Secure Program Partitioning in Dynamic Networks

Dan Søndergaard

Kongens Lyngby 2006
IMM-M.Sc-2006-92

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Abstract

A shortcoming of many systems dealing with sensitive data is that they do not control the propagation of information in an appropriate way. This includes both the denial of access for unauthorized principals, and the control of the data's integrity. Previous work has shown that security-typed programs can successfully address this shortcoming.

Security-typed programs can safely be distributed and executed in a network, as shown by Zdanewic et al. The distributed programs obey, by construction, all annotations with respect to access rights. The approach does not, however, support dynamic changes to the network or the trust model.

In this thesis, the original framework for distribution of security-typed programs has been extended to also consider dynamic systems. The main contribution is the development of a trust model with support for dynamic systems. Moving to a dynamic setting introduces new problems, e.g., the choice between several feasible distributions of a program. To address this, a metric is developed which can be used to find the most trustworthy distribution based on a user's preferences.

The proposed concepts have been proven to work through the implementation of a prototype.

Keywords: Information Flow, Distributed Systems, Trust, Program Partitioning

Resumé

En svaghed i mange IT systemer er, at man ikke kontrollerer, hvordan informationer bliver videredistribueret. Dette inkluderer både manglende kontrol af hvem der læser informationerne, såvel som at opretholde informationernes integritet. Det er blevet påvist, at programmer med sikkerhedstyper indbygget i sproget kan adressere denne svaghed.

Zdanewic et al. har vist hvordan programmer med indbygget sikkerhed kan distribueres og eksekveres i et netværk. De distribuerede programmer vil overholde programmets sikkerhedskrav. Denne metode understøtter dog ikke ændringer i netværket.

I dette projekt er det originale framework, til at distribuere programmer med, blevet udvidet til også at kunne understøtte dynamiske netværk. Hovedformålet har været at udvikle en trust model, der understøtter dynamiske netværk. I et dynamisk netværk optræder der nye udfordringer, såsom at vælge mellem flere gyldige distributioner af et program. Denne udfordring er blevet håndteret ved at udvikle en *metric*, der kan bruges til at finde den mest troværdige distribution fra en brugers synspunkt.

Der er blevet udviklet en prototype, der demonstrerer, at det udviklede koncept virker.

Nøgleord: Information Flow, Distribuerede Systemer, Trust, Program Partitioning

Preface

This Master's thesis was carried out at the Department of Informatics and Mathematical Modelling at the Technical University of Denmark. The work was carried under supervision of Assistant Professor Christian W. Probst. The project ran from March to October, 2006.

Part of the work presented in this thesis has also been published in the article *Program partitioning using dynamic trust* [SPJH06]. The article was accepted at the *Formal Aspects of Secuirty and Trust 2006* workshop (FAST2006) held in Hamilton, Canada.

I would like to thank Christian W. Probst for his support and guidance throughout the project. I would also like thank the co-authors of the FAST2006 article, Christian D. Jensen, René R. Hansen, and again Christian W. Probst. Finally, My thanks also go to friends and family for their support. Special thanks to Maja L. Strang and Kristian S. Knudsen for reading and commenting on the thesis.

Dan Søndergaard
Kongens Lyngby, October 2006

Contents

Chapter 1

Introduction

Security is always excessive until it's not enough.
– Robbie Sinclair, Head of Security, Country Energy

The increased use of *distributed systems* together with added complexity, leaves many modern computing systems vulnerable to a wide range of attacks. The importance of these systems is hard to over-estimate, so the consequences of these systems being compromised are wide-reaching. The aim of this thesis is to contribute to the ongoing research of making modern distributed systems safer.

Information flow policies have been proposed to increase the security of computing systems. Information flow policies are a low-level approach to security, where all information is annotated with a security policy stating how the information must be used. The advantage of this approach is that it not only restricts data-access, but also *propagation*. This means that even when the information is released to another person, process or server, it can only be used in the way the policy states. This is a lot more secure than current approaches, where all control is lost when the information once released.

Denning and Denning [Den76] introduced static validation of information flow, called *Secure Information Flow.* This is a language-based technique which can check the security of programs at compile time. Zdanewic et al. [ZZNM01] proposed a framework, *Secure Program Partitioning* which apply Secure Information Flow to distributed systems. Throughout this thesis, the term *distributed system* will be used as in ([ZZNM01]) to refer to a system, which uses multiple hosts divided by a network during execution. In this definition distributed systems are not necessarily executed in parallel. The Secure Program Partitioning framework takes security-typed programs and distributes the execution onto multiple hosts. The framework, however, has several shortcomings:

- The Secure Program Partitioning framework only considers static networks. This is a very limiting short-coming, as most distributed systems have a dynamic user base.
- The trust model used in the framework is too simