```

```
        Int getCharity();
}

interface TaxServer extends TaxServer4charity {
        Unit registerTaxRecord(TaxRecord tr);
}

class TaxServerImpl implements TaxServer {
        List<TaxRecord> taxRecords = Nil;

        Unit registerTaxRecord(TaxRecord tr) {
                taxRecords = appendright(taxRecords, tr);
        }

        Int getCharity() {
                Int totalAmount = 0;
                Int nbRecords = length(taxRecords);
                Int i = 0;
                while ( i < nbRecords ) {
                        TaxRecord tr = nth(taxRecords, i);
                        Fut<Int> fut = tr!getCharity();
                        Int amount = fut.get;
                        totalAmount = totalAmount + amount;
                        i = i + 1;
                }
                return totalAmount;
        }
}

interface TaxRecord4taxPayer {
        Int getTaxes();
        Int getAmountPayed();
        Int payTaxes(Int charity, Int amount);
}

interface TaxRecord4taxChecker {
        Int verifyPayment();
        Unit freeze();
}

interface TaxRecord extends TaxRecord4taxPayer, TaxRecord4taxChecker {
        Unit registerChecker(TaxChecker checker);
        Int getCharity();
}

class TaxRecordImpl(Int init_salary) implements TaxRecord {
        Int salary = init_salary;
        TaxChecker checker = null;
        Int charity = 0;
        Int amountPayed = 0;
        Bool frozen = False;

        Int computeTaxes() {
                return salary/10;
        }

        Unit registerChecker(TaxChecker checker) {
                this.checker = checker;
        }

        Int setCharity(Int amount) {
                if ( ~this.frozen ) {
                        if (amount < 0) amount = 0;
                        this.charity = amount;
                }
                return this.charity;
        }

        Int getAmountPayed() {
                return this.amountPayed;
        }
```

```
        Int getTaxes() {
                return this.computeTaxes();
        }

        Int getTaxBalance() {
                Int taxAmount = this.computeTaxes();
                Int taxBalance = this.amountPayed - (taxAmount + this.charity);
                return taxBalance;
        }

        Int payTaxes(Int charity, Int amount) {
                if ( ~this.frozen ) {
                        this.setCharity(charity);
                        this.amountPayed = this.amountPayed + amount;
                        if (checker != null) checker!checkTaxes(this);
                }
                Int taxBalance = this.getTaxBalance();
                return taxBalance;
        }

        Int verifyPayment() {
                Int taxBalance = this.getTaxBalance();
                if (taxBalance < 0) { taxBalance = -1; }
                return taxBalance;
        }

        Unit freeze() {
                this.frozen = True;
        }

        Int getCharity() {
                await this.frozen;
                return this.charity;
        }
}

interface TaxPayer {
        Int getTaxes();
        Int getAmountPayed();
        Int payTaxes(Int charity, Int amount);
        Unit payTaxesInFull(Int charity);
}

class TaxPayerImpl(TaxRecord4taxPayer taxRecord) implements TaxPayer {

        Int getTaxes() {
                Fut<Int> fut = taxRecord!getTaxes();
                Int taxes = fut.get;
                return taxes;
        }

        Int getAmountPayed() {
                Fut<Int> fut = taxRecord!getAmountPayed();
                Int amountPayed = fut.get;
                return amountPayed;
        }

        Int payTaxes(Int charity, Int amount) {
                Fut<Int> fut = taxRecord!payTaxes(charity, amount);
                Int taxBalance = fut.get;
                return taxBalance;
        }

        Unit payTaxesInFull(Int charity) {
                Int taxes = this.getTaxes();
                Int amountPayed = this.getAmountPayed();
                Int balance = (charity + taxes) - amountPayed;
                this.payTaxes(charity, balance);
        }
}

interface TaxChecker {
```

```
        Int checkTaxes(TaxRecord4taxChecker taxRecord);
}

class TaxCheckerImpl() implements TaxChecker {

        Int checkTaxes(TaxRecord4taxChecker taxRecord) {
                Fut<Int> fut = taxRecord!verifyPayment();
                Int balance = fut.get;
                if ( balance >= 0 ) {
                        taxRecord!freeze();
                }
                return balance;
        }
}

interface Charity {
        Int getCharity();
}

class CharityImpl(TaxServer4charity server) implements Charity {

        Int getCharity() {
                Fut<Int> fut = server!getCharity();
                Int amount = fut.get;
                return amount;
        }
}

{
        TaxServer server = new cog TaxServerImpl();
        TaxChecker checker = new cog TaxCheckerImpl();

        TaxRecord tr1 = new cog TaxRecordImpl(50000);
        server!registerTaxRecord(tr1);
        tr1!registerChecker(checker);
        TaxPayer alice = new cog TaxPayerImpl(tr1);

        TaxRecord tr2 = new cog TaxRecordImpl(75000);
        server!registerTaxRecord(tr2);
        tr2!registerChecker(checker);
        TaxPayer bob = new cog TaxPayerImpl(tr2);

        Charity charity = new cog CharityImpl(server);
        charity!getCharity();

        Fut<Int> fut = alice!payTaxes(50, 100);
        bob!payTaxes(10,10000);
        fut.get; alice!payTaxesInFull(15);
}
```

In this example, a taxpayer class is defined. Each object in this class is associated with a tax record object that peforms the computation of payable taxes, and keeps a record of how much of the taxes have already been payed. A taxpayer can pay his/her taxes in full (at once) or in several parts. Additionally, a taxpayer can direct a certain amount of his/her taxes towards a charity.

A tax checker object has a single method that verifies whether all the taxes associated with a record have been payed; this method is queried each time a payment is made. A charity object may query how much money has been directed towards it by all taxpayers in the system. The answer is given by a tax server object that stores all tax records and can add up the respective amounts. The amount donated to a charity can only be determined after the tax record has been frozen. The freezing is done by the tax checker object, after all taxes have been payed.

Hence we have a rather complex control flow, with many different objects communicating. The sequence diagram of created objects, tasks, asynchronous calls and returned futures is depicted in Fig. 2.1. This diagram has been created using the ABS plugin for Eclipse.

The Java code generated from this ABS model relies heavily on a ABS specific library whose analysis is problematic for some of the tools presented in this chapter (mainly due to the implementation of communication between cogs). Therefore, Java implementations of this case study have been directly im-

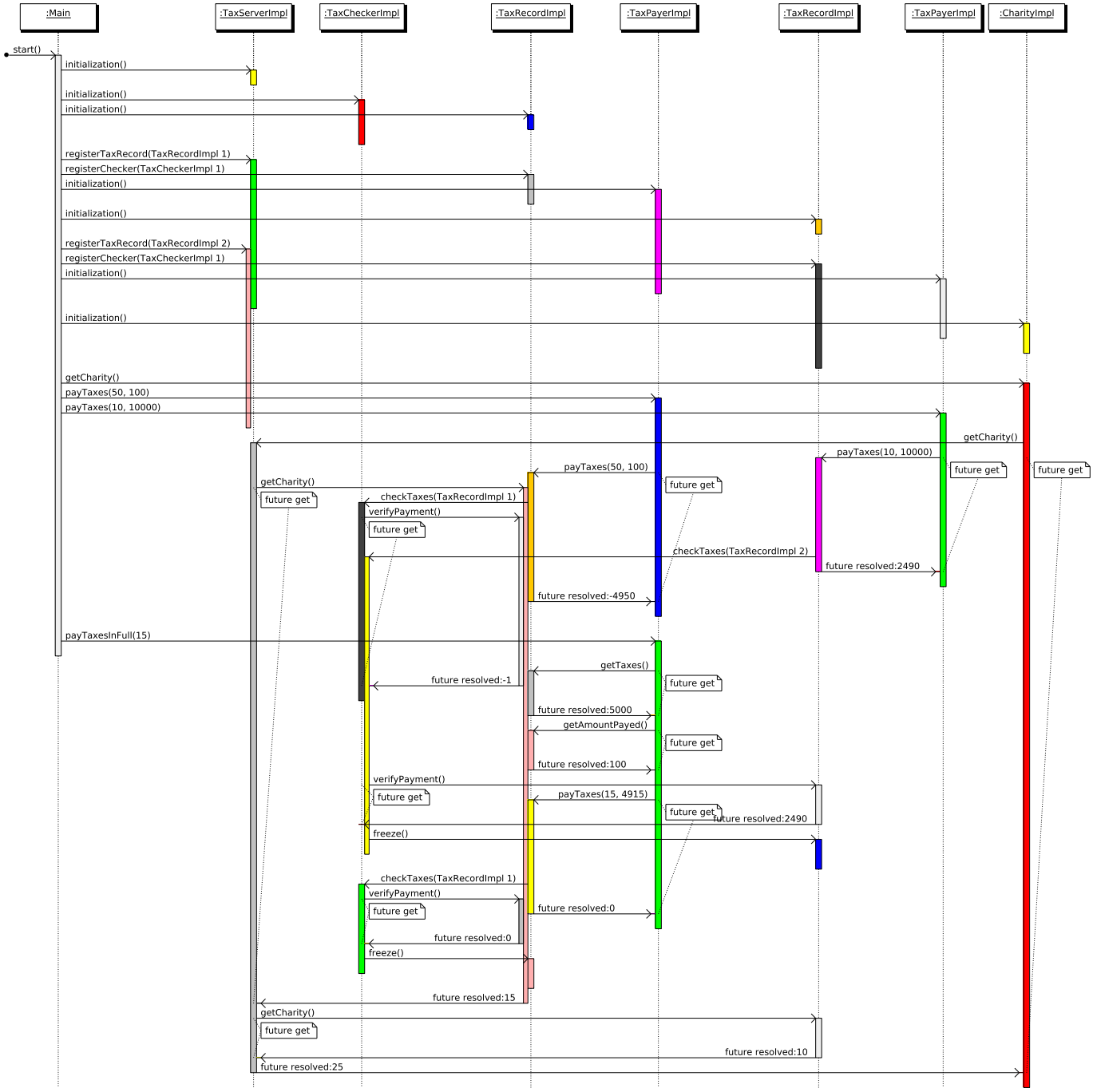


Figure 2.1: Sequence diagram of the case study

plemented. One of them is monothreaded (for tools that do not handle multithreading yet), the other one is multithreaded.

## 2.3 Example with loops

In order to experiment with loops and their effect to the information flow analysis (through possible non-termination), we have made the tax rate calculation more complex. We have replaced the method TaxRecordImpl::computeTaxes with the following code.

```
Int const_SLICE_WIDTH = 2000;

Int computeTaxes() {
        Int taxes = 0;
        Int untaxedSalary = this.salary;
        Int sliceNb = 0;
        while ( untaxedSalary > 0 ) {
                Int taxedSlice = 0;
                if (untaxedSalary < const_SLICE_WIDTH) { taxedSlice = untaxedSalary; }
                else { taxedSlice = const_SLICE_WIDTH; }
                taxes = taxes + ((taxedSlice * (sliceNb * 5)) / 100);
                untaxedSalary = untaxedSalary - taxedSlice;
                sliceNb = sliceNb + 1;
        }
        return taxes;
}
```

# Chapter 3

# Type-based methods

Types are syntactically defined, automatically decidable assertions about program behaviour. This chapter explores type-based methods for ensuring secure information flow. Compared to other analysis methods, types have the benefit of being intuitive, automatic and scalable. This makes them applicable to real-world applications, such as considered in HATS.

The first well-known type-based analysis for secure information flow in a simple imperative language (the WHILE-language) was proposed by Volpano et al. [97]. Later, their analysis has been extended in many different directions, including treated language constructs, and the versatility of the tools for defining information flow properties.

Type-based methods usually partition the set of program variables into low (public) and high (private). These define the security types for variables. Type rules are given to determine security types for other program parts, e.g. expressions and method definitions. Only those program parts that do not contain leaks (but not necessarily all of those) should be typable in the type system. Also an operational semantics is given for the language. Then a certain non-interference property is proved, stating that if a program is well typed in the type system then executing it according to the operational semantics, its public outputs do not depend on the private inputs. Private inputs may be, for example, the initial values of the private variables, and public outputs may be the final values of the public variables. Thus the type system can be used to statically check if a program has secure information flow.

The analysis of Volpano et al. [97] gives types to variables (locations), expressions and commands. Each type is basically just a security class. The type of an expression is the upper bound of the types of all of its subexpressions; the type of a variable (as expression) equals the type of that variable as location. The type of a command is the upperbound of the types of all expressions that may have affected whether the control flow of this program may reach that program. The type system enforces a kind of a "no write down" policy:

- An expression may only be stored in a variable with equal or higher type.
- A command may only write into variables of equal or higher type.

The structuredness of the WHILE-language programs plays a big role in the precision of the described type system. It allows to precisely identify which expressions (in the form of guards in `if`- and `while`-statements) determine whether the control flow may reach a certain program point. On the other hand, type systems for secure information flow in unstructured languages are much more complicated [12].

As mentioned above, the type of a program certifies certain claims about its information flow. Anyone seeing both the program and its typing can easily check the correctness of that typing and thereby be convinced of the security of the information flow. Type checking is generally a simpler procedure than type inference. But a typing certifies the properties only for a certain program in a certain programming language. If that program is compiled into a different language, the assurance provided by the typing is lost (or present only if the original program and a correctness proof for compilation are also given). Type-preserving compilation [88, 67] allows the inferred type for the program in the source language to be

"compiled" to a type for the program in the target language. This avoids the need for performing type inference on the target language.

**State of the art** Volpano et. al [97] gave their type system only for the basic WHILE-language. Real languages extend it with various constructions. Each construction brings its own peculiarities in information flow. Also, the constructions can interact with each other in unexpected ways.

Also, the type system of [97] only certifies programs for *termination-insensitive* information flow. Termination is one of the many *covert channels* that can leak information. It is well-known that for parallel programs with shared, mutable state, termination and timing behaviour of threads can be used for inter-thread communication, which has obvious information-flow consequences. For concurrent languages, even nontermination only can be used to leak an unbounded number of bits.

Secure information flow in a language with parallel threads operating on a shared state was considered in [89]. Parallel threads require some kind of scheduling. One way is to use a purely nondeterministic scheduler, like in [89], where any thread can be chosen for the next step. In this way, one gets *possibilistic* non-interference — if a particular behaviour is possible for one set of high inputs, it is also possible for all other sets of high inputs. Another way is to use a probabilistic scheduler and use a *probabilistic* notion of non-interference. In this setting, the number of scheduling points can influence the probability of choosing a certain thread. This can lead to leaks if the number depends on high variables. One way of preventing these leaks is to require the same number of scheduling points in each branch of a high conditional in a thread that can later make a low side effect. This is done in [90]. Another way is to partition the set of threads into low threads (these can make low side effects) and high threads (these cannot) and to forbid scheduling another low thread if the scheduling point occurs in a low thread in high context. This approach is used in [84].

If a probabilistic scheduler is used then on each scheduling point the scheduler must choose the next thread to execute. Each thread has a probability that it will be chosen. This probability distribution may be different on every scheduling event. Relative probabilities for low threads (threads that may still make a low side effect) must not depend on high state, otherwise leaks through the scheduler can occur. A special case of this condition is the uniform scheduler, which is considered in several papers, including [90].

In [89], high while cycles needed to be forbidden, to be able to prove noninterference. In [90], this restriction was relieved, so that high while cycles are allowed but no low assignment can occur after a high cycle in the same thread. This allows threads without low side effects to use unrestricted high cycles. The body of a high cycle must still contain at least one scheduling point, i.e. the whole cycle cannot be executed atomically because it might not terminate and thus the high cycle might block low side effects in other threads, leading to leaks.

In concurrent languages, threads often communicate using synchronization primitives instead of shared state. Noninterference for a concurrent language with synchronization primitives was considered in [86]. High synchronization was not allowed in this language.

Compared to concurrency, object-orientedness (structured data and procedures / methods) brings with it relatively less issues for tracking flows of information. Ever since the JFlow language [78] was proposed, the basic scheme is to annotate each field of each object with a security class. This means that the information flows affecting which data can be stored in which fields are accounted in a context- and flow-insensitive manner. Similarly, the types of methods contain the security classes of their arguments and return values, as well as the lower bound of the side effects they may perform.

**HATS contributions** We have proposed a basic type system for checking the security of the information flow in the object-oriented fragment of the ABS language. While certain features of the language have simplified the type system, the concurrency model has still presented significant challenges and has actually forced us to change the model somewhat. We have also proposed information flow type systems for a concurrent fragment of the Java Virtual Machine, and proved that Java programs which are provably secure by information flow type systems are compiled into Java bytecode programs that are accepted by information flow-aware bytecode verifiers.

In the future work, we plan to study type-preserving compilation from ABS to lower-level languages. This would allow us to relate the different type systems we have proposed in HATS for languages at different levels.

## 3.1 Information flow type system for Core ABS

In [79], we have developed an information flow type system for a slightly modified version of the object-oriented fragment of Core ABS (HATS deliverable D1.1a [45], Chapter 5). Core ABS has three levels of concurrency: intra-object tasks with shared memory, objects inside a single concurrent object group (cog), and different cogs running in parallel. This makes its information flow analysis more complex than for languages with only one level of concurrency. The first two levels collapse into one because no two tasks in the same cog can run at the same time regardless of whether the tasks are in the same object or not. The difference between the two levels is that tasks inside one object can have shared memory but tasks in different objects cannot. The lack of shared state between different objects and cogs simplifies the type system by removing a possible channel for information leaks. However, the information between the tasks of the same object can still flow through the fields of that object. This forces us to be careful with the places for possible context switches inside a cog. Additionally, there is a limited form of inter-cog information flow through the futures in await-statements, where one task suspends until a different one has finished its execution.

The operational semantics does not distinguish the three levels of concurrency. It allows to choose nondeterministically any task (in any cog) that can make a step. Inside a single cog, it achieves cooperative scheduling by using a lock (called the high lock) in each cog. Each task grabs the high lock of its cog before it makes any steps. The scheduler of a cog can only switch to a different task at the scheduling points where a task releases the lock and then tries to grab it again. The cooperative scheduling allows one task to block others from running; this is yet another side channel.

On the other hand, the parallelism of different cogs is achieved by allowing any cog to be chosen for the next step (because tasks in one cog are not restricted by the locks of other cogs), regardless of which cog executed the previous step or whether it arrived at a scheduling point or not. This can be compared to the scheduling of parallel threads where the scheduler can switch to a different thread after each step.

To avoid the restriction of needing to have the same number of scheduling points in both branches of a conditional statement, we distinguish high and low scheduling points depending on whether the scheduling point appears in high context (such as in a conditional with a high guard) or low context. On a high scheduling point, the scheduler can only switch to a high task or to the one low task that most recently suspended. It cannot switch to another low task. This is achieved in the operational semantics using a second lock (the low lock) in each cog. A low task needs the low lock to execute and the low lock is released only on low scheduling points. Thus in the low view of a cog (where we ignore high tasks, which are invisible to the low observer), scheduling points only occur in low context.

Core ABS also has loops. We allow high while loops but only in the context of a high task, because the termination of a low task is low-observable and low side effects must not occur after a high loop in the same task. We also require a high while loop to have at least one scheduling point per iteration (this is added automatically by the operational semantics), so that a nonterminating high loop cannot block low side effects in the task that currently holds the low lock.

In addition to while loops, Core ABS also has await loops, which are used for synchronization. The nondeterministic operational semantics of an await loop is equivalent to a while loop polling the variable (e.g. the future) where the awaited task writes its return value when it terminates, although the actual implementation (where some of the nondeterminism is resolved) might be more efficient. The difference with using a variable for synchronization is that the two tasks may be in different objects or different cogs and thus do not have shared memory but the future can be considered to contain a temporary variable that can be used for one-way communication: one task can write to it and another (or several other) tasks can read it. Thus the restrictions on await loops are the same as for while loops: a high await loop (waiting for

a high task that is not guaranteed to terminate) can only occur in a high task and it must suspend after each polling iteration.

The operational semantics also distinguishes a special kind of tasks, called high-low tasks. These are high tasks that are guaranteed (can be proved) to terminate. An await loop polling for such a task is also guaranteed to terminate and thus is allowed in low tasks. The operational semantics also allows an implicit high scheduling point to be created before any low step if certain conditions are filled. Thus there can be considered to be at least one high scheduling point between any two low side effects but these scheduling points are invisible to the low observer (the actual implementation might, for efficiency, choose the already running task most of the time at these scheduling points).

We consider termination-sensitive noninterference. Thus leaks through nontermination are avoided but timing leaks can still occur. It might be possible to avoid timing leaks if we would move every high-context block in a low task into a separate high task (this would not change the low-observable semantics) but we do not consider this in [79].

### 3.1.1 Types

The types in the type system are the following:

$$
\begin{aligned}
T &::= \mathsf{Int}_l \mid C_l \mid \mathsf{Fut}^{\ell}_{l}(T) \mid \mathsf{Guard}^{\ell}_{l} \mid \mathsf{Exp}^{l}(T) \mid \mathsf{Cmd}^{l} \mid \mathsf{Cmd}^{l}(T) \mid (l_0, \overline{T}) \xrightarrow{l[,i]} \mathsf{Cmd}^{l_1}(T) \\
l &::= L \mid H \\
\ell &::= l \mid i \\
i &::= 0 \mid 1 \mid \ldots
\end{aligned}
$$

Thus we can have integers, objects of class $C$, futures, guards, possibly non-terminating expressions, commands, commands (method bodies) returning a value, and methods. Here the subscript represents the security level of the value. For integers and objects, this corresponds to the upper bound on the security levels of the inputs that may have affected this value. For futures and guards, this is the upper bound on the (control flow) information that may affect which task this future is referring to. The security level on top of the arrow of the method type corresponds to the minimum level of side effects this method is allowed to perform. If this is high, then the side effects of this method do not affect the low part of the computation. The level $l_0$ in the method type denotes the security level of the receiver of the method call (i.e., this-argument). The superscripts on the types are the upper bound on information that may affect whether this future, guard, expression, or command eventually returns a value or terminates. If this information is high, then the effects of any computation that follows are high, too.

We also allow an integer $i$ to be added to the security level of the context. This is used to guarantee termination for high-low tasks. For high-low tasks, while cycles are forbidden but cycles could still occur through cycles in the await graph and to disallow this, each of these methods has a positive integer $i > 0$ and can only await after a task with a smaller integer. This makes the await graph of high-low tasks acyclic. Here is the type rule that handles this restriction:

$$
\frac{\gamma, l, i \vdash e : \mathsf{Guard}^{i_1}_{l} \qquad i_1 < i}{\gamma, l, i \vdash \mathsf{await}_l(e) : \mathsf{Cmd}^{L}} \ (\mathsf{Await}_2)
$$

Here $i$ is the integer corresponding to the current high-low task and $i_1$ is the integer corresponding to the awaited task. The type of a high-low method has the form $(l, \overline{T}) \xrightarrow{H,i} \mathsf{Cmd}^{L}(T_1)$ and the corresponding future has type $\mathsf{Fut}^{i}_{H}(T_1)$.

To guarantee that a low side effect does not follow a loop in the same task, we use the following type rules:

$$
\frac{\gamma, l \vdash s_1 : \mathsf{Cmd}^{l_1} \qquad \gamma, l \vee l_1 \vdash s_2 : \mathsf{Cmd}^{l_2}}{\gamma, l \vdash s_1; s_2 : \mathsf{Cmd}^{l_1 \vee l_2}} \ (\mathsf{Seq}_1)
$$

$$\frac{\gamma, l \vdash s_1 : \mathsf{Cmd}^{l_1} \qquad \gamma, l \vee l_1 \vdash s_2 : \mathsf{Cmd}^{l_2}(T)}{\gamma, l \vdash s_1; s_2 : \mathsf{Cmd}^{l_1 \vee l_2}(T)} \ (\mathsf{Seq}_2)$$

If the statement $s_1$ contains a high loop then it has the type $\mathsf{Cmd}^H$ (thus $l$ is $H$ and the rules force the statement $s_2$ that follows $s_1$ to be executed in high context ($l \vee l_1$ is high)).

### 3.1.2 Modifications to semantics

We made some modifications to the operational semantics in deliverable D1.1a [45], to remove some possible leaks.

As mentioned above, we have two locks (or in general: the same number of locks as there are security classes) per cog instead of one. Thus the run-time configurations are as follows:

$$P ::= b[n_1, n_2] \mid o[b, C, \sigma] \mid n\,\langle b, o, \sigma, s\rangle \mid P \parallel P$$

The objects and tasks are represented as in D1.1a, only the representation of cogs has changed. Each cog $b$ now has two locks: the high lock (owned by task $n_2$) corresponds to the original lock in D1.1a and the low lock (owned by task $n_1$) is added.

The operational semantics of some statements ($\mathsf{suspend}_l$, $\mathsf{await}_l\ g$, $e.\mathsf{get}_l$, $\mathsf{while}_l\ (e)\ s$, $e!_l m(\overline{e})$) now depends on the security context $l$ (high or low) and thus they are annotated by $l$. These annotations are added during type checking, not by the programmer. The annotated commands can introduce scheduling points and the annotation is used to determine whether a high or a low suspend is used. The difference between a high and a low suspend is seen in the following reduction rules:

$$\frac{}{n\,\langle b, o, \sigma, \mathsf{suspend}_l; s\rangle \rightsquigarrow n\,\langle b, o, \sigma, \mathsf{release}_l; \mathsf{grab}_l; s\rangle} \ (\mathsf{suspend})$$

$$\frac{}{b[\bot, \bot] \parallel n\,\langle b, o, \sigma, \mathsf{grab}_L; s\rangle \rightsquigarrow b[n, n] \parallel n\,\langle b, o, \sigma, s\rangle} \ (\mathsf{grab_L})$$

$$\frac{}{b[n', \bot] \parallel n\,\langle b, o, \sigma, \mathsf{grab}_H; s\rangle \rightsquigarrow b[n', n] \parallel n\,\langle b, o, \sigma, s\rangle} \ (\mathsf{grab_H})$$

$$\frac{}{b[n, n] \parallel n\,\langle b, o, \sigma, \mathsf{release}_L; s\rangle \rightsquigarrow b[\bot, \bot] \parallel n\,\langle b, o, \sigma, s\rangle} \ (\mathsf{release_L})$$

$$\frac{}{b[n', n] \parallel n\,\langle b, o, \sigma, \mathsf{release}_H; s\rangle \rightsquigarrow b[n', \bot] \parallel n\,\langle b, o, \sigma, s\rangle} \ (\mathsf{release_H})$$

We see that suspending in low context releases and grabs both locks but suspending in high context affects only the high lock.

Now consider the following example of an insecure flow:

- A task $n_1$ with future type $\mathsf{Fut}^H_H(T_1)$ in cog $b_1$ makes a high while loop (e.g. while $h$ do skip) whose termination depends on secret data
- A task $n_2$ with future type $\mathsf{Fut}^L_L(T_2)$ in cog $b_1$ is about to make a low side effect (e.g. call a method in cog $b_2$ that does $\mathsf{l} := 0$)
- The low side effect can be blocked by a non-terminating high loop

This situation occurs in the following fragment of an ABS program:

```
class TaxRecordImpl(Int init_salary) implements TaxRecord {
        // init_salary : Int_H — we want to keep this value secret
        Int salary = init_salary;
        // salary : Int_H
        TaxChecker checker = null;
        // checker : TaxChecker_L

        Int const_SLICE_WIDTH = 2000;

        Int computeTaxes() {
```

```
        // computeTaxes : (H) -H-> Cmd^H(Int_H)
                Int taxes = 0;
                // taxes : Int_H
                Int untaxedSalary = this.salary;
                // untaxedSalary : Int_H
                Int sliceNb = 0;
                // sliceNb : Int_H
                while ( untaxedSalary > 0 ) {
                        Int taxedSlice = 0;
                        // taxedSlice : Int_H
                        if (untaxedSalary < const_SLICE_WIDTH) { taxedSlice = untaxedSalary; }
                        else { taxedSlice = const_SLICE_WIDTH; }
                        taxes = taxes + ((taxedSlice * (sliceNb * 5)) / 100);
                        untaxedSalary = untaxedSalary - taxedSlice;
                        sliceNb = sliceNb + 1;
                }
                return taxes;
        }

        Unit registerChecker(TaxChecker checker) {
        // registerChecker : (L, TaxChecker_L) -L-> Cmd^L(Unit_L)
                this.checker = checker;
                checker!newRecordAdded();
        }
}

class TaxCheckerImpl() implements TaxChecker {
        Int nbRecords = 0;
        // nbRecords : Int_L

        Unit newRecordAdded() {
        // newRecordAdded : (L) -L-> Cmd^L(Unit_L)
                nbRecords = nbRecords + 1;
        }
}

{
        TaxChecker checker = new cog TaxCheckerImpl();
        // checker : TaxCheckerImpl_L
        TaxRecord tr = new cog TaxRecordImpl(50000);
        // tr : TaxRecordImpl_L
        tr!computeTaxes();
        tr!registerChecker(checker);
}
```

Here tr is in cog $b_1$, checker is in cog $b_2$, the task created for computeTaxes is $n_1$, the task created for registerChecker is $n_2$. If the scheduler chooses to activate $n_1$ before $n_2$ then the final value of the low variable nbRecords depends on whether computeTaxes terminates, which may depend on the secret value init_salary. Thus information is leaked from a secret value to a low variable.

To prevent this insecure flow, we add an implicit suspend after each iteration of while and await loops:

$$\frac{}{n\,\langle b, o, \sigma, \mathsf{while}_l\ (e)\ s_1; s_2\rangle \rightsquigarrow \rightsquigarrow n\,\langle b, o, \sigma, \mathsf{if}\ (e)\ (s_1; \mathsf{suspend}_l; \mathsf{while}_l\ (e)\ s_1)\ \mathsf{else\ skip}; s_2\rangle}\ (\mathsf{while})$$

$$\frac{}{n\,\langle b, o, \sigma', \mathsf{await}_l(n'?); s\rangle \parallel n'\,\langle b', o', \sigma, x\rangle \rightsquigarrow \rightsquigarrow n\,\langle b, o, \sigma', s\rangle \parallel n'\,\langle b', o', \sigma, x\rangle}\ (\mathsf{await}_1)$$

$$\frac{}{n\,\langle b, o, \sigma', \mathsf{await}_l(n'?); s\rangle \parallel n'\,\langle b', o', \sigma, s'; x\rangle \rightsquigarrow \rightsquigarrow n\,\langle b, o, \sigma', \mathsf{suspend}_l; \mathsf{await}_l(n'?); s\rangle \parallel n'\,\langle b', o', \sigma, s'; x\rangle}\ (\mathsf{await}_2)$$

In D1.1a, loops could execute without suspending at all.

Now we consider an example of an insecure flow using high-low tasks:

- Low task $n$ in cog $b$ is executing a high conditional that in one of the branches (but not in the other) awaits for a high-low task $n_2$ in cog $b'$

- The high lock of $b'$ is held by a low task $n_3$ in cog $b'$
- Here it may depend on which branch of the high condional in $n$ we are in (an thus on the high variables in $n$) whether low steps must be made in $n_3$ before the next low step in $n$ or not

The following rule removes this dependency:

$$\frac{\text{the next step of } s_1 \text{ is low and the task } n_2 \text{ is high-low}}{\begin{array}{c} n\,\langle b, o, \sigma', \mathsf{await}_H(n_2?); s\rangle \parallel n_2\,\langle b', o', \sigma, \mathsf{grab}_H; s'; x\rangle \parallel \\ \parallel n_3\,\langle b', o_1, \sigma_1, s_1\rangle \parallel b'[n_3, n_3] \rightsquigarrow n\,\langle b, o, \sigma', \mathsf{suspend}_H; \mathsf{await}_H(n_2?); s\rangle \parallel \\ \parallel n_2\,\langle b', o', \sigma, s'; x\rangle \parallel n_3\,\langle b', o_1, \sigma_1, \mathsf{grab}_H; s_1\rangle \parallel b'[n_3, n_2] \end{array}}\ (\mathsf{await}_3)$$

This rule allows an implicit high scheduling point to be created before a low step in task $n$ in the situation occurring in the example. This scheduling point is used by the scheduler to take the high lock from task $n_3$ and allow task $n_2$ to grab it. This allows $n_2$ to terminate and $n$ can continue to make its next low step without having to wait for a low step in $n_3$. Thus the relative order of the next low step in $n$ and the next low step in $n_3$ is not fixed.

We have also removed some constructs from the language, to simplify the analysis. Thus we have no synchronous method calls (but we have asynchronous calls), no boolean guards (but we have future guards), and no interfaces (but we have classes).

### 3.1.3 Type-based analysis of the baseline example

Any type-based analysis would assign the security level H to certain values that the program operates with. In our example, we state that the salary of the taxpayer has high security level. In this case, the program is typable if we give the security level H to all other values, too. However, if we try to keep the levels of variables and fields as low as possible, we get the typing shown below. We list the entire program again, but near the declaration of each method, field or variable, we also indicate its type according to [79]. We have somewhat extended the type system with the handling of interfaces, templates and local calls (which are currently modeled as if they were inlined).

```
interface TaxServer4charity {
        Int getCharity();
        // getCharity : (L) -H-> Cmd^H(Int_L)
}

interface TaxServer extends TaxServer4charity {
        Unit registerTaxRecord(TaxRecord tr);
        // registerTaxRecord : (L, TaxRecord_L) -L-> Cmd^L(Unit_L)
}

class TaxServerImpl implements TaxServer {
        List<TaxRecord> taxRecords = Nil;
        // taxRecords : List⟨TaxRecord_L⟩_L

        Unit registerTaxRecord(TaxRecord tr) {
        // registerTaxRecord : (L, TaxRecord_L) -L-> Cmd^L(Unit_L)
                taxRecords = appendright(taxRecords, tr);
        }

        Int getCharity() {
        // getCharity : (L) -H-> Cmd^H(Int_H)
                Int totalAmount = 0;
                // totalAmount : Int_H
                Int nbRecords = length(taxRecords);
                // nbRecords : Int_H
                Int i = 0;
                // i: Int_H
                while ( i < nbRecords ) {
                        TaxRecord tr = nth(taxRecords, i);
                        // tr : TaxRecord_H
```

```
                        Fut<Int> fut = tr!getCharity();
                        // fut : Fut^H_H(Int_H)
                        Int amount = fut.get;
                        // amount : Int_H
                        totalAmount = totalAmount + amount;
                        i = i + 1;
                }
                return totalAmount;
        }
}

interface TaxRecord4taxPayer {
        Int getTaxes();
        // getTaxes : (H) -L-> Cmd^L(Int_H)
        Int getAmountPayed();
        // getAmountPayed : (H) -L-> Cmd^L(Int_H)
        Int payTaxes(Int charity, Int amount);
        // payTaxes : (L, Int_L, Int_H) -L-> Cmd^L(Int_H)
}

interface TaxRecord4taxChecker {
        Int verifyPayment();
        // verifyPayment : (L) -H,1-> Cmd^L(Int_H)
        Unit freeze();
        // freeze : (H) -H,1-> Cmd^L(Unit_H)
}

interface TaxRecord extends TaxRecord4taxPayer, TaxRecord4taxChecker {
        Unit registerChecker(TaxChecker checker);
        // registerChecker : (L, TaxChecker_L) -L-> Cmd^L(Unit_L)
        Int getCharity();
        // getCharity : (H) -H-> Cmd^H(Int_H)
}

class TaxRecordImpl(Int init_salary) implements TaxRecord {
        // init_salary : Int_H — we want to keep this value secret
        Int salary = init_salary;
        // salary : Int_H
        TaxChecker checker = null;
        // checker : TaxChecker_L
        Int charity = 0;
        // charity : Int_H
        Int amountPayed = 0;
        // amountPayed : Int_H
        Bool frozen = False;
        // frozen : Bool_H

        Int computeTaxes() {
        // computeTaxes : (H) -H,1-> Cmd^L(Int_H)
                return salary/10;
        }

        Unit registerChecker(TaxChecker checker) {
        // registerChecker : (L, TaxChecker_L) -L-> Cmd^L(Unit_L)
                this.checker = checker;
        }

        Int setCharity(Int amount) {
        // setCharity : (H, Int_H) -H,1-> Cmd^L(Int_H)
                if ( ~this.frozen ) {
                        if (amount < 0) amount = 0;
                        this.charity = amount;
                }
                return this.charity;
        }

        Int getAmountPayed() {
```

```
        // getAmountPayed : (L) -L-> Cmd^L(Int_H)
                return this.amountPayed;
        }

        Int getTaxes() {
        // getTaxes : (H) -L-> Cmd^L(Int_H)
                return this.computeTaxes();
        }

        Int getTaxBalance() {
        // getTaxBalance : (H) -H,1-> Cmd^L(Int_H)
                Int taxAmount = this.computeTaxes();
                // taxAmount : Int_H
                Int taxBalance = this.amountPayed - (taxAmount + this.charity);
                // taxBalance : Int_H
                return taxBalance;
        }

        Int payTaxes(Int charity, Int amount) {
        // payTaxes : (L, Int_L, Int_H) -L-> Cmd^L(Int_H)
                if ( ~this.frozen ) {
                        this.setCharity(charity);
                        this.amountPayed = this.amountPayed + amount;
                        if (checker != null) checker!checkTaxes(this);
                        // anonymous future : Fut^2_H(Unit_H)
                }
                Int taxBalance = this.getTaxBalance();
                // taxBalance : Int_H
                return taxBalance;
        }

        Int verifyPayment() {
        // verifyPayment : (L) -H,1-> Cmd^L(Int_H)
                Int taxBalance = this.getTaxBalance();
                // taxBalance : Int_H
                if (taxBalance < 0) { taxBalance = -1; }
                return taxBalance;
        }

        Unit freeze() {
        // freeze : (H) -H,1-> Cmd^L(Unit_H)
                this.frozen = True;
        }

        Int getCharity() {
        // getCharity : (H) -H-> Cmd^H(Int_H)
                while( ~this.frozen )
                        suspend;
                return this.charity;
        }
}

interface TaxPayer {
        Int getTaxes();
        // getTaxes : (L) -L-> Cmd^L(Int_H)
        Int getAmountPayed();
        // getAmountPayed : (L) -L-> Cmd^L(Int_H)
        Int payTaxes(Int charity, Int amount);
        // payTaxes : (L, Int_L, Int_H) -L-> Cmd^L(Int_H)
        Unit payTaxesInFull(Int charity);
        // payTaxesInFull : (L, Int_L) -L-> Cmd^L(Unit_H)
}

class TaxPayerImpl(TaxRecord4taxPayer taxRecord) implements TaxPayer {
// taxRecord : TaxRecord4taxPayer_L

        Int getTaxes() {
```

```
        // getTaxes : (L) -L-> Cmd^L(Int_H)
                Fut<Int> fut = taxRecord!getTaxes();
                // fut : Fut^L_L(Int_H)
                Int taxes = fut.get;
                // taxes : Int_H
                return taxes;
        }

        Int getAmountPayed() {
        // getAmountPayed : (L) -L-> Cmd^L(Int_H)
                Fut<Int> fut = taxRecord!getAmountPayed();
                // fut : Fut^L_L(Int_H)
                Int amountPayed = fut.get;
                // amountPayed : Int_H
                return amountPayed;
        }

        Int payTaxes(Int charity, Int amount) {
        // payTaxes : (L, Int_L, Int_H) -L-> Cmd^L(Int_H)
                Fut<Int> fut = taxRecord!payTaxes(charity, amount);
                // fut : Fut^L_L(Int_H)
                Int taxBalance = fut.get;
                // taxBalance : Int_H
                return taxBalance;
        }

        Unit payTaxesInFull(Int charity) {
        // payTaxesInFull : (L, Int_L) -L-> Cmd^L(Unit_H)
                Int taxes = this.getTaxes();
                // taxes : Int_H
                Int amountPayed = this.getAmountPayed();
                // amountPayed : Int_H
                Int balance = (charity + taxes) - amountPayed;
                // balance : Int_H
                this.payTaxes(charity, balance);
        }
}

interface TaxChecker {
        Int checkTaxes(TaxRecord4taxChecker taxRecord);
        // checkTaxes : (H, TaxRecord4taxChecker_L) -H,2-> Cmd^L(Int_H)
}

class TaxCheckerImpl() implements TaxChecker {

        Int checkTaxes(TaxRecord4taxChecker taxRecord) {
        // checkTaxes : (H, TaxRecord4taxChecker_L) -H,2-> Cmd^L(Int_H)
                Fut<Int> fut = taxRecord!verifyPayment();
                // fut : Fut^1_H(Int_H)
                Int balance = fut.get;
                // balance : Int_H
                if ( balance >= 0 ) {
                        taxRecord!freeze();
                }
                return balance;
        }
}

interface Charity {
        Int getCharity();
        // getCharity : (H) -H-> Cmd^H(Int_H)
}

class CharityImpl(TaxServer4charity server) implements Charity {

        Int getCharity() {
        // getCharity : (H) -H-> Cmd^H(Int_H)
                Fut<Int> fut = server!getCharity();
```

```
                // fut : Fut^H_H(Int_H)
                Int amount = fut.get;
                // amount : Int_H
                return amount;
        }
}

{
        TaxServer server = new cog TaxServerImpl();
        // server : TaxServerImpl_L
        TaxChecker checker = new cog TaxCheckerImpl();
        // checker : TaxCheckerImpl_L

        TaxRecord tr1 = new cog TaxRecordImpl(50000);
        // tr1 : TaxRecordImpl_L
        server!registerTaxRecord(tr1);
        tr1!registerChecker(checker);
        TaxPayer alice = new cog TaxPayerImpl(tr1);
        // alice : TaxPayerImpl_L

        TaxRecord tr2 = new cog TaxRecordImpl(75000);
        // tr2 : TaxRecordImpl_L
        server!registerTaxRecord(tr2);
        tr2!registerChecker(checker);
        TaxPayer bob = new cog TaxPayerImpl(tr2);
        // bob : TaxPayerImpl_L

        Charity charity = new cog CharityImpl(server);
        // charity : CharityImpl_L
        charity!getCharity();
        // anonymous future : Fut^H_H(Int_H)

        Fut<Int> fut = alice!payTaxes(50, 100);
        // fut : Fut^L_L(Int_H)
        bob!payTaxes(10,10000);
        fut.get; alice!payTaxesInFull(15);
}
```

In this example, we want TaxRecordImpl::init_salary to be in class H. Hence TaxRecordImpl::salary also has class H. Any other variables that are assigned values computed by expressions involving the field TaxRecordImpl::salary or the return values of the methods TaxRecordImpl::computeTaxes or TaxRecordImpl::getTaxes are likewise forced into the class H.

An interesting flow of information forces the class of the field TaxRecordImpl::frozen into H (in turn, affecting the level of the field TaxRecordImpl::charity and the return value of TaxRecordImpl::getCharity through the implicit flow in TaxRecordImpl::setCharity). Namely, the variable TaxRecordImpl::frozen is changed in the method TaxRecord::freeze which is called from TaxCheckerImpl::checkTaxes inside a high-context block. In TaxRecordImpl::getCharity we can also see how the termination of a method can carry sensitive information. The statement await this.frozen; is analyzed by our type system as the while-loop we can see in this method. The loop guard is of class H, hence the termination of this loop also carries information of class H.

### 3.1.4 Type-based analysis of the example with loops

When performing the type inference for the example program modified as described in Sec. 2.3, the constant const_SLICE_WIDTH will obviously get the type $\mathsf{Int}_L$. The type of TaxRecordImpl::computeTaxes changes — as the termination now depends on high variables, the new type is $(H) \xrightarrow{H} \mathsf{Cmd}^H(H)$. All integer variables in the method have the type $\mathsf{Int}_H$. The change of the method type also propagates to other methods:

- The methods getTaxes and getTaxBalance of class TaxRecordImpl now also have the type $(H) \xrightarrow{H} \mathsf{Cmd}^H(H)$.
- The method TaxPayerImpl::getTaxes gets the type $(L) \xrightarrow{H} \mathsf{Cmd}^H(H)$.
- The method TaxPayerImpl::payTaxes gets the type $(L, \mathsf{Int}_L, \mathsf{Int}_H) \xrightarrow{H} \mathsf{Cmd}^H(\mathsf{Int}_H)$.

- The method TaxPayerImpl::payTaxesInFull gets the type $(\mathsf{L}, \mathsf{Int}_{\mathsf{L}}) \xrightarrow{\mathsf{H}} \mathsf{Cmd}^{\mathsf{H}}(\mathtt{Unit}_{\mathsf{H}})$.
- The method TaxRecordImpl::payTaxes gets the type $(\mathsf{L}, \mathsf{Int}_{\mathsf{L}}, \mathsf{Int}_{\mathsf{H}}) \xrightarrow{\mathsf{H}} \mathsf{Cmd}^{\mathsf{H}}(\mathsf{Int}_{\mathsf{H}})$.
- The method TaxRecordImpl::verifyPayment gets the type $(\mathsf{L}) \xrightarrow{\mathsf{H}} \mathsf{Cmd}^{\mathsf{H}}(\mathsf{Int}_{\mathsf{H}})$.

We see that now the termination of all these methods also depends on high variables. All changes are also reflected in the types of interface methods that are implemented by listed methods.

The future variables storing the futures resulting from calls to the methods with changed types also have their types changed accordingly. Interestingly, the future fut defined in the main program now has the type $\mathsf{Fut}^{\mathsf{H}}_{\mathsf{H}}(\mathsf{Int}_{\mathsf{H}})$. This means that the very last call alice!payTaxesInFull(15) is now made in a high context. This is acceptable because the method can accept calls made from high contexts.

## 3.2 Information flow type system for JVM

The Java Virtual Machine (JVM) is a natural target for compiling ABS programs. To ensure that compiled ABS programs are executed securely and will not leak confidential information, it is therefore important to develop methods that are able to detect illegal information flows in JVM programs. In this task, UPM has developed a sound information flow type system for two fragments of the JVM.

The first fragment [18] covers a significant subset of (sequential) JVM programs including objects, arrays, methods, and exceptions. The type system improves substantially over previous work, notably in terms of language coverage and precision. In particular, it provides an accurate treatment of Java exceptions, and it relies on a refined notion of control dependence region to detect illicit flows; for the latter, we rely on preliminary static analyses that allow to curb the explosion in the control-flow graph that exceptions introduce. Because of the complexity in the definitions and proofs, we have developed a machine-checked formalization that includes several operational semantics of the Java Virtual Machine, the definitions of the security policy, of the type system and of the non-interference property, and a proof that the type system enforces non-interference, provided the preliminary analyses are correct. Each component is a significant piece of formalization in itself. For instance, the formalization of the operational semantics contains a significant number of rules; for example, the JVM virtual call has 5 different transitions (call on a null reference which generates a null pointer exception that may be caught or not, normal termination of the callee, termination by an exception that may be caught or not in the caller context). Second, the type system contains over 60 rules, and many rules have a large (up to 10) number of premises. Third, the proof of non-interference relies on unwinding lemmas that require reasoning about two program executions, leading to a very large number of cases in proofs. Moreover, the proof of correctness of the type system is stratified: one must first prove that control dependence regions are checked correctly, then prove that the type system is correct.

We have illustrated the expressiveness of our type system on a (simplified) JVM implementation of the TaxRecord case study; for readability, we present the corresponding Java source program. The code of the program is given in Figure 3.1 with its information flow type annotations given in a *Jif-like* syntax, and safety annotations given in comments. The program computes income taxes from an input array of taxable incomes and marital status. The program takes as argument an array `input` of inputs and a tax table `taxTable`. Then, for each index `i` in the array range, it performs a binary search to find an index `lo` such that

$$\mathtt{taxTable[lo].brackets} \leq \mathtt{input[i].taxableIncome} < \mathtt{taxTable[lo+1].brackets}$$

and updates the output array `out.tax[i]` with the computed tax (depending on marital status either `taxTable[lo].married` or `taxTable[lo].single` ) and then increment a counter `out.married_nb` or `out.single_nb` to count the whole number of married and single tax returns. The taxable incomes (field `taxableIncome`) and the array content of the income taxes (field `tax`) are given a high security level while other data are low.

We briefly comment on the annotations for runtime exceptions (`NP` means NullPointer, `NAS` means NegativeArraySize, `AOB` means ArrayOutofBounds). Most of these annotations can be easily proved with a simple null pointer analysis that maintains the invariant `this` $\neq$ `null`. The others require more complex arithmetic reasoning, for example a relational numeric static analysis.

```
class Output {
    int{L} single_nb;  int{L} married_nb;   int[]{L[H]} tax;

    Output{L}(int nbPeople) {
        single_nb = 0; // no NP exception
        married_nb = 0; // no NP exception
        tax = new int[nbPeople]; // no NP exception, no NAS exception
    }
    void updateMarried{L}(int{L} i, int{H} tax_data) {
        tax[i] = tax_data; // no NP exception, no AOB exception
        married_nb++; // no NP exception
    }
    void updateSingle{L}(int{L} i, int{H} tax_data) {
        tax[i] = tax_data; // no NP exception, no AOB exception
        single_nb++; // no NP exception
    }}

class Input {int{H} taxableIncome;   boolean{L} maritalStatus;}
class Tax {int{L} single;   int{L} married;   int{L} brackets;}

class TaxCalculation {

    Output{L} main{L}(Input[]{L[L]} input, Tax[]{L[L]} taxTable) {
        Output{L} out = new Output(input.length);
        for (int{L} i=0; i < input.length; i++) {
            int{H} lo = 0;
            int{H} hi = taxTable.length;
            try { while (lo+1 < hi) {
                    int{H} mid = (lo + hi) / 2;
                    if (input[i].taxableIncome
                        < taxTable[mid].brackets) //no AOB exception
                      {hi = mid;} else {lo = mid;};
                  };
            } catch (NullPointerException e){};
            if (input[i].maritalStatus)
                 { out.updateMarried(i,taxTable[lo].married);}
            else { out.updateSingle(i,taxTable[lo].single);};
        };
        return out;
    }}
```

Figure 3.1: The Tax Calculation program.

The second fragment [21] extends the first with a treatment of concurrency. Developing flexible information flow type systems for concurrent languages is notoriously difficult, because the concurrent execution of two secure programs may be insecure. This has led many information flow type systems for concurrent programs to impose stringent restrictions on programs, making such type systems overly restrictive for practical purposes. In [84, 85], Russo and Sabefeld introduce the notion of secure scheduler to provide a more flexible (and yet sound) approach to handle concurrency. Building on their idea, Barthe, Rezk, Russo and Sabelfeld [20] give a modular method for extending an information flow type system for a sequential low-level language into an information flow type system for a concurrent low-level language; the crux of the method is to make schedulers aware of the security environment that maps program points to a lower bound for the security level of their enclosing effects and guards, and to require that schedulers do not leak information through internal timing. Although the method is modular, its application to the type system of [18] raises a significant difficulty, because the soundness of the latter is based on a mix-step semantics, in which method calls are performed in one step. While the mix-step semantics makes soundness proofs considerably simpler, it is inappropriate for reasoning about concurrent programs, since the execution of separate threads might be interleaved. UPM has recently addressed this problem by enhancing the proof method of [18] so that

all reasoning about program executions is performed directly with respect to a small-step semantics. This requires substantial generalizations in the notions of state equivalence, and substantial adaptations in the unwinding lemmas. Moreover, we provide a more realistic treatment of thread creation, so that it matches more closely the operational semantics of the JVM, and recast one technical hypothesis of [20] in terms of control dependence regions, making its verification simpler.

In addition to develop sound information flow type systems for Java bytecode, we have also investigated the problem of type-preserving compilation. Type-preserving compilation is an essential tool to ensure that applications developed using information-flow aware programming languages are compiled into code that will be analyzed as secure by an enhanced bytecode verifier that enforces information flow policies. In addition, type-preserving compilation provides a proof technique for showing soundness of source type systems: more specifically, one can conclude that a source type system is sound from a proof of type-preserving compilation into a sound type system, and a proof that the compiler is semantics-preserving.

We have proved for both fragments a type-preserving compilation result: that is, we have proved that Java source programs which are provably secure by an information flow type system are compiled into JVM programs that are provably secure by our information flow type system. An appealing characteristic of type-preserving compilation for the concurrent fragment is *non-restrictiveness*: although the source-level type system is no more restrictive than a typical type system for a sequential language (i.e. we do not impose restrictions on the sequential constructs of the language to ensure that the concurrent composition of two sequential programs is secure), the compilation of typable programs is guaranteed to be typable at low-level. Future work includes extending these results to the compilation from ABS to the Java Virtual Machine; as a preliminary step, we intend to provide a formal definition of the compilation from ABS to the Java Virtual Machine, and to show that the compilation is semantics-preserving.

# Chapter 4

# Logical methods

This chapter explores logic-based methods for enforcing security policies of programs. These methods offer significant advantages:

- expressiveness: complex observations about programs and systems are naturally captured by logic. For instance, program logics allow to state arbitrary trace properties with pre- and post-conditions, and intermediate assertions. In addition, non-functional properties can be captured using ghost state and ghost statements. One main benefit of logic-based formalisms is that they can express in a general setting the wide range of security policies considered in HATS, including declassification policies and policies that describe explicitly adversary knowledge.

- precision: logic-based formalisms are equipped with powerful proof systems for reasoning about intricate program behavior. The versatility of these proof systems allows on the one hand to reason directly about the security policy of interest (rather than about an alternative policy that would be more suitable for specification and verification in more limited formalisms), and on the other hand to reason accurately about the security of programs. Precision is of special significance for security analyses, as security analyses often generate many false alarms, leading to secure programs being rejected.

- integration: logic-based formalisms are routinely used to verify safety and functional properties of programs. Often, the security of a program depends on its adherence to basic safety policies, or on its functional behavior. Logic-based formalisms allow to express in a unified framework a wide range of safety, correctness, and security properties, and thus favors modular proofs.

- tool support: program verification methods are supported by effective verification environments, that are increasingly used with great effect on large-scale, industrial-strength software. Adopting logic-based methods to verify security properties allows leveraging recent advances in verification.

This chapter addresses three outstanding issues: first, we study how to cast confidentiality properties of programs in a formalism that is supported by existing program verification environments. Specifically, we study how information flow properties, which customarily quantify over two executions of a program $P$, can be formulated as properties of a single execution of a program built from $P$. More precisely, we show that for many settings of interest one can reduce the verification of information flow policies of $P$ to the verification of a trace property of a program built from $P$. It follows that information flow policies can be verified using state-of-the-art verification tools. Second, we study how symbolic execution can be used for reasoning about information flow, building on the self-composition technique just described. The advantage of automatically employing symbolic execution is that fewer verification conditions need to be inserted into a program, and thus checking can proceed automatically. Finally, we study how to express different variants of information flow security policies (noninterference and declassification variants) in epistemic logic. The knowledge operator (K) introduced in this logic allows a simple and elegant formulation of those information

flow policies. We then show that those epistemic formulas can be automatically verified using a model checker based on symbolic execution and a satisfiability modulo theory (SMT) solver.

## 4.1 Self-composition

The first step towards verifying information flow policies using program verification tools is to express such policies in a form that can be handled by these tools. Encoding of information flow policies into standard program logics is not immediate, because such policies are not safety properties, but rather properties of two execution traces. In fact, existing proposals to encode an information flow policy for a program $P$ as a property of the same program are unsatisfactory: they either require extending the program logic, or approximating non-interference as a safety property. Thus, we take a different perspective on the problem, and reduce information flow policies of a program $P$ to a property about single program executions (universally quantified over all possible program inputs) of another program $\hat{P}$ constructed from $P$. We call "self-composition" the reduction of information flow policies to a safety property: an information flow policy of a program $P$ reduces to a property about single program executions (universally quantified over all possible program inputs) of the program $P; P'$, where $P'$ is a renaming of $P$. Thanks to self-composition, general-purpose logics such as Hoare-like logics or temporal logics, which provide a standard means to specify and verify safety properties of programs, can also be used to verify a wide range of information flow policies, and these policies can be handled with standard verification infrastructures.

Consider the following code snippet from the case study:

```
Int getCharity() {
 Int totalAmount = 0;
 Int nbRecords = length(taxRecords);
 Int i = 0;
 while ( i < nbRecords ) {
  TaxRecord tr = nth(taxRecords, i);
  Fut<Int> fut = tr!getCharity();
  Int amount = fut.get;
  totalAmount = totalAmount + amount;
  i = i + 1;
}
 return totalAmount;
}
```

Assume we want to prove that the result $totalAmount$ does not depend on the field salary of the taxpayer (private information), but only on the field charity (public information). In order to do so, we use the precondition $\Phi \doteq \forall i, i \in [1..length(taxRecords)].nth(taxRecords, i).charity = nth(taxRecords', i).charity$. Our desired postcondition is $\Psi \doteq totalAmount = totalAmount'$. Checking this property then amounts to carrying out the following verification task:

```
Int totalAmount = 0;
Int nbRecords = length(taxRecords);
Int i = 0;
Int totalAmount' = 0;
Int nbRecords' = length(taxRecords);
Int i' = 0;
 while ( i < nbRecords ) {
 TaxRecord tr = nth(taxRecords, i);
 Fut<Int> fut = tr!getCharity();
 Int amount = fut.get;
 totalAmount = totalAmount + amount;
 i = i + 1;
}
while ( i' < nbRecords' ) {
 TaxRecord tr' = nth(taxRecords', i');
 Fut<Int> fut' = tr'!getCharity();
 Int amount' = fut'.get;
 totalAmount' = totalAmount' + amount';
 i' = i' + 1;
}
```

In order to establish the desired property, the intermediate postcondition has to be strong enough to describe the value of variable *totalAmount* in terms of the initial values of the remaining variables. Achieving this requires the following invariant for the first loop (and similarly for the second loop):

$$totalAmount = \sum_{k=0}^{i} nth(taxRecords, k).charity$$

This task can be carried out using any off-the-shelf tool for program verification. This illustrates the main virtue of self-composition, enabling the use of verification tools for establishing that programs conform to a certain class of security policies.

Since its introduction, self-composition has been widely used for verifying information flow properties of programs. The idea of self-composition was further generalized in a series of papers, leading to the notion of hyperproperties by Clarkson and Schneider [37]. In a recent breakthrough, Milushev and Clarke [76] introduce the notions of holistic and incremental hyperproperties. Specifications of holistic hyperproperties tend to be more intuitive to read but are difficult to verify, whereas incremental hyperproperty specifications have a straightforward verification approach. Since most interesting security related hyperproperties are in the class of holistic hyperproperties, Milushev and Clarke introduce the process of incrementalization to convert holistic specifications into incremental ones. Further, Milushev and Clarke develop three incrementalizable classes of holistic hyperproperties and a give respective verification method for each of them.

In spite of its theoretical appeal, self-composition suffers from an important practical drawback, specially when dealing with programs that include loops, as pointed out by Terauchi and Aiken [93]. Note that the required invariant to establish the desired security policy of the previous example is non-linear. Since invariant inference has proved to be one of the major bottlenecks in program verification, keeping invariants simple is a key factor to keep program verification feasible and scalable. With this goal in mind, a more general notion, coined product programs was introduced [16]. In this setting, the relational program verification task in the previous example is reduced to the following standard verification task:

```
Int totalAmount = 0;
Int nbRecords = length(taxRecords);
Int i = 0;
Int totalAmount' = 0;
Int nbRecords' = length(taxRecords);
while ( i < nbRecords ) {
 TaxRecord tr = nth(taxRecords, i);
 Fut<Int> fut = tr!getCharity();
 Int amount = fut.get;
 totalAmount = totalAmount + amount;
 TaxRecord tr' = nth(taxRecords', i);
 Fut<Int> fut' = tr'!getCharity();
 Int amount' = fut'.get;
 totalAmount' = totalAmount' + amount';
 i = i + 1;
}
```

In order to carry out this verification, the required invariant is $totalAmount = totalAmount' \wedge \Phi$, which is much simpler.

The main observation that justifies the soundness of this method is the fact that the loops of the two programs can be "synchronized", since their guards are equivalent. While establishing such property in the general case of relational properties automatically is not feasible, in the setting of non-interference security policies, this property amounts to checking that both guards depend only on public values. Hence, this kind of property can be established using a secure information flow type system such as those presented in Chapter 3.

Within Task 4.3, we are studying techniques based on product programs, with the aim of providing automated tool support for security policies such as secure information flow, and relational properties in general.

## 4.2 Symbolic execution

Symbolic execution [64] is a program analysis technique used to investigate the possible execution traces of a program. The idea is to replace program inputs with input symbols and thus instead of executing the program with concrete values, to execute it with symbolic expressions over the input symbols. Symbolic execution is a general approach that can be used to check or prove a range of properties of programs.

Because self-composition (Section 4.1) reduces information flow to a safety property, it follows that symbolic execution can be used in combination with self-composition to increase the precision of traditional information flow analyses. Indeed, this idea has already been explored by Darvas et al. [43] who integrate with a theorem proving approach. The idea has been further explored in recent work by Milushev and Clarke [75] and the proposed approach is summarized next. The key difference between these two works is that Milushev and Clarke require fewer annotations in the program text. To illustrate the approach, the following simple program exhibiting implicit information flow is used:

```
Int l;
//# high
Int h;
if (l > h) { l = 1; } else { l = 0; }
```

Listing 4.1: Annotated program

The approach starts with partitioning the program variables into *low* and *high*, based on programmer specified annotations of the high ones. An example high variable is the *salary* of a Tax Payer, as in the Tax Payer case study discussed in Section 4.1. Line 2 in the program in Listing 4.1 illustrates the annotation of the only high variable. Then the variables are made symbolic and a type-directed transformation adopted from the work of Terauchi and Aiken [92] is applied to the program; the transformation is a variant of the self-compositional approach (Section 4.1), the result can be seen in Program 4.2. Programs are then translated into C and analysed using the KLEE symbolic execution framework [31]. The approach described here could easily have used the KeY tool [24].

```
Int l0; Int l1;
//# klee_make_symbolic(&l0, sizeof(Int), "Int l0"); klee_make_symbolic(&l1, sizeof(Int), "Int l1");
//# high
Int h0; Int h1;
//# klee_make_symbolic(&h0, sizeof(Int), "Int h0"); klee_make_symbolic(&h1, sizeof(Int), "Int h1");
if (l0 > h0) { l0 = 1; } else { l0 = 0; }
if (l1 > h1) { l1 = 1; } else { l1 = 0; }
```

Listing 4.2: Transformed program

Certain extensions of the transformation in order to deal with procedures and dynamically allocated data structures are developed; the latter require reasoning about the heap and a modified definition of noninterference. These are not needed for the example, more detail is available in the attached paper [75]. After the transformation is complete assumptions and assertions specifying the noninterference policy have to be placed. In this simple case only the statements on lines 3 and 9 are needed:

```
int l0; int l1;
//# klee_make_symbolic(&l0, sizeof(Int), "Int l0"); klee_make_symbolic(&l1, sizeof(Int), "Int l1");
//# klee_assume(l0 == l1);
//# high
int h0; int h1;
//# klee_make_symbolic(&h0, sizeof(Int), "Int h0"); klee_make_symbolic(Inth1, sizeof(Int), "Int h1");
if (l0 > h0) { l0 = 1; } else { l0 = 0; }
if (l1 > h1) { l1 = 1; } else { l1 = 0; }
//# klee_assert(l0 == l1);
```

Listing 4.3: Transformed program with assumptions and assertions

Finally, symbolic execution can be used as a program analysis tool for noninterference. If it is able to analyze all possible paths in the transformed program, it can decide whether the program is secure or not. Otherwise the tool may either eventually return an error, implying the program is insecure or keep running

indefinitely; in the latter case the proposed approach cannot determine whether the program is secure or not. Even if not all paths can be covered, the approach in general is useful for testing. The paper [75] also shows how to tackle a notion of declassification in the *what* dimension. The feasibility of the presented approach has been illustrated using a prototypical tool based on the KLEE symbolic execution engine [31]. In our example, the tool transforms the program in Listing 4.1 into the program in Listing 4.3 and then passes the latter to KLEE. Thus the implicit flow in the original program materializes in an assertion fail error on the condition $(l_0 == l_1)$.

The prototypical tool handles various examples of explicit and implicit information flow as well as a notion of information release. Because the transformations are based on semantic methods, the approach is more precise than typical information flow type systems resulting in the lack of false positives. The approach could potentially be used together with a flow-sensitive type system (Chapter 3): first, the type system checks the program; then, if the program is rejected, the proposed approach could be used to further investigate the error and try to find out if the type system was possibly too strict and the program is actually secure.

At present the techniques are limited to bodies of core ABS methods. Dealing with a more extensive language fragment is a challenge, but this can be achieved by exploiting the symbolic execution that is being developed as a part of the verification of ABS models, as is discussed in Deliverable D2.5 [46] and Task 4.3, as soon as it becomes available. Beyond expanding the language fragment dealt with, the approach suffers from traditional weaknesses of symbolic execution, such as problems with scalability for large numbers of paths, dependence on the power of the constraint solver and difficult interaction with the environment. A formal soundness result and the investigation of scalability are left for future work.

## 4.3 Epistemic logic

As stated above and demonstrated by numerous research publications, information flow security policies can be and are expressed using different formalisms. However, it seems that little work has been done to express information flow security policies as formulas in the epistemic logic. Epistemic logic has been developed to reason about knowledge, or absence of knowledge. It therefore seems to be an adequate candidate to express and reason about policies concerning the absence of knowledge gained by an attacker observing the execution of a program.

The work presented in one of the attached papers [9] explores how to express the standard noninterference policy (a strong information flow policy) and different variants of declassification policies (weaker but more useful information flow policies) using epistemic temporal logic formulas. Without entering into the details, in the special case of a program having a single public input variable $l$, a single secret input variable $h$ and an output statement generating values observable by the attacker during the execution, noninterference can be expressed using the following epistemic formula where $l_0$ is the initial value of $l$, $h_0$ is the initial value of $h$ and $L$ is the possibilistic epistemic operator:

$$\forall v.(l_0 = v \rightarrow \forall w.L(l_0 = v \wedge h_0 = w))$$

Assuming that $\mathcal{O}(\pi, i)$ is the sequence of values observed by the attacker during the execution $\pi$ up to the $i^{\text{th}}$ step. The formula above states that at any step $i$ of any execution $\pi$, for all values $v$ and $w$ such that $v$ is the initial value of the public input $l$ for the current execution $\pi$, it is possible to find, among the execution steps $(\pi', i')$ such that $\mathcal{O}(\pi, i) = \mathcal{O}(\pi', i')$, an execution step $(\pi'', i'')$ for which $l_0 = v \wedge h_0 = w$ holds, i.e. the initial public input of $\pi''$ is $v$ and the initial secret input of $\pi''$ is $w$. This means that any execution started with the public input $v$ and any secret input could have generated the values observed up to the execution step $i$ of $\pi$. Therefore, from the point of view of the attacker, knowing that the initial public input is $v$ and observing the execution $\pi$ is not sufficient to learn about the secret input.

The above mentioned attached paper [9] shows that this formula can be naturally extended to express noninterference for any number of inputs, and that formulas to express declassification in the "what", "where" and "when" dimensions follows the same structure. Ongoing work, which is presented in an attached

draft [10], tackles the problem of verifying such formulas. The approach taken follows a different path but ultimately bears some similarities with the work presented in Section 4.2. A prototype called Encover is under finalization. The approach followed consists in extracting a behavioral model from a program using a concolic testing tool called Symbolic PathFinder (SPF) [96, 63]. This behavioral model, called a symbolic output tree (SOT), expresses potential outputs and the conditions under which they can occur using expressions built from variables standing for the initial input values of the program. The prototype then follows two different approaches to verify, on this behavioral model, an information flow policy expressed using an epistemic formula. The first approach is based on an epistemic model checker called MCMAS [69], the second approach generates a formula, whose negation holds only if the security policy holds for the behavioral model, and feeds it to a satisfiability modulo theory (SMT) solver such as Z3 [44].

The current version of the prototype of the second approach has been applied to the TaxRecord case study.

### 4.3.1 Application of Encover to the case study

In order to analyze the case study program using Encover, some small adaptations of the code had to be made. First, due to various restrictions and constraints in the tools used, the current prototype of Encover handles only integer arithmetic. Therefore, the computation of taxes has been modified such that incomes ($X$) are entered in thousands of dollars (K$), and the computation of $Y\%$ of the income is computed using the formula: $(X \times 10) \times Y$. Moreover, the case study is intrinsically an interactive program whose behavior depends on the actions of the tax payers (the rest is mainly deterministic). However, Symbolic PathFinder (SPF) analyzes programs whose behavior depends only on initial inputs and internal (possibly random) computations (more advanced interaction could be simulated by coding specific Choice classes, however this is outside of the scope of the current prototype). In order to handle this problem, 3 different scenarios have been coded and analyzed by Encover. The first scenario (`smpl`), involves a single tax payer (`Alice`) which queries for her amount of taxes and pays that exact amount without adding any donation. The only input in this scenario is the income of `Alice`. The second scenario (`oneP`) involves the same tax payer which starts by performing a first payment involving a donation, then, if she has under-payed, queries for her amount of taxes and pays what remains, including the donation. The inputs in this scenario are Alice's income, donation and first payment. The last scenario (`twoP`) involves two tax payers, `Alice` and `Others` standing for all the other tax payers, that concurrently (or sequentially in the case of the monothreaded variant) act as Alice in the second scenario. In this scenario, there are 6 inputs: incomes, donations and first payments of Alice and Others.

For every scenario and case study implementation variant, Encover is used multiple times to verify the non-interfering behavior of the program with regard to the 3 different principals (Alice, tax checker and charity, ranged over by $P$) under different policies regarding values that have to be protected from those principals. Every analysis involves a different configuration of Encover. Among other parameters such as input domains, there are 3 main parameters to configure: the input values (or more generally expressions) known by $P$ at the beginning (the low values in the theory), the input expressions that should be kept secret from $P$ (the high values in the theory), and finally the events and associated values that are observable by $P$. This last parameter is configured by providing a string with wild-cards (e.g. `*.print*(0)`) specifying which method calls are observable by $P$, and which parameter or return value $P$ will observe. In the case of the tax checker, resp. charity, the configuration of this parameter indicates that the return value of any method in TaxRecord4taxChecker, resp. TaxServer4charity, is observable. In the case of Alice, specifying that the return value of any method in TaxRecord4taxPayer is observable would prevent Encover to distinguish between observations made by `Alice` and `Others`. Therefore, the generated behavioral model (SOT) would contain observations made by both, instead of only the observations made by Alice. However, the regular-like expression specifying observable events may include some runtime values of method call parameters. To specify the events observable by Alice, a method $m$ taking as parameter a tax payer name and another value is coded with an empty body, and a call to this method is inserted in any method specified in TaxRecord4taxPayer with parameters the name of the tax payer for this tax record and the value that is

to be returned by the method in which a call to $m$ is inserted.

Table 4.1 contains the evaluation results of Encover run on some multiple monothreaded tests, including the ones described previously. The multithreaded tests generate the exact same behavioral model as the monothreaded tests (for this specific program whose behavior specification is the same for the two variants). Therefore there is no difference between them from the point of view of the non-interference answer from Encover (the only difference lies in the time Symbolic PathFinder (SPF) takes to extract the behavioral model from the program due to the higher complexity and number of paths in the multithreaded case). The remainder of this section focuses on the study of the non-interference analysis results returned by Encover for some of the monothreaded configurations. The attached paper [10] contains an additional study of those results from an efficiency/performance point of view. The answer of Encover regarding non-interference is given in row 2 (NI) in Table 4.1. In this figure, the Tax Record related tests are named *S-P-R*, where *S* indicates the scenario used for this test, *P* is the name of the principal for which the behavior of the program is verified to be non-interfering, and finally *R* specifies whether the test has been run on the program variant having a `Fix` or `Variable` tax rate.

| RUN | NI | Timing (in ms) | | | | Fml | | |
|---|---|---|---|---|---|---|---|---|
| | | O (in s) | E | G | S | V | A | I |
| smpl-Alice-F | Y | .6 | 173 | 3 | 8 | 2 | 17 | 62 |
| smpl-Alice-V | Y | 3.3 | 1528 | 249 | 1054 | 2 | 18777 | 103926 |
| smpl-charity-F | Y | .6 | 179 | 3 | 8 | 2 | 8 | 26 |
| smpl-charity-V | Y | 2.5 | 1470 | 86 | 576 | 2 | 5968 | 31098 |
| smpl-taxChecker-F | Y | .6 | 167 | 3 | 8 | 2 | 8 | 36 |
| smpl-taxChecker-V | Y | 2.7 | 1452 | 88 | 724 | 2 | 5968 | 37650 |
| oneP-Alice-F | Y | 2.3 | 1900 | 6 | 12 | 6 | 42 | 236 |
| oneP-Alice-V | Y | 6.9 | 3659 | 240 | 2546 | 6 | 29114 | 179100 |
| oneP-charity-F | N | 2.3 | 1861 | 4 | 42 | 6 | 28 | 107 |
| oneP-charity-V | N | 4.7 | 3517 | 96 | 637 | 6 | 7916 | 42957 |
| oneP-taxChecker1-F | N | 2.3 | 1872 | 4 | 27 | 6 | 32 | 154 |
| oneP-taxChecker2-F | Y | 2.4 | 1895 | 4 | 25 | 6 | 32 | 154 |
| oneP-taxChecker3-F | Y | 2.3 | 1844 | 4 | 24 | 6 | 32 | 164 |
| oneP-taxChecker4-V | Y | 128.9 | 3632 | 129 | 124709 | 6 | 14500 | 84462 |
| twoP-Alice-F | Y | 6.3 | 5820 | 6 | 26 | 12 | 57 | 266 |
| twoP-charity-F | Y | 6.5 | 5962 | 10 | 45 | 12 | 107 | 588 |
| twoP-taxChecker3-F | Y | 6.6 | 5852 | 12 | 250 | 12 | 159 | 1134 |

- NI: `Y` iff Encover concludes that the program is non-interfering
- Timing: given in ms (O: overall (in s); E: model extraction (JPF+symbc); G: interference formula generation; S: interference formula satisfiability checking)
- Fml: information related to the interference formula (V: number of distinct variables; A: number of atomic formulas; I: number of instances of variables or constants)

Table 4.1: Encover evaluation results

In the case of the `smpl` scenario, all configurations are found non-interfering. In this scenario, the only input is the income of Alice, which is known by Alice and has no relation to the values observed by charity (0, as there is no donation in this scenario) and taxChecker (0, as Alice pays directly the exact amount of taxes she has to pay). It can therefore be concluded that Encover successfully handles the different configurations for the `smpl` scenario. In the case of the `oneP` scenario, the inputs are the income, donation and first payment of Alice which are known by Alice and should be kept secret from charity and taxChecker. Obviously, this scenario is non-interfering from the point of view of Alice, but can not be from the point of

view of charity. Indeed, charity observes the sum of donations, which in this scenario includes only Alice's donation and is therefore "statically" equal to a value that is supposed to be kept secret from charity. For the principal taxChecker, many different configurations have been tested: in the taxChecker1 case, the value "$income \times F\% + donation > payment$" is declassified where $F$ is the fix tax rate; for taxChecker2, "$income \times F\% + donation - payment$" is declassified. Encover finds the configuration interfering for taxChecker1 and non-interfering for taxChecker2. Indeed, the value revealed to taxChecker in the specification of TaxRecord is "if $income \times F\% + donation > payment$ then $-1$ else $payment - (income \times F\% + donation)$". The principal taxChecker3 corresponds exactly to the declassification of this formula. For the variable tax rate case, the expression computing the taxes $((\sum_{n=1}^{N} n \times V\% \times slice) + ((N+1) \times V\% \times (income \mod slice))$ where the $n^{th}$ slice is taxed $(n \times V)\%$ and $N = income \div slice$ is the number of full slices) can be declassified to the taxChecker by unfolding $\sum_{n=1}^{N} n$ into $((N+1) \times N/2)$. This declassification corresponds to the principal taxChecker4. The case of the `twoP` scenario, is similar to the previous case for Alice and taxChecker. However, this time there are two different donations, one from Alice and one from Others. By declassifying "$donationAlice + donationOthers$" to charity, Encover concludes that charity does not learn more than is allowed. In conclusion, apart from potential efficiency problems linked to Symbolic PathFinder in presence of multithreading and numerous paths, as well as the current memory inefficient internal representation of formulas, the Encover prototype reacts as expected and can handle the majority of configurations of the tax record scenarios.

# Chapter 5

# Dynamic methods

As stated previously, information flow security policies are more precise and "safer" than the majority of security policies widely deployed on modern devices, including access control policies. The vast majority of work on enforcement of information flow policies concern static methods as those presented in Chapter 3. However, there exists a lesser known line of work concerning dynamic enforcement of information flow related security policies. This chapter introduces an introductory survey on dynamic information flow security [66] and presents two dynamic enforcement mechanisms applied to the HATS problem domain.

The vast majority of work presented in the attached survey [66] is theoretical, but practical tools are starting to emerge. With the increase of personal digital devices carrying personal data and requiring to communicate *specific data complying with different security policies* to the outside world, the need for security mechanisms at the granularity of information flow increases; as well as the public interest for such mechanisms, as attested by the lawsuits against Apple [83] and Google [74], or the good coverage received in generalist medias [23, 1, 73, 53, 34] by TaintDroid [49] (an Android version extended with an information flow security mechanism).

Dynamic enforcement of information flow related security policies faces one major difficulty to circumvent: information must not be considered equivalent to data. More precisely, it is not sufficient to observe a data to be able to determine what information is carried by this specific data. As stated by Ashby [6], a data carries more information than its intrinsic value.

> "the information carried by a particular message depends on the set it comes from. The information conveyed is not an intrinsic property of the individual message." [6, § 7/5 page 124].

For example, if someone is allowed a single word to describe the dominant color of a picture, if the data "green" is used, the majority of people will not only get from this word that the picture is dominantly green, but also that it is *potentially* a landscape picture and that, the picture is *definitely* not a black and white one. Pushing further, if people receiving the data "green" know from which set of pictures the described one comes from, then they may even know precisely which picture it is. In this example, the *data* carried by the message is "green" but the *information* it carries depends on the context and is usually more important than the intrinsic information contained in the data "green". To determine what information is carried by a data which becomes observable to the attacker, it is necessary to know what other data could have been observed at the same point and under which conditions.

Static analyses provide an answer valid for any potential execution of the analyzed program. As static techniques for information flow security already have to take into consideration any potential execution of a program, this difference between data and information does not affect them as badly as dynamic techniques which observe a single execution. In order to fulfill their goal, dynamic techniques for information flow security policies have to be enhanced with abilities to predict or constrain potential variation of behavior from one execution to the other.

The presentation of those dynamic mechanisms in the attached survey [66] is organized along the level at which they apply: *computing base* or *computed load*. The first part of the survey explores the state-of-the-art of dynamic information flow security mechanisms described at the computing base level. Those mechanisms

apply at different levels, mainly binary code or system events. They take the form of monitors and involve enhanced execution environments (to the extent of enhanced hardware). Based on those techniques, practical tools have been proposed, such as the architectural frameworks RIFLE [94] and Raksha [40]. The majority of information flow security mechanisms applying on the computed load are compile-time analyses. Those analyses rely on different well-known techniques such as type systems, abstract interpretation or constraint resolution [87]. There are fewer mechanisms, applying on the program under scrutiny, that are dynamic. Those are described in the second part of the survey. They can serve as the basis for the development of special purpose interpreters, program transformation, or testing environments.

Lots of dynamic information flow security mechanisms are related to the notion of *confinement* introduced by Lampson in 1973 [65]. A confined process has really limited capabilities to interact with the rest of the system and with the outside world. It is a general notion aiming at enforcing that a process or information system does not unduly leak confidential data. Lampson states that a process can be confined only if it is not able to maintain secret information longer than its own execution. Such processes are called *memoryless* [51, 54]. Additionally, it is required to call only secure or confined programs, and to communicate with the outside world using masked outputs. A masked output can only send values belonging to a predefined set of safe output values. Lipner [68] explores the available techniques which can be used to confine a process. Sandboxing [52, 55] is a popular generic technique to enforce a confined execution of a program [82]. However, even if confinement is really restrictive, which could be a problem by itself, masked outputs do not guaranty absence of leakage. The allowed output values can be reused as a new alphabet to encode disallowed output values.

The technique described in Section 5.1 can be seen as a form of strong confinement. The program to execute is run in parallel in different confined processes, one for each security level. Masking occurs for inputs with a security level higher than the confined process and consists in replacing the value by a default one. Section 5.2 presents a different approach where dynamic information flow security mechanism is not the goal but one of the means. The goal here is to enforce "data processing" security policies by relying on the tracking mechanism of a dynamic information flow mechanism, in this case TaintDroid [49].

## 5.1 Secure multi-execution

Secure multi-execution (SME) is a dynamic enforcement technique to ensure secure information flow by running one execution of the program per security level, and by reinterpreting input/output operations w.r.t. to its associated security level.

SME is sound, in the sense that the execution of a program under SME is non-interfering, and precise, in the sense that for programs that are non-interfering in the usual sense, the semantics of a program under SME coincides with its standard semantics. A further virtue of SME is that its core idea is language-independent; it can be applied to a broad range of languages.

Consider the following snippet taken from the case study, slighltly modified and obviously insecure:

Int getCharity() Int totalAmount = 0; Int nbRecords = length(taxRecords); Int i = 0; while ( i ¡ nbRecords ) TaxRecord tr = nth(taxRecords, i); Fut¡Int¿ fut = tr!getCharity(); Int amount = fut.get; totalAmount = totalAmount + amount; i = i + 1; Fut¡Int¿ fut' = tr!getSalary(); evil!leak(fut'); return totalAmount; Assume that in this setting we want to give access to an external party to the value of the total amount donated to charity, while preventing private information, such as the salary of individuals, to be leaked. It is clear that the previous snippet of code does not comform to this policy: it obviously leaks the salary of each individual through a direct call to an evil third party.

In the setting of ABS we treat function arguments and calls to object procedures as inputs, and return values and arguments of calls to objects procedures as outputs. A security policy can then be realised as a mapping from objects, and object fields to security labels. In this example, we would set the salary field as private, and the charity field as public. Also, we would assign the evil object a public label, and we would assume that the context in which the procedure will be called as public—hence the return value should be public as well. In an SME based environment, the following two copies of the program would be executed, with a modified semantics for input and output statements:

```
Int getCharity() {
  Int totalAmount = 0;
  Int nbRecords = length(taxRecords);
  Int i = 0;
  while ( i < nbRecords ) {
    TaxRecord tr = nth(taxRecords, i);
    Fut<Int> fut = tr!getCharity();
    Int amount = fut.get;
    totalAmount = totalAmount + amount;
    i = i + 1;
    Fut<Int> fut' = tr!getSalary(); undefined;
    evil!leak(fut');
  }
  return totalAmount;
}
```

The high copy is then executed as follows:

```
Int getCharity() {
  Int totalAmount = 0;
  Int nbRecords = length(taxRecords);
  Int i = 0;
  while ( i < nbRecords ) {
    TaxRecord tr = nth(taxRecords, i);
    Fut<Int> fut = tr!getCharity() undefined;
    Int amount = fut.get;
    totalAmount = totalAmount + amount;
    i = i + 1;
    Fut<Int> fut' = tr!getSalary();
    evil!leak(fut');
  }
  returntotalAmount void;
}
```

It is clear that this execution model prevents undesired flows by "disconnecting" low outputs from high inputs. While this transformation modifies semantics of insecure programs, the semantics of those that are secure to begin with is preserved.

In spite of its many virtues secure multi-execution suffers from one fundamental drawback: it is not easy to deploy. All existing implementations of SME require modifications to the underlying computing infrastructure (OS [32], browser [25], virtual machine [47], trusted libraries [60]). Specifically, it is hard to deploy SME-based enforcement techniques for heterogeneous infrastructures. Such is the case of ABS, since the deployment of SME would entail rewriting all of its many backends.

In order to ease the deployment of SME, KUL and UPM have developed a new implementation technique that circumvents the need to modify the computing infrastructure for SME-based enforcement. Our starting point is the essential insight of secure multi-execution: non-interference can be guaranteed by executing multiple copies of the program (one per security level) and by ensuring that the copy at level $l$ only outputs to channels at level $l$, and that it only gets access to inputs from channels that are below or equal to $l$. Instead of relying on a purpose-specific semantics, we devise static program transformations that achieve an equivalent effect. Specifically, we introduce a program transformation that takes a sequential program $P$ and builds a concurrent program $[P]_{conc}$ containing multiple suitably synchronized copies of $P$—one per security level. Then, we sequentialize the resulting program to obtain a program $[P]_{seq}$, written in the same language as $P$. We show that the SME semantics of $P$ are equivalent to the standard semantics of $[P]_{conc}$ and $[P]_{seq}$, and conclude that the transformations are sound, transparent, and practical.

Future work includes developing SME to concurrent object-oriented languages, and in particular to the concurrent fragment of the JVM considered in Chapter 3, and to core ABS.

## 5.2 Data flow analysis

In our work we present a framework for monitoring and enforcing policies regarding direct data flows. In relation to our previous work, [41, 42] and similar work by others, [59, 58, 95], our technique can be positioned somewhere between control flow monitoring and information flow monitoring. It is similar to

control flow monitoring in the sense that the framework is capable of monitoring the order of observable actions but rather than enforcing temporal policies such as "$f$ may be evaluated after $g$ but not vice versa" the framework handles policies such as "$f$ may be applied to the result of $g$ but not vice versa". It is also similar to information flow monitoring in the sense that the flow of data plays a central role. It should however be made clear that the focus in the study is not on arbitrary information flow but rather on direct and explicit flows.

The observable actions, i.e. the actions performed by the target program which are observed (and possibly rejected) by the execution monitor, is in this framework defined to be the external function calls. This is a common choice in runtime monitoring, c.f. [58, 95]. What is novel in our case is that the framework not only takes the function identifier and arguments into account, but also the full history of function calls (refered to as the *computational history*) that were made in order to compute the arguments. This means that the framework is able to enforce a policy stating for instance that $f$ may be applied to $g(x)$ but not to $h(x)$ even though $g$ and $h$ are extensionally equal. A more realistic example would be a policy stating that `sanitize` accepts any string as argument, while the function `runQuery` only accepts strings returned by `sanitize`.

All aspects of the framework that we present are both formalized rigorously at a low level and evaluated practically at a higher level.

**Formalization** The program model of the framework is formalized in terms of an applied $\lambda$-calculus which includes externally defined functions and values ranged over by $f \in \mathsf{F}$ and $c \in \mathsf{C}$ respectively. In addition to the usual rules the calculus provides a simple rule for applying the externally defined functions to arguments: $f\ c_1\ \ldots\ c_n \to [\![f(c_1, \ldots, c_n)]\!]$.

In order to provide a way of reasoning about function applications and observable actions we define a scheme for augmenting every value with its computational history. A value $c$ resulting from an application of $f$ to $g()$ is written $f(g()) : c$ and the annotation $f(g())$ is refered to as a *function application tree*. Furthermore we expose any observable actions by writing $t \xrightarrow{f(\tau_1,\ldots,\tau_n)} t'$ if $t$ reduces to $t'$ through an application of a function $f$ to $\tau_1 : c_1, \ldots, \tau_n : c_n$.

**Labels** The function application trees fully capture the computational history of every value present in the term. Such trees are convenient from a theoretical point of view, but keeping them around is not feasible in practice. If a policy for instance requires an argument to be the result of an even number of applications of `toggle`, the function application trees could grow indefinitely, despite the fact that a boolean value would suffice to maintain and describe the relevant computational history.

Due to this observation, we provide a variation of the calculus in which we have substituted the function application trees with *labels*. Intuitively the set of labels can be seen as an abstract domain over the function application trees, although technically speaking this not a formal requirement. Necessary conditions for a safe transition from function application trees to labels is presented and proven sound.

**Policies and enforcement** A policy is formalized as a predicate over the set of sequences of observable actions. In other words, a policy either *accepts* or *rejects* an execution $t_0 \xrightarrow{\tau_0} \ldots \xrightarrow{\tau_{n-1}} t_n$ based on the observable actions $\tau_0, \ldots, \tau_n$. At the level of the $\lambda$-calculus the enforcement mechanism is implemented simply as a premise of the function application rule, i.e. the rule for applying an externally defined function to some arguments is allowed to fire, only if it results in an observable action allowed by the policy.

In our work we focus on *local* and *subtree closed* policies. A local policy decides if a given sequence of observable actions is accepted or not by inspecting each action in isolation. For a (local) subtree closed policy the set of observable actions is also subtree closed which means that a policy can not for instance accept $f(g())$ but reject $g()$. The first property relieves the framework from dealing with global execution monitor state (a solution, which could easily be adapted to this framework, for dealing with such state has been presented in our earlier work, [41, 42]) and the second property allows us to provide results regarding completeness.

Furthermore we present the syntax and semantics of a language for describing policies. An example policy which makes sure that the data received from one host is not allowed to be sent to another host is presented in Figure 5.1. The semantics is conveniently defined in terms of tree automata over function

```
// Each time a socket is connected, it is given a taint unique for its host.
Socket.connect(String host) = fresh(host)

// The taint is copied to its input/output stream.
Socket.getInputStream(Socket this: A) = A
Socket.getOutputStream(Socket this: A) = A

// The taint is copied to any data read from such input/output stream.
InputStream.read(InputStream this: A)  = A

// Any data written to a (tainted) output stream must either be untainted or
// have the same taint as the output stream. (The result is always untained.)
OutputStream.write(OutputStream this : A, int x : B) = (
    A = 0 -> 0
    B = 0 -> 0
    A = B -> 0
)
```

Figure 5.1: Policy stating that data from one host may not be sent to another host.

application trees (i.e. trees constructed from function F nodes and constant C leaves ). The states $Q$ of the automaton correspond to the set of labels and each clause in the policy is translated into a set of automata moves $\Delta$. The automaton corresponding to the policy in Figure 5.1 is shown in Figure 5.2. As can be seen, it propagates the label of a socket to its input and output streams ($\Delta_1$ and $\Delta_2$), ensures that the label of the result of a read operation complies with the label of the input stream ($\Delta_3$), and ensures that the label of an argument to a write operation complies with the label of the output stream ($\Delta_4$).

A policy defined in this language is by construction both local and subtree closed.

**Practical evaluation and case studies** The approach presented here relies on (and therefore assumes the existence of) a dynamic taint tracking mechanism. There currently exists no dynamic taint tracking mechanism at the ABS level. However, Java bytecode can be generated from ABS code, and *TaintDroid* is an Android variant including a dynamic taint tracking mechanism that can run Java bytecode linked to the Android application programming interface (API). For practical reasons, the evaluation of the approach will then be conducted on Java code running on top of TaintDroid. The work presented in this section would apply similarly to ABS. The main work to be accomplished, for the application of the approach at the ABS level, would be the development of a dynamic taint tracking mechanism for ABS. The study of the effects on taint tracking of the language features specific to ABS should benefit from work already produced in Section 3.1. KTH plans to study the applicability of this task for a master project.

In the Java implementation of the framework for Android, we will rely on the existing taint tracking mechanism of TaintDroid in order to keep track of the labels of the values, and in order to initiate taints and enforce a policy (i.e. realize taint sources and taint sinks) we will rely on bytecode rewriting, or *inlining*, similar to the approach taken in our previous work. The source code of *TaintDroid* has already been partially adapted to our needs (the access methods for the taints have been exposed and the built in taint sources have been removed). The inlining tool is under development.

One case study (more adapted to the type of policies enforceable by our approach) has already been performed by means of manual inlining of the taint sources and sinks. The application in the study is a basic, open source, Internet banking application called *BankDroid*, and the policy is a slightly extended version of the policy in Figure 5.1.

$$
\begin{aligned}
Q &= \mathbb{N} \cup \{unit\} \\
\mathcal{F} &= \mathsf{C} \cup \mathsf{F} \\
Q_f &= Q \\
\Delta &= \{c \to unit(c) \mid c \in \mathsf{C}\} \\
&\cup \{\texttt{Socket.connect}(q(x)) \to fresh(x)(\texttt{Socket.connect}(x)) \mid q \in Q\} \\
&\cup \{\texttt{Socket.getInputStream}(q(x)) \to q(\texttt{Socket.getInputStream}(x)) \mid q \in Q\} \quad (\Delta_1) \\
&\cup \{\texttt{Socket.getOutputStream}(q(x)) \to q(\texttt{Socket.getOutputStream}(x)) \mid q \in Q\} \quad (\Delta_2) \\
&\cup \{\texttt{InputStream.read}(q(x)) \to q(\texttt{InputStream.read}(x)) \mid q \in Q\} \quad (\Delta_3) \\
&\cup \{\texttt{OutputStream.write}(q(x), q'(y)) \to unit(\texttt{OutputStream.write}(x, y)) \mid \\
&\qquad q, q' \in Q \wedge q = unit \vee q' = unit \vee q = q'\} \quad (\Delta_4)
\end{aligned}
$$

Figure 5.2: Tree automaton corresponding to the syntax in Figure 5.1.

# Chapter 6

# Automated construction of cryptographic protocol implementations

This chapter describes a tool that generates executable ABS implementations from high-level descriptions of security protocols. The generator takes as input a compact and high-level specification of a security protocol, written in a domain-specific language. The specification consists of a header and a body; informally, the header describes the roles (for instance, the initiator and the responder) and the participants (for instance, Alice as the initiator and Bob as the responder), whereas the body describes the sequence of messages between participants (for instance, Alice sends an encrypted message to Bob). The tool outputs an ABS program that implements the view of each role of the protocol (in the case of a two-party protocol, one view for the initiator and one view for the responder).

The workflow of the tool is described in Figure 6.2:

- in the first step, the input specification is expanded into a fully-fledged specification that makes explicit all assumptions about credentials, keys and cryptographic operations. Then, the intermediate specification is further refined into a specification in a temporal epistemic logic that includes a *belief* operator $\mathsf{Bel}_a(\cdot)$, a temporal operator *before*, and the additional action operator $\mathsf{Enforce}_a(\cdot)$. To conclude this step, this refinement is split into partial specifications that contain one view of the protocol for each role. Each such filter on the protocol contains only the assumptions relevant to the agent $A$, and the incoming and outgoing messages that $A$ participates in;

- in the second step, each view of the protocol is translated into a ABS code. Pleasingly, the translation makes extensive use of ABS features, and yields compact and readable code. For instance, the concurrency model of ABS, and in particular the notion of concurrent object group, provides a direct means to model and implement the behavior of independent components: each role in the component give rise to a concurrent object group. In a similar way, the ABS construct *await* provides a direct means to model that a component waits for an incoming message, or is allowed to proceed when it receives a matching message. The output ABS program is built on top of ABS libraries for modeling protocols, and for implementing communication over the network. We sometimes use the acronym ABS-sec to refer to the extension of ABS with these two libraries.

### Contents of the Chapter and online material

Section 6.1 describes the automatic generation of an implementation for a well-known security protocol, the Internet Key Exchange protocol (IKE). Section 6.2 reports on the experimental validation of the compiler; the examples of the generated code are available online. Section 6.3 provides pointers to related work and concludes with directions for future work.

$$
\begin{array}{l}
\mathsf{protocol}[\text{IKEversion2mac}, 0, \\
\quad \mathsf{role}(A) \wedge \mathsf{role}(B), \\
\quad \mathsf{role}(A) \wedge \mathsf{role}(B), \\
\quad \mathsf{role}(A), \\
\\
A \longrightarrow B : \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge \mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \\
B \longrightarrow A : \mathsf{Text}(\text{CryptoOffer IKE SA}, B) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \\
A \longrightarrow B : E[\text{SymKey} : \mathsf{Agent}(A) \wedge \text{AUTH-A} \wedge \mathsf{Text}(\text{CryptoOffer CHILD SA}, A)] \\
B \longrightarrow A : E[\text{SymKey} : \mathsf{Agent}(B) \wedge \text{AUTH-B} \wedge \mathsf{Text}(\text{CryptoOffer CHILD SA}, B)] \\
]
\end{array}
$$

Figure 6.1: Internet Key Exchange version2 with MAC.

## 6.1 An example: Internet Key Exchange

We illustrate the workflow of the tool with the Internet Key Exchange Protocol (IKE).

### 6.1.1 Protocol description

IKE establishes mutual authentication between two parties $A$ and $B$ communicating over the Internet and a security association (SA) including shared secret information that can be used to efficiently establish security associations for encrypting security payload or authentication in addition to establishment of cryptographic algorithms to be used [62, p. 3].

IKE has several variants; in this chapter, we consider the version based on using Message Authentication Code (MAC). Figure 6.1 provides the high-level specification that is input to the generator. The specification contains two main parts, a *protocol header*, and a *protocol body*. The protocol header contains the protocol name, session instance (book-keeping several potential instances of protocol sessions running in parallel), the complete set of roles in the protocol, the roles a particular agent are permitted to play, and finally the initiator role in the protocol. The protocol body contains a sequence of message clauses, containing sender receiver, and payload. For instance the final message:

$$B \longrightarrow A : E[\text{SymKey} : \mathsf{Agent}(B) \wedge \text{AUTH-B} \wedge \mathsf{Text}(\text{CryptoOffer CHILD SA}, B)]$$

states that the agent $B$ sends to agent $A$ a payload structure containing the encryption of the payload consisting of $B$'s name a composite data structure called AUTH-B, and a crypto-offer for B, encrypted with the symmetric key SymKey generated based on previous data-elements in the protocol session. Concatenation is represented by the $\wedge$ operator, while encryption of some payload Payload, using a Key, is written $E[\text{Key} : \text{Payload}]$.

The first message:

$$A \longrightarrow B : \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge \mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(ANONCE, A))$$

adopts similar conventions; it additionally introduces the constructions $\mathsf{n}(\cdot, \cdot)$ and $\mathsf{isNonce}(\cdot)$. The former $\mathsf{n}(\cdot, \cdot)$ is a term constructor; its first argument is a nonce variable, and its second argument is the originator of the nonce. The latter $\mathsf{isNonce}(\cdot)$ is a predicate stating that a term is a valid nonce.

### 6.1.2 Refinement and filtering of specifications

Refinement proceeds in two successive passes. First, macros are unfolded. In the case of IKE, the specification uses three macros: the composite symmetric key, denoted SymKey, an authentication component

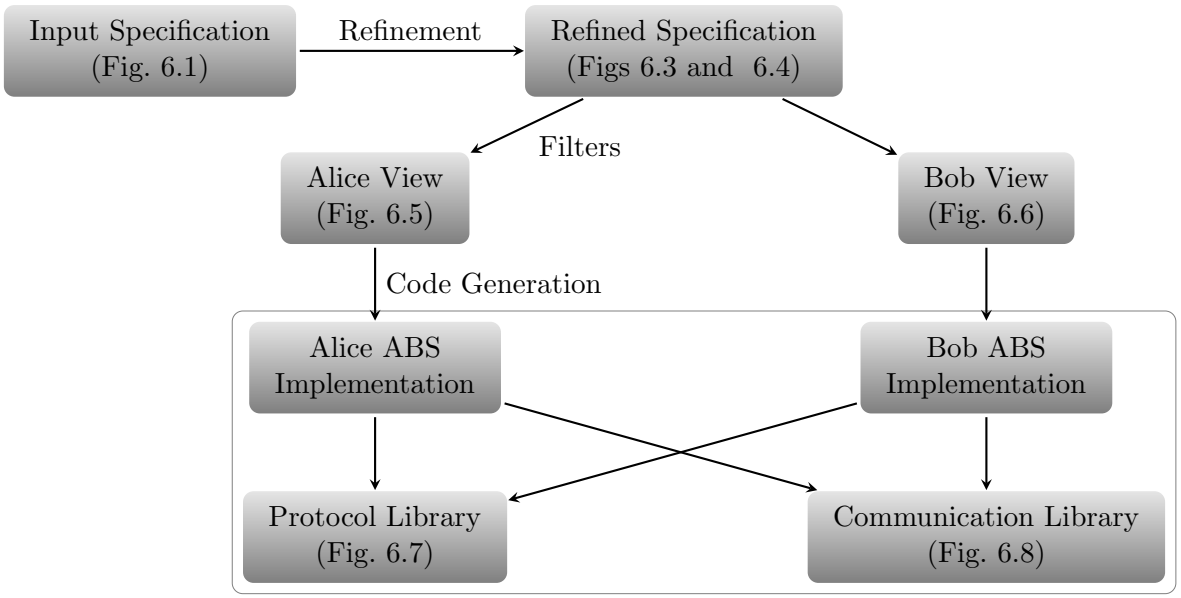


Figure 6.2: Overview of the module for generating implementations with references to figures.

for agent Alice ($A$), and one authentication component for agent Bob ($B$). The definition of the macros is given below:

$$\begin{array}{l}\mathrm{SymKey} = \mathsf{key}(\mathsf{Hash}[\mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \wedge \\ \qquad \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge \mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B))))])\\ \\ \text{AUTH-A} = \\ \quad \mathsf{Hash}[\mathsf{isKey}(\mathsf{key}(\mathsf{s}, A, B, PSK)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge \mathsf{isNonce}(\mathsf{n}(KEYA, A)) \\ \qquad \wedge \mathsf{isNonce}\,(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B))] \\ \\ \text{AUTH-B} = \\ \quad \mathsf{Hash}[\mathsf{isKey}(\mathsf{key}(\mathsf{s}, A, B, PSK)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, B) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B)) \\ \qquad \wedge \mathsf{isNonce}\,(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B))]\end{array}$$

Second, the formal specification is refined automatically into one single specification that contains the local assumptions about cryptographic operations. The refinement step extends ideas from [57] and uses an epistemic temporal logic that includes the *belief* operator $\mathsf{Bel}_A(\cdot)$, the temporal operator *before*, and the action operator *enforce* $\mathsf{Enforce}_A(\cdot)$.

- The belief operator is used to model the pre- and post-conditions for each message exchange. In particular, it is used on the sender side to specify the components that must be known to form the message, and on the receiver side to specify the knowledge gained by the receiver upon reception of the message.
- The enforce operator guarantees that participants behave according to the protocol. In particular, it is used on the sender side to enforce that data-elements are fresh (e.g. nonces or time-stamps), or valid (e.g. hashes or cipher-texts), and on the receiver side to enforce that the receiver compares hashes and decrypt messages elements if it possesses the appropriate keys.

The refined view of the protocol is depicted in Figures 6.3 and 6.4.

Filtering splits the specification into pieces containing one view of the protocol for each role. Each such filter on the protocol contains only the assumptions relevant to the agent $A$, and the relevant incoming and

outgoing messages that $A$ participates in. In the case of IKE, the *Alice*-filter is described in Figure 6.5, and the *Bob*-filter is depicted in Figure 6.6.

### 6.1.3 Generation of ABS code

Generation outputs for each agent in the protocol an ABS class that realizes its behaviour. The generated ABS class starts with a prelude that collects variable declarations for keys, credentials, guards, miscellaneous elementary payload intermediate cryptographic computations and messages. The prelude introduces a Zombie agent used for initialising some of the general functions (see the functions getReceiver and getSender in Figure 6.8). The ABS code of the prelude for IKE thus includes the following snippet:

```
// The protocol: IKE version 2 mac
 {
 Agent zombieAgent =  new cog ZombieAgent(AgentName("TestZombie") )  ;
    Network network  ;
    network =  new cog Network(zombieAgent)  ;
    Agent objectA ;
    objectA = new cog Aclass (AgentVar( "A" ) , network) ;
    network!register(AgentVar( "A" ), objectA)  ;
    Agent objectB ;
    objectB = new cog Bclass (AgentVar( "B" ) , network) ;
    network!register(AgentVar( "B" ), objectB)  ;
  }
```

The ABS class additionally models the transactions (events) performed by the agent as sequential events in the run-method of the class. There are three different kinds of events.

- *Sending out messages.* This is modelled using the ABS library for communications that is given in Figure 6.8.
- *Waiting for incoming messages.* The modeling is significantly more intricate than for sending messages. Indeed, the agent might receive messages not intended to be part of the current protocol session or protocol messages might be interleaved. The agents connected to the network subscribe to the network by a receive method that administrates the collection of await for message patterns that are valid patterns in a protocol session. Each receive event in the protocol is accompanied by an ABS *await* clause, that blocks the succeeding steps in the protocol. The receive method tests the payload structure of incoming messages and checks if it matches some of the message patterns that the receiver is waiting for. If it matches, then a boolean placeholder for the particular guard is set to true and the await released.
- *Constructing and reconstructing data using cryptographic operations.* Constructing and decomposing message payload using cryptographic operations is represented by assignments using the ABS library for protocols that is given in Figure 6.7.

In more detail, the generation algorithm proceeds in two steps. First, it recursively traverses the participant view of the protocol and translates each element into a ABS code snippet. Second, it organises the appropriate code snippets in the correct location according to ABS language specification.

1. For a given role $A$ (agent or component) in the protocol, all local actions, assumptions, send and receive events for $A$ are collected into lists of strings containing information about:
   (a) the next item in the refined specification;
   (b) receive-flags used to signal messages received (releasing awaits);
   (c) receive-patterns;
   (d) receive-method;

(e) ABS declarations;
(f) elements in the run method;
(g) class signature code (class name, references and implementation of interfaces).

2. The resulting tuple of ABS-code pieces structured as step 1, is then put together by flattening string-elements from step 1 by concatenating them in the following order:

(a) receive patterns (functions);
(b) class signature code (the start of the class definition);
(c) the boolean receive-flags;
(d) ABS declarations of all variables used later in the run method;
(e) the start of the run method;
(f) assignments and events in the run method;
(g) receive-methods;
(h) end of class syntax.

To illustrate the algorithm we consider the first two clauses in Alice's local view on the protocol (from Figure 6.5) and how they are translated into ABS:

$$\begin{array}{c} \vdots \quad \vdots \\ \mathsf{Bel}_A(\mathsf{Text}(\text{CryptoOffer IKE SA}, A)) \\ \mathsf{Enforce}_A(\mathsf{Bel}_A(\mathsf{newNonce}(\mathsf{n}(KEYA, A)))) \\ \vdots \quad \vdots \end{array}$$

The first clause is an assumption about a piece of data that Alice should have. In the Alice class this is translated into an ABS declaration and an assignment in the run method. A variable-name is created to be used both places. Hence the assumption $\mathsf{Bel}_A(\mathsf{Text}(\text{CryptoOffer IKE SA}, A))$ is translated into the clause in the ABS protocol language:

```
PayloadElement textCryptoOfferIKESAA = UndefinedElem ;
```

The declaration is located in the beginning of the class. The variable textCryptoOfferIKESAA is made by flattening the constructor of the payload (Text) and all its parameters (the text-string CryptoOffer IKE SA and the agent-variable $A$ into one continous sequence of letters. In the run-method the variable is assigned a value:

```
textCryptoOfferIKESAA = TextOrigin("CryptoOffer␣IKE␣SA", AgentVar("A")) ;
```

The second clause is an assertion about a fresh nonce that is created by Alice to be used as one of the building blocks in the composite symmetric key to be constructed during the protocol session.

```
PayloadElement newnonceKEYAA = UndefinedElem ;
```

Freshness is specified by the New(...) operator. The predicate Nonce takes two arguments, a nonce variable "KEYA" and an agent variable "A".

```
newnonceKEYAA = New(Nonce( NonceVar ("KEYA"), AgentVar("A") )) ;
```

After the second step in the generation algorithm, the final run-method is of the form:

```
Unit run(){ await(network != null) ;

  textCryptoOfferIKESAA = TextOrigin("CryptoOffer␣IKE␣SA", AgentVar("A")) ;

newnonceKEYAA = New(Nonce( NonceVar ("KEYA"), AgentVar("A") )) ;
```

Construction of real nonces is not given a concrete implementation. The same applies to timestamps and keys. A symbolic interpretation is given in the execution environment by assigning each new fresh nonce a successive number. Any practical use of the algorithm for automated code generation relies on a healthy implementation of nonces, timestamps and keys in addition to deployment of sound cryptographic algorithms.

## 6.2 Evaluation

The implementation has been successfully evaluated on a large collection of protocols, including all protocols from the Clark-Jacob library [35], samples protocols from the online library AVISPA [81], and several more recent protocols like the Transport Layer Security (TLS) and Internet Key Exchange (IKE). Figure 6.9 reports data of both the protocol specification and the automatically generated ABS code: From the specification we derive data about the number of *roles* involved (**#Roles**), the total number *messages* (**# Msg**), the *length* of the refined specification (**#Refined**), the nesting depth of cryptographic operations (**#Depth**). The size of the automatically generated ABS code is given by the number of code-lines written by the generator (**# lines**) and number of bytes (**# lines**).

$$
\begin{array}{l}
\mathsf{protocol}[\mathrm{IKEversion2mac}, 0, \\
\quad \mathsf{role}(A) \wedge \mathsf{role}(B), \quad \mathsf{role}(A) \wedge \mathsf{role}(B), \quad \mathsf{role}(A), \\
\\
\mathsf{Bel}_A(\mathsf{Text}(\mathrm{CryptoOffer\ IKE\ SA}, A)) \\
\mathsf{Enforce}_A(\mathsf{Bel}_A(\mathsf{newNonce}(\mathsf{n}(KEYA, A)))) \\
\mathsf{Enforce}_A(\mathsf{Bel}_A(\mathsf{newNonce}(\mathsf{n}(ANONCE, A)))) \\
\mathsf{Bel}_A(\mathsf{Agent}(B)) \\
A \longrightarrow B : \mathsf{Text}(\mathrm{CryptoOffer\ IKE\ SA}, A) \wedge \mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \\
\mathsf{Bel}_B(\mathsf{Agent}(A)) \\
\mathsf{Bel}_B(\mathsf{Text}(\mathrm{CryptoOffer\ IKE\ SA}, A)) \\
\mathsf{Bel}_B(\mathsf{isNonce}(\mathsf{n}(KEYA, A))) \\
\mathsf{Bel}_B(\mathsf{isNonce}(\mathsf{n}(ANONCE, A))) \\
\mathsf{Bel}_B(\mathsf{Text}(\mathrm{CryptoOffer\ IKE\ SA}, B)) \\
\mathsf{Enforce}_B(\mathsf{Bel}_B(\mathsf{newNonce}(\mathsf{n}(KEYB, B)))) \\
\mathsf{Enforce}_B(\mathsf{Bel}_B(\mathsf{newNonce}(\mathsf{n}(BNONCE, B)))) \\
B \longrightarrow A : \mathsf{Text}(\mathrm{CryptoOffer\ IKE\ SA}, B) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \\
\mathsf{Bel}_A(\mathsf{Text}(\mathrm{CryptoOffer\ IKE\ SA}, B)) \\
\mathsf{Bel}_A(\mathsf{isNonce}(\mathsf{n}(KEYB, B))) \\
\mathsf{Bel}_A(\mathsf{isNonce}(\mathsf{n}(BNONCE, B))) \\
\mathsf{Bel}_A(\mathsf{Agent}(A)) \\
\mathsf{Enforce}_A(\mathsf{Bel}_A(\mathsf{newKey}(\mathsf{key}(\mathsf{s}, A, B, PSK)))) \\
\mathsf{Enforce}_A(\mathsf{Bel}_A(\mathsf{Hash}[\mathsf{isKey}(\mathsf{key}(\mathsf{s}, A, B, PSK)) \wedge \mathsf{Text}(\mathrm{CryptoOffer\ IKE\ SA}, A) \wedge \\
\qquad \mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B))])) \\
\mathsf{Bel}_A(\mathsf{Text}(\mathrm{CryptoOffer\ CHILD\ SA}, A)) \\
\mathsf{Enforce}_A(\mathsf{Bel}_A(\mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B)))))) \\
\mathsf{Enforce}_A(\mathsf{Bel}_A(\mathsf{Hash}[\mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \wedge \\
\qquad \mathsf{Text}(\mathrm{CryptoOffer\ IKE\ SA}, A) \wedge \mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B))))])) \\
\mathsf{Enforce}_A(\mathsf{Bel}_A(\mathsf{isKey}(\mathsf{key}(\mathsf{Hash}[\mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \wedge \\
\qquad \mathsf{Text}(\mathrm{CryptoOffer\ IKE\ SA}, A) \wedge \mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B))))])))) \\
\mathsf{Enforce}_A(\mathsf{Bel}_A(E[\mathsf{key}(\mathsf{Hash}[\mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \wedge \\
\qquad \mathsf{Text}(\mathrm{CryptoOffer\ IKE\ SA}, A) \wedge \mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B))))]) : \mathsf{Agent}(A) \wedge \\
\qquad \mathsf{Hash}[\mathsf{isKey}(\mathsf{key}(\mathsf{s}, A, B, PSK)) \wedge \mathsf{Text}(\mathrm{CryptoOffer\ IKE\ SA}, A) \wedge \mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \\
\qquad \mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B))] \wedge \mathsf{Text}(\mathrm{CryptoOffer\ CHILD\ SA}, A)])) \\
\end{array}
$$

Figure 6.3: Refined version of the complete specification (part 1).

## 6.3 Concluding remarks

We have presented a tool that generates automatically ABS implementations from high-level specifications of cryptographic libraries. The tool has been evaluated successfully on a large set of benchmarks.

### 6.3.1 Related work

It is not possible to review the state-of-the-art in cryptographic protocol generation and verification. We limit ourselves to pointing to some relevant publications. In the security protocol literature there have been two major topics of investigation: analysis of specifications and mining of implementations. The analyses of security protocol specifications have typically been using manual or automated methods - merely considering design ideas. In the late 1990s, new attacks were found on classical cryptographic protocols, originating

from the well known work by Gavin Lowe [70], who used the CSP-based model checking tool FDR [29] to find a previously unknown attack on the Needham-Schroeder public key authentication protocol. This inspired many research groups [39, 71, 26, 72, 22, 5, 4, 27, 91, 38, 28, 33, 48] to build special purpose model checking tools that could be used to analyse large collections of security protocols [35, 50, 81]. In practice implementations deviate from specifications on many points: the level of detail in payload content, which messages are used, etc. Therefore several researchers have been working on extracting high level views on low level implementations using mining techniques [2, 56].

More recently, there has been some research into generating JAVA code from the Extensible Spi-calculus [8], [30]. Our method differs in that no manual process of ascribing types to the cryptographic credentials is needed. Moreover, our method does not impose any limit on the number of roles, messages and encryption nesting.

### 6.3.2 Further research

There are several important topics for further research, including:

- There are several tools that can verify the security of protocol specifications. In the current chapter we have not shown that the transformation from specifications to implementations preserve security. Having such a proof would give us provably secure implementations.
- Generating JAVA or C#: Java code can be generated from ABS automatically. However, there is currently no formal proof that the compilation from ABS to JAVA is semantics-preserving, and there is no mechanism to link the symbolic cryptography used by ABS programs with JAVA cryptographic APIs. Being able to generate JAVA code that is provably equivalent to the ABS implementations (under the assumptions of perfect cryptography) would narrow the gap between provably secure implementations and deployed code.

$A \longrightarrow B : E[\mathsf{key}(\mathsf{Hash}[\mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \wedge$
$\quad \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge \mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B))))]) : \mathsf{Agent}(A) \wedge$
$\quad \mathsf{Hash}[\mathsf{isKey}(\mathsf{key}(\mathsf{s}, A, B, PSK)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge \mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge$
$\quad \mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B))] \wedge \mathsf{Text}(\text{CryptoOffer CHILD SA}, A)]$
$\mathsf{Enforce}_B(\mathsf{Bel}_B(\mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B))))))$
$\mathsf{Enforce}_B(\mathsf{Bel}_B(\mathsf{Hash}[\mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge$
$\quad \mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B))))]))$
$\mathsf{Enforce}_B(\mathsf{Bel}_B(\mathsf{isKey}(\mathsf{key}($
$\quad \mathsf{Hash}[\mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge$
$\quad \mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B))))])))$
$\mathsf{Enforce}_B(\mathsf{Bel}_B(D[\mathsf{key}(\mathsf{Hash}[\mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge$
$\quad \mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B))))]) :$
$\quad\quad E[\mathsf{key}(\mathsf{Hash}[\mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge$
$\quad \mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B))))]) : \mathsf{Agent}(A) \wedge$
$\quad \mathsf{Hash}[\mathsf{isKey}(\mathsf{key}(\mathsf{s}, A, B, PSK)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge \mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge$
$\quad \mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B))] \wedge \mathsf{Text}(\text{CryptoOffer CHILD SA}, A)]]))$
$\mathsf{Bel}_B(\mathsf{Hash}[\mathsf{isKey}(\mathsf{key}(\mathsf{s}, A, B, PSK)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge \mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge$
$\quad \mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B))])$
$\mathsf{Bel}_B(\mathsf{Text}(\text{CryptoOffer CHILD SA}, A))$
$\mathsf{Bel}_B(\mathsf{Agent}(B))$
$\mathsf{Enforce}_B(\mathsf{Bel}_B(\mathsf{newKey}(\mathsf{key}(\mathsf{s}, A, B, PSK))))$
$\mathsf{Enforce}_B(\mathsf{Bel}_B(\mathsf{Hash}[\mathsf{isKey}(\mathsf{key}(\mathsf{s}, A, B, PSK)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, B) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B)) \wedge$
$\quad \mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B))]))$
$\mathsf{Bel}_B(\mathsf{Text}(\text{CryptoOffer CHILD SA}, B))$
$\mathsf{Enforce}_B(\mathsf{Bel}_B(E[\mathsf{key}(\mathsf{Hash}[\mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge$
$\quad \mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B))))]) : \mathsf{Agent}(B) \wedge$
$\quad \mathsf{Hash}[\mathsf{isKey}(\mathsf{key}(\mathsf{s}, A, B, PSK)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, B) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B)) \wedge$
$\quad \mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B))] \wedge \mathsf{Text}(\text{CryptoOffer CHILD SA}, B)]))$
$B \longrightarrow A : E[\mathsf{key}(\mathsf{Hash}[\mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge$
$\quad \mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B))))]) : \mathsf{Agent}(B) \wedge$
$\quad \mathsf{Hash}[\mathsf{isKey}(\mathsf{key}(\mathsf{s}, A, B, PSK)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, B) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B)) \wedge$
$\quad \mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B))] \wedge \mathsf{Text}(\text{CryptoOffer CHILD SA}, B)]$
$\mathsf{Enforce}_A(\mathsf{Bel}_A(D[\mathsf{key}(\mathsf{Hash}[\mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge$
$\quad \mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B))))]) :$
$\quad E[\mathsf{key}(\mathsf{Hash}[\mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge$
$\quad \mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B))))]) : \mathsf{Agent}(B) \wedge$
$\quad \mathsf{Hash}[\mathsf{isKey}(\mathsf{key}(\mathsf{s}, A, B, PSK)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, B) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B)) \wedge$
$\quad \mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B))] \wedge \mathsf{Text}(\text{CryptoOffer CHILD SA}, B)]]))$
$\mathsf{Bel}_A(\mathsf{Hash}[\mathsf{isKey}(\mathsf{key}(\mathsf{s}, A, B, PSK)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, B) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B)) \wedge$
$\quad \mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B))])$
$\mathsf{Bel}_A(\mathsf{Text}(\text{CryptoOffer CHILD SA}, B))$

Figure 6.4: Refined version of the complete specification (part 2).

$\mathsf{protocol}[\text{IKEversion2mac}, 0,$
$\quad \mathsf{role}(A) \wedge \mathsf{role}(B), \quad \mathsf{role}(A) \wedge \mathsf{role}(B), \quad \mathsf{role}(A),$

$\mathsf{Bel}_A(\mathsf{Text}(\text{CryptoOffer IKE SA}, A))$
$\mathsf{Enforce}_A(\mathsf{Bel}_A(\mathsf{newNonce}(\mathsf{n}(KEYA, A))))$
$\mathsf{Enforce}_A(\mathsf{Bel}_A(\mathsf{newNonce}(\mathsf{n}(ANONCE, A))))$
$\mathsf{Bel}_A(\mathsf{Agent}(B))$
$A \longrightarrow B : \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge \mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(ANONCE, A))$
$B \longrightarrow A : \mathsf{Text}(\text{CryptoOffer IKE SA}, B) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B))$
$\mathsf{Bel}_A(\mathsf{Text}(\text{CryptoOffer IKE SA}, B))$
$\mathsf{Bel}_A(\mathsf{isNonce}(\mathsf{n}(KEYB, B)))$
$\mathsf{Bel}_A(\mathsf{isNonce}(\mathsf{n}(BNONCE, B)))$
$\mathsf{Bel}_A(\mathsf{Agent}(A))$
$\mathsf{Enforce}_A(\mathsf{Bel}_A(\mathsf{newKey}(\mathsf{key}(\mathsf{s}, A, B, PSK))))$
$\mathsf{Enforce}_A(\mathsf{Bel}_A(\mathsf{Hash}[\mathsf{isKey}(\mathsf{key}(\mathsf{s}, A, B, PSK)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge$
$\quad \mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B))]))$
$\mathsf{Bel}_A(\mathsf{Text}(\text{CryptoOffer CHILD SA}, A))$
$\mathsf{Enforce}_A(\mathsf{Bel}_A(\mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B))))))$
$\mathsf{Enforce}_A(\mathsf{Bel}_A(\mathsf{Hash}[\mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \wedge$
$\quad \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge \mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B))))]))$
$\mathsf{Enforce}_A(\mathsf{Bel}_A(\mathsf{isKey}(\mathsf{key}(\mathsf{Hash}[\mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \wedge$
$\quad \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge \mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B))))]))))$
$\mathsf{Enforce}_A(\mathsf{Bel}_A(E[\mathsf{key}(\mathsf{Hash}[\mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \wedge$
$\quad \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge \mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B))))]) : \mathsf{Agent}(A) \wedge$
$\quad \mathsf{Hash}[\mathsf{isKey}(\mathsf{key}(\mathsf{s}, A, B, PSK)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge \mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge$
$\quad \mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B))] \wedge \mathsf{Text}(\text{CryptoOffer CHILD SA}, A)]))$
$A \longrightarrow B : E[\mathsf{key}(\mathsf{Hash}[\mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \wedge$
$\quad \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge \mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B))))]) : \mathsf{Agent}(A) \wedge$
$\quad \mathsf{Hash}[\mathsf{isKey}(\mathsf{key}(\mathsf{s}, A, B, PSK)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge \mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge$
$\quad \mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B))] \wedge \mathsf{Text}(\text{CryptoOffer CHILD SA}, A)]$
$B \longrightarrow A : E[\mathsf{key}(\mathsf{Hash}[\mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge$
$\quad \mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B))))]) : \mathsf{Agent}(B) \wedge$
$\quad \mathsf{Hash}[\mathsf{isKey}(\mathsf{key}(\mathsf{s}, A, B, PSK)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, B) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B)) \wedge$
$\quad \mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B))] \wedge \mathsf{Text}(\text{CryptoOffer CHILD SA}, B)]$
$\mathsf{Enforce}_A(\mathsf{Bel}_A(D[\mathsf{key}(\mathsf{Hash}[\mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge$
$\quad \mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B))))]) :$
$\quad E[\mathsf{key}(\mathsf{Hash}[\mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge$
$\quad \mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B))))]) : \mathsf{Agent}(B) \wedge$
$\quad \mathsf{Hash}[\mathsf{isKey}(\mathsf{key}(\mathsf{s}, A, B, PSK)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, B) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B)) \wedge$
$\quad \mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B))] \wedge \mathsf{Text}(\text{CryptoOffer CHILD SA}, B)]]))$
$\mathsf{Bel}_A(\mathsf{Hash}[\mathsf{isKey}(\mathsf{key}(\mathsf{s}, A, B, PSK)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, B) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B)) \wedge$
$\quad \mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B))])$
$\mathsf{Bel}_A(\mathsf{Text}(\text{CryptoOffer CHILD SA}, B))$

Figure 6.5: Alice's local view on the protocol.

$$
\begin{array}{l}
\mathsf{protocol}[\mathrm{IKEversion2mac}, 0, \\
\quad \mathsf{role}(A) \wedge \mathsf{role}(B), \quad \mathsf{role}(A) \wedge \mathsf{role}(B), \quad \mathsf{role}(A), \\
\\
A \longrightarrow B : \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge \mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \\
\mathsf{Bel}_B(\mathsf{Agent}(A)) \\
\mathsf{Bel}_B(\mathsf{Text}(\text{CryptoOffer IKE SA}, A)) \\
\mathsf{Bel}_B(\mathsf{isNonce}(\mathsf{n}(KEYA, A))) \\
\mathsf{Bel}_B(\mathsf{isNonce}(\mathsf{n}(ANONCE, A))) \\
\mathsf{Bel}_B(\mathsf{Text}(\text{CryptoOffer IKE SA}, B)) \\
\mathsf{Enforce}_B(\mathsf{Bel}_B(\mathsf{newNonce}(\mathsf{n}(KEYB, B)))) \\
\mathsf{Enforce}_B(\mathsf{Bel}_B(\mathsf{newNonce}(\mathsf{n}(BNONCE, B)))) \\
B \longrightarrow A : \mathsf{Text}(\text{CryptoOffer IKE SA}, B) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \\
A \longrightarrow B : E[\mathsf{key}(\mathsf{Hash}[\mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \wedge \\
\qquad \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge \mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B))))]) : \mathsf{Agent}(A) \wedge \\
\qquad \mathsf{Hash}[\mathsf{isKey}(\mathsf{key}(\mathsf{s}, A, B, PSK)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge \mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \\
\qquad \mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B))] \wedge \mathsf{Text}(\text{CryptoOffer CHILD SA}, A)] \\
\mathsf{Enforce}_B(\mathsf{Bel}_B(\mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B)))))) \\
\mathsf{Enforce}_B(\mathsf{Bel}_B(\mathsf{Hash}[\mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge \\
\qquad \mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B))))])) \\
\mathsf{Enforce}_B(\mathsf{Bel}_B(\mathsf{isKey}(\mathsf{key}( \\
\qquad \mathsf{Hash}[\mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge \\
\qquad \mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B))))])))) \\
\mathsf{Enforce}_B(\mathsf{Bel}_B(D[\mathsf{key}(\mathsf{Hash}[\mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge \\
\qquad \mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B))))]) : \\
\qquad\quad E[\mathsf{key}(\mathsf{Hash}[\mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge \\
\qquad \mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B))))]) : \mathsf{Agent}(A) \wedge \\
\qquad \mathsf{Hash}[\mathsf{isKey}(\mathsf{key}(\mathsf{s}, A, B, PSK)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge \mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \\
\qquad \mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B))] \wedge \mathsf{Text}(\text{CryptoOffer CHILD SA}, A)]])) \\
\mathsf{Bel}_B(\mathsf{Hash}[\mathsf{isKey}(\mathsf{key}(\mathsf{s}, A, B, PSK)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge \mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \\
\qquad \mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B))]) \\
\mathsf{Bel}_B(\mathsf{Text}(\text{CryptoOffer CHILD SA}, A)) \\
\mathsf{Bel}_B(\mathsf{Agent}(B)) \\
\mathsf{Enforce}_B(\mathsf{Bel}_B(\mathsf{newKey}(\mathsf{key}(\mathsf{s}, A, B, PSK)))) \\
\mathsf{Enforce}_B(\mathsf{Bel}_B(\mathsf{Hash}[\mathsf{isKey}(\mathsf{key}(\mathsf{s}, A, B, PSK)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, B) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B)) \wedge \\
\qquad \mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B))])) \\
\mathsf{Bel}_B(\mathsf{Text}(\text{CryptoOffer CHILD SA}, B)) \\
\mathsf{Enforce}_B(\mathsf{Bel}_B(E[\mathsf{key}(\mathsf{Hash}[\mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge \\
\qquad \mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B))))]) : \mathsf{Agent}(B) \wedge \\
\qquad \mathsf{Hash}[\mathsf{isKey}(\mathsf{key}(\mathsf{s}, A, B, PSK)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, B) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B)) \wedge \\
\qquad \mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B))] \wedge \mathsf{Text}(\text{CryptoOffer CHILD SA}, B)])) \\
B \longrightarrow A : E[\mathsf{key}(\mathsf{Hash}[\mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, A) \wedge \\
\qquad \mathsf{isKey}(\mathsf{key}(\mathsf{isNonce}(\mathsf{n}(KEYA, A)) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B))))]) : \mathsf{Agent}(B) \wedge \\
\qquad \mathsf{Hash}[\mathsf{isKey}(\mathsf{key}(\mathsf{s}, A, B, PSK)) \wedge \mathsf{Text}(\text{CryptoOffer IKE SA}, B) \wedge \mathsf{isNonce}(\mathsf{n}(KEYB, B)) \wedge \\
\qquad \mathsf{isNonce}(\mathsf{n}(ANONCE, A)) \wedge \mathsf{isNonce}(\mathsf{n}(BNONCE, B))] \wedge \mathsf{Text}(\text{CryptoOffer CHILD SA}, B) \\
]
\end{array}
$$

Figure 6.6: Bob's local view on the protocol.

```
module ABSProtocols;
export *;

type Variable = String ;
type Descriptor = String;

type Counter = Int;
type Value = Int;

data AgentTerm = AgentName(String) | AgentVar(Variable);
data NonceTerm = NonceValue(Int) | NonceVar(Variable);
data TimeTerm = TimeValue(Int) | TimeVar(Variable);

data KeyTerm = KeyValue(Int) | KeyVar(Variable);
data KeyCounterTerm = KeyCounterValue(Int) | KeyCounterVar(Variable);

data  Key =  KeyComposite(Payload)
           | KeyPKIcomp(PKIRole, AgentTerm, KeyPolicy, KeyCounterTerm)
           | KeyPKI(PKIRole, AgentTerm)
           | KeySymmetric(AgentTerm, AgentTerm)
           | KeySymmetricFresh(AgentTerm, AgentTerm, KeyTerm);

data PKIRole = Private | Public;
data KeyPolicy = AUTHENTICATE | INTEGRITY | CONFIDENTIALITY;

type Payload = List<PayloadElement>;

data PayloadElement =
          Nonce(NonceTerm, AgentTerm)
      | TimeStamp(TimeTerm, AgentTerm)
      | Current(TimeTerm, AgentTerm)
      | Agent(AgentTerm)
      | Key(Key)
      | Text(String)
      | TextOrigin(String, AgentTerm)
      | New(PayloadElement)
      | Hash(Payload)
      | HMAC(Key, Payload)
      | Encrypt(Key, Payload)
      | Decrypt(Key, Payload)
      | UndefinedElem;

data Role = Role(AgentTerm) ;

data ProtocolClause = Message(AgentTerm, AgentTerm, Payload)
        | Believes(AgentTerm, Payload)
        | Enforce(AgentTerm, Payload)
        | UndefinedClause;

type ProtocolName = String;
type SessionCounter = Int;
type TotalRoles = List<Role>;
type AgentRoles = List<Role>;
type StartRole = Role;
type ProtocolBody = List<ProtocolClause>;

data ProtocolSpecification =
     Protocol(ProtocolName,
              SessionCounter,
              TotalRoles,
              AgentRoles,
              StartRole, ProtocolBody);
```

Figure 6.7: The ABS protocol language.

```
module ExecutionEnvironment;
export *;
import * from ABSProtocols;

data NonceGenerator = NonceGenerator(Int);
data TimestampGenerator = TimestampGenerator(Int);

interface Network {
   Unit send(ProtocolClause msg);
   Unit register(AgentTerm agentname, Agent agent );
}

 def Agent getReceiver(ProtocolClause msg,
     Map<AgentTerm, Agent> connections, Agent zombieAgent)
 = case msg  {
          Message(senderName, receiverName, payload)    =>
                lookupDefault(connections, receiverName,  zombieAgent );};

def Agent getSender(ProtocolClause msg,
     Map<AgentTerm, Agent>  connections, Agent zombieAgent )
 = case msg  {
         Message(senderName, receiverName, payload)     =>
                lookupDefault(connections, senderName,  zombieAgent );
              };

type     Connectors = Map< AgentTerm, Agent >;
type     MsgContainer = Set< ProtocolClause >;

class Network(Agent zombieAgent)  implements Network {

   MsgContainer msgcontainer = EmptySet;
   Connectors connectors = EmptyMap;

Unit send(ProtocolClause msg)
  {
    msgcontainer = insertElement(msgcontainer, msg);
    Agent receiverAgent = getReceiver(msg, connectors, zombieAgent);
    Agent senderAgent   = getSender  (msg, connectors, zombieAgent);
  if
   (and(not(receiverAgent == zombieAgent), not(senderAgent == zombieAgent)) )
      {msgcontainer = remove(msgcontainer, msg);
       receiverAgent!receive(msg) ;}
                }

   Unit register(AgentTerm agentname, Agent agent)
        {
                connectors = insert( connectors, Pair(agentname, agent));
                this.refreshSending( agentname, msgcontainer);
        }

   Unit refreshSending(AgentTerm agentname, MsgContainer msgcontainer)
  {while (hasNext(msgcontainer) )
   {Pair < MsgContainer, ProtocolClause > decomposeSet =
                                        next (msgcontainer);
        ProtocolClause msgElement = snd (decomposeSet );
        msgcontainer = fst (decomposeSet );
        this.send(msgElement );
      } } }

interface Agent {
Unit receive(ProtocolClause msg); }

class ZombieAgent(AgentTerm name) implements Agent {
   Unit receive(ProtocolClause msg) {}   }
```

Figure 6.8: Execution environment for protocol implementations.

| | PROSA specification | | | | ABS code | |
|---|---|---|---|---|---|---|
| **Protocol name** | **Roles** | **# Msg** | **#Refined** | **#Depth** | **# lines** | **# bytes** |
| *Symmetric key* | | | | | | |
| ISO One pass unilateral | 2 | 1 | 14 | 1 | 161 | 5335 |
| ISO Two pass unilateral | 2 | 2 | 17 | 1 | 206 | 6885 |
| ISO Two pass mutual | 2 | 2 | 24 | 1 | 244 | 9027 |
| ISO Three pass mutual | 2 | 3 | 27 | 1 | 291 | 11364 |
| Non-reversible func. | 2 | 5 | 27 | 2 | 308 | 11944 |
| Andrew Secure RPC | 2 | 5 | 30 | 1 | 345 | 12926 |
| ISO One pass unilat. CCF | 2 | 1 | 12 | 1 | 153 | 4978 |
| ISO Two pass unilat. CCF | 2 | 2 | 14 | 1 | 193 | 6381 |
| ISO Two pass mutual CCF | 2 | 2 | 20 | 1 | 227 | 8536 |
| ISO Three pass CCF | 2 | 3 | 22 | 1 | 269 | 10506 |
| *Symmetric key with TTP* | | | | | | |
| Needham Schroeder sym. | 3 | 6 | 34 | 2 | 428 | 17026 |
| Denning Sacco | 3 | 5 | 28 | 1 | 323 | 13681 |
| Otway Rees | 3 | 5 | 36 | 1 | 423 | 19336 |
| Wide Mouthed Frog | 3 | 3 | 25 | 1 | 270 | 10497 |
| Yahalom | 3 | 5 | 35 | 1 | 403 | 17233 |
| Carlsen's Secret Key | 3 | 7 | 37 | 1 | 447 | 20244 |
| ISO Four pass | 3 | 6 | 55 | 1 | 472 | 24301 |
| ISO Five pass | 3 | 7 | 55 | 1 | 529 | 26047 |
| Woo Lam Pi | 3 | 5 | 25 | 2 | 375 | 11798 |
| Kerberos (ver. 5) | 4 | 6 | 69 | 1 | 682 | 36767 |
| Neuman Stubblebine | 3 | 8 | 49 | 1 | 583 | 26226 |
| Kehne Langendorfer | 3 | 10 | 53 | 1 | 645 | 30081 |
| Kao Chow | 3 | 8 | 46 | 1 | 474 | 25531 |
| *Public key* | | | | | | |
| ISO PKI One pass | 2 | 1 | 19 | 1 | 193 | 7333 |
| ISO PKI Two pass | 2 | 2 | 24 | 1 | 247 | 9800 |
| ISO PKI Two pass mutual | 2 | 2 | 32 | 1 | 298 | 12745 |
| ISO Three pass | 2 | 3 | 33 | 1 | 339 | 15382 |
| ISO Two pass Parallel | 2 | 4 | 40 | 1 | 400 | 17423 |
| Bilateral Key Exchange | 2 | 5 | 34 | 2 | 340 | 13440 |
| Needham Schroeder PKI | 3 | 7 | 43 | 1 | 527 | 18200 |
| SPLICE/AS | 3 | 6 | 50 | 2 | 539 | 22325 |
| Denning Sacco PKI | 3 | 4 | 44 | 2 | 420 | 19004 |
| CCITT X.509 | 2 | 3 | 40 | 2 | 375 | 16192 |
| Shamir Rivest Adelman | 2 | 3 | 18 | 2 | 259 | 8767 |
| Encrypted Key Exchange | 2 | 6 | 31 | 2 | 397 | 15527 |
| Davis Swick | 3 | 3 | 32 | 1 | 355 | 13852 |
| Davis Swick w/ Key Distr. | 3 | 5 | 51 | 2 | 538 | 25149 |
| *AVISPA specifications* | | | | | | |
| IKE Signatures | 2 | 4 | 44 | 2 | 446 | 30064 |
| IKE Signatures extension | 2 | 6 | 58 | 2 | 610 | 41685 |
| IKE Mac | 2 | 4 | 40 | 2 | 430 | 29847 |
| IKE Mac extension | 2 | 6 | 54 | 2 | 594 | 41602 |
| IKE Signatures ext Child | 2 | 8 | 68 | 2 | 760 | 55331 |
| IKE Mac ext Child | 2 | 8 | 64 | 2 | 744 | 55248 |
| IKE EAP Archie | 2 | 8 | 89 | 3 | 1017 | 83683 |
| TLS 2way w/Certificates | 3 | 12 | 88 | 4 | 1189 | 77136 |

Figure 6.9: Automated implementation of protocols from Clark/Jacob and AVISPA.

# Chapter 7

# Conclusion

The first part of this deliverable reports on methods to enforce information flow policies in ABS applications, both at the ABS level and at the bytecode level. Three complementary methods have been considered: type-based methods, logical methods, and dynamic methods, and evaluated on a common case study. The second part of this deliverable reports on the automatic generation of protocol implementations in ABS. The prototype has been successfully evaluated on a large set of protocols.

Our main results include:

- provably sound information flow type systems for core fragments of ABS and Java bytecode, and type-preserving compilation results (Chapter 3);
- logical methods to verify (classes of) hyperproperties, including a method based on epistemic logic to verify permissive information flow policies (Chapter 4);
- a source-to-source transformation that captures the effects of secure multi-execution, and a dynamic method to enforce data processing security policies (Chapter 5);
- a method to generate from high-level descriptions protocol implementations in ABS (Chapter 6).

There are several important directions for future work. Several of them will be/are being explored in related tasks, including Task 4.3 and Task 5.4.

# Bibliography


[1] Doug Aamoth. Taintdroid tattles on misbehaving android apps. TIME Techland website, September 30 2010. `http://techland.time.com/2010/09/30/taintdroid-tattles-on-misbehaving-android-apps/`.

[2] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. *SIGPLAN Not.*, 37:4–16, January 2002.

[3] G. R. Andrews and R. P. Reitman. An axiomatic approach to information flow in programs. *ACM Transactions on Programming Languages and Systems*, 2(1):56–76, 1980.

[4] A. Armando and L. Compagna. Automatic SAT-Compilation of Protocol Insecurity Problems via Reduction to Planning. In D.A. Peled and M.Y. Vardi, editors, *Proceedings of 22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*, LNCS 2529, pages 210–225. Springer-Verlag, Houston, Texas, November 2002. Available at `www.avispa-project.org`.

[5] Charu Arora and Mathieu Turuani. Validating Integrity for the Ephemerizer's Protocol with CL-Atse. In *Formal to Practical Security: Papers Issued from the 2005-2008 French-Japanese Collaboration*, volume 5458 of *Lecture Notes in Computer Science*, pages 21–32. Springer, 2009.

[6] William Ross Ashby. *An Introduction to Cybernetics*. Chapman & Hall, London, 1956.

[7] Aslan Askarov and Andrei Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In *ESORICS*, pages 197–221, 2005.

[8] Michael Backes, Alex Busenius, and Catalin Hritcu. On the Development and Formalization of an Extensible Code Generator for Real Life Security Protocols. In *4th NASA Formal Methods Symposium (NFM 2012)*, 2010.

[9] M. Balliu, M. Dam, and G. Le Guernic. Epistemic Temporal Logic for Information Flow Security. In *Proc. W. Programming Languages and Analysis for Security*, 2011.

[10] Musard Balliu, Mads Dam, and Gurvan Le Guernic. Encover: Concolic testing of java programs for information flow security. This is a draft of a future paper, January 2012.

[11] Anindya Banerjee and David A. Naumann. Stack-based access control and secure information flow. *J. Funct. Program.*, 15(2), 2005.

[12] G. Barthe, A. Basu, and T. Rezk. Security types preserving compilation. *Journal of Computer Languages, Systems and Structures*, 2005.

[13] G. Barthe, P. D'Argenio, and T. Rezk. Secure Information Flow by Self-Composition. In R. Foccardi, editor, *Computer Security Foundations Workshop*, pages 100–114. IEEE Press, 2004.

[14] G. Barthe, D. Naumann, and T. Rezk. Deriving an information flow checker and certifying compiler for Java. In *Symposium on Security and Privacy*. IEEE Press, 2006.

[15] Gilles Barthe, Juan Manuel Crespo, Dominique Devriese, Frank Piessens, and Exequiel Rivas. Secure multi-execution through static program transformation. Submitted.

[16] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In *FM*, pages 200–214, 2011.

[17] Gilles Barthe, Pedro D'Argenio, and Tamara Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 2011.

[18] Gilles Barthe, David Pichardie, and Tamara Rezk. A certified lightweight non-interference java bytecode verifier. In *ESOP*, pages 125–140, 2007.

[19] Gilles Barthe, David Pichardie, and Tamara Rezk. A certified lightweight non-interference java bytecode verifier. *Mathematical Structures in Computer Science*, 2012.

[20] Gilles Barthe, Tamara Rezk, Alejandro Russo, and Andrei Sabelfeld. Security of multithreaded programs by compilation. *ACM Trans. Inf. Syst. Secur.*, 13(3), 2010.

[21] Gilles Barthe and Exequiel Rivas. Static enforcement of information flow policies for a concurrent jvm-like language. In *TGC*, 2011.

[22] David A. Basin, Sebastian Mödersheim, and Luca Viganò. An on-the-fly model-checker for security protocol analysis. In Einar Snekkenes and Dieter Gollmann, editors, *ESORICS*, volume 2808 of *Lecture Notes in Computer Science*, pages 253–270. Springer, 2003.

[23] BBC. Google android apps found to be sharing data. BBC News website, September 30 2010. `http://www.bbc.co.uk/news/technology-11443111`.

[24] Bernhard Beckert, Reiner Hähnle, and Peter Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer-Verlag, 2007.

[25] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive non-interference for a browser model. In *NSS*, 2011.

[26] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

[27] Yohan Boichut. *Approximations pour la verification automatique de protocoles de securite*. PhD thesis, These de Doctorat d'Université, Universite de Franche-Comté, September 2006.

[28] Liana Bozga, Yassine Lakhnech, and Michaël Périn. Hermes: An automatic tool for verification of secrecy in security protocols. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 219–222. Springer, 2003.

[29] Philippa J. Broadfoot and A. W. Roscoe. Tutorial on fdr and its applications. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN*, volume 1885 of *Lecture Notes in Computer Science*, page 322. Springer, 2000.

[30] Alex Busenius. Expi2java Tutorial. Information Security and Cryptography Group,, 2011. Saarland University, Germany.

[31] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[32] R. Capizzi, A. Longo, V. N. Venkatakrishnan, and A. Prasad Sistla. Preventing information leaks through shadow executions. In *ACSAC*, 2008.

[33] Manuel Cheminod, Ivan Cibrario Bertolotti, Luca Durante, Riccardo Sisto, and Adriano Valenzano. Tools for cryptographic protocols analysis: A technical and experimental comparison. *Comput. Stand. Interfaces*, 31:954–961, September 2009.

[34] Jack Clark. Many android apps reveal user data. ZDNet website, September 30 2010. `http://www.zdnet.com/news/many-android-apps-reveal-user-data/470361`.

[35] John Clark and Jeremy Jacob. A Survey of Authentication Protocol Literature, 1997. Version 1.0, Unpublished Report, University of York, `http://cs.york.ac.uk/~jac/papers/drareview.ps.gz`.

[36] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *CSF*, pages 51–65, 2008.

[37] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[38] Ricardo Corin and Ro Etalle. An improved constraint-based system for the verification of security protocols. In *9th Int. Static Analysis Symp. (SAS), volume LNCS 2477*, pages 326–341. Springer-Verlag, 2002.

[39] C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Ph.D. dissertation, Eindhoven University of Technology, 2006.

[40] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a flexible information flow architecture for software security. In *International Symposium on Computer Architecture*, pages 482–493. ACM, 2007.

[41] Mads Dam, Bart Jacobs, Andreas Lundblad, and Frank Piessens. Security monitor inlining for multithreaded Java. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genova, Italy, July 6-10, 2009, Proceedings,*, pages 546–569. Springer-Verlag, July 2009.

[42] Mads Dam, Bart Jacobs, Andreas Lundblad, and Frank Piessens. Provably Correct Inline Monitoring for Multithreaded Java-like Programs. *Journal of Computer Security*, 18(1):37–59, 2010.

[43] Ádám Darvas, Reiner Hähnle, and David Sands. A theorem proving approach to analysis of secure information flow. In *SPC*, pages 193–209, 2005.

[44] Leonardo De Moura and Nikolaj BjÃ¸rner. Z3: An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems*, 4963/2008:337–340, 2008.

[45] Report on the Core ABS Language and Methodology: Part A, March 2010. Part of Deliverable 1.1 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[46] Verification of Behavioral Properties, March 2011. Deliverable 2.5 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[47] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *IEEE Symposium on Security and Privacy*, pages 109–124, 2010.

[48] David L. Dill. A retrospective on mur*hi*. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 77–88. Springer, 2008.

[49] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. Symp. Operating Systems Design and Implementation*, pages 1–6, October 2010.

[50] Laboratoire SpÃ©cification et VÃ©rification. SPORE Security Protocol Open REpository. `http://www.lsv.ens-cachan.fr/spore/`.

[51] Jeffrey Stewart Fenton. Memoryless subsystems. *The Computer Journal*, 17(2):143–147, 1974.

[52] J. S. Fritzinger and M. Mueller. Java security. Technical report, Sun Microsystems, Inc., 1996.

[53] Priya Ganapati. Study shows some android apps leak user data without clear notifications. WIRED website, September 30 2010. `http://www.wired.com/gadgetlab/2010/09/data-collection-android/`.

[54] Israel Gat and Harry J. Saal. Memoryless execution: A programmer's viewpoint. *Software: Practice and Experience*, 6(4):463–471, October 1976.

[55] Li Gong. Java security: present and near future. *IEEE Micro*, 17(3):14–19, May 1997.

[56] Claire Goues and Westley Weimer. Specification mining with few false positives. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, TACAS '09, pages 292–306, Berlin, Heidelberg, 2009. Springer-Verlag.

[57] Anders Moen Hagalisletto. Automated Refinement of Security Protocols. In *Workshop SSN in the Proceedings of 20th IEEE International Parallel and Distributed Processing Symposium*, 2006.

[58] Kevin W. Hamlen and Micah Jones. Aspect-oriented in-lined reference monitors. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 11–20, Tucson, Arizona, June 2008.

[59] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Certified in-lined reference monitoring on .NET. In *Proceedings of the 2006 workshop on Programming languages and analysis for security*, PLAS '06, pages 7–16, New York, NY, USA, 2006. ACM.

[60] Mauro Jaskelioff and Alejandro Russo. Secure multi-execution in haskell. In *PSI*, 2011.

[61] Rajeev Joshi and K. Rustan M. Leino. A semantic approach to secure information flow. *Sci. Comput. Program.*, 37(1-3):113–138, 2000.

[62] C. Kaufman. Internet Key Exchange (IKEv2) Protocol. RFC 4306 (Proposed Standard), December 2005. Obsoleted by RFC 5996, updated by RFC 5282.

[63] Sarfraz Khurshid, Corina Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer Berlin / Heidelberg, 2003.

[64] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19:385–394, July 1976.

[65] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, October 1973.

[66] Gurvan Le Guernic. Draft of an Introductory Survey to Dynamic Information Flow Security. In preparation.

[67] Christopher League, Zhong Shao, and Valery Trifonov. Precision in practice: A type-preserving java compiler. In Görel Hedin, editor, *CC*, volume 2622 of *Lecture Notes in Computer Science*, pages 106–120. Springer, 2003.

[68] Steven B. Lipner. A comment on the confinement problem. In *Proc. ACM Symp. on Operating System Principles*, pages 192–196. ACM, 1975.

[69] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. Mcmas: A model checker for the verification of multi-agent systems. In *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 682–688. Springer Berlin / Heidelberg, 2009.

[70] Gavin Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In *Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 147–166. Springer-Verlag, 1996.

[71] Gavin Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6:53–84, 1998.

[72] Catherine Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[73] Elinor Mills. What's that android app doing with my data? CNET news website, September 29 2010. `http://news.cnet.com/8301-27080_3-20018102-245.html`.

[74] Elinor Mills. Google sued over Android data location collection. CNET news website, April 28 2011. `http://news.cnet.com/8301-27080_3-20058493-245.html`.

[75] Dimiter Milushev, Wim Beck, and Dave Clarke. Noninterference via symbolic execution. Submitted to IFIP International Information Security and Privacy Conference, 2012.

[76] Dimiter Milushev and Dave Clarke. Towards incrementalization of holistic hyperproperties. In *Lecture Notes in Computer Science*. Springer, March 2012.

[77] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Principles of Programming Languages*, pages 228–2141. ACM 1Press, 1999. Ongoing development at `http://www.cs.cornell.edu/jif/`.

[78] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *POPL*, pages 228–241, 1999.

[79] Martin Pettai and Peeter Laud. Securing the future — an information flow analysis of a distributed oo language. In *SOFSEM*, 2012. To appear.

[80] François Pottier. A simple view of type-secure information flow in the p-calculus. In *CSFW*, pages 320–330, 2002.

[81] The AVISPA project. Automated validation of internet security protocols and applications. `http://avispa-project.org/library/sip.html`.

[82] Charles Reis, Adam Barth, and Carlos Pizano. Browser security: lessons from Google Chrome. *Commun. ACM*, 52:45–49, August 2009.

[83] Joel Rosenblatt. Apple Sued Over Applications Giving Information to Advertisers. Businessweek website, January 05 2011. `http://www.businessweek.com/news/2011-01-05/apple-sued-over-applications-giving-information-to-advertisers.html`.

[84] Alejandro Russo and Andrei Sabelfeld. Securing interaction between threads and the scheduler. In *CSFW*, pages 177–189, 2006.

[85] Alejandro Russo and Andrei Sabelfeld. Security for multithreaded programs under cooperative scheduling. In *Ershov Memorial Conference*, pages 474–480, 2006.

[86] Andrei Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Ershov Memorial Conference*, pages 225–239, 2001.

[87] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[88] Zhong Shao and Andrew W. Appel. A type-based compiler for standard ml. In *PLDI*, pages 116–129, 1995.

[89] G. Smith and D. Volpano. Secure Information Flow in a Multi-threaded Imperative Language. In *Principles of Programming Languages*, pages 355–364, 1998.

[90] Geoffrey Smith. A new type system for secure information flow. In *CSFW*, pages 115–125, 2001.

[91] Dawn Song, Sergey Berezin, and Adrian Perrig. Athena, a Novel Approach to Efficient Automatic Security Protocol Analysis. *Journal of Computer Security*, 9(1,2):47–74, 2001.

[92] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *12th International Static Analysis Symposium*, pages 352–367, September 2005.

[93] Tachio Terauchi and Alexander Aiken. Secure information flow as a safety problem. In *SAS*, pages 352–367, 2005.

[94] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. RIFLE: An architectural framework for user-centric information-flow security. In *Proc. Symp. Microarchitecture*, December 2004.

[95] Dries Vanoverberghe and Frank Piessens. A caller-side inline reference monitor for object-oriented intermediate language: extended version. CW Reports CW512, Department of Computer Science, K.U.Leuven, March 2008.

[96] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10:203–232, 2003.

[97] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[98] Dennis M. Volpano and Geoffrey Smith. A type-based approach to program security. In *TAPSOFT*, pages 607–621, 1997.

# Glossary

**ABS** Abstract Behavioral Specification language. An executable class-based, concurrent, object-oriented modeling language based on Creol, created for the HATS project.

**COG** Concurrent Object Group, the unit of parallelism in ABS.

**Core ABS** The behavioral functional and object-oriented core of the ABS modeling language.

**Creol** A precursor language to ABS, where the concurrency model of ABS with concurrent objects, asynchronous method calls, and cooperative scheduling, was developed.

**Epistemic logic** A modal logic that is concerned with reasoning about knowledge.

**KeY** The KeY system is a software development tool that aims at integrating design, implementation, formal specification and formal verification of object-oriented software as seamlessly as possible.

**Non-Interference** Security property ensuring confidentiality with respect to a given security policy.

**Secure multi-execution** Dynamic enforcement technique of security policies based on modified semantics for input/output events and concurrent execution of copies of the same program.

**Self-composition** Technique used to establish non-interference properties through the use of standard program logics.

**Symbolic execution** Program analysis technique used to investigate the possible execution traces of a program.

**Runtime monitoring** Technique that aims at detecting errors in software during execution.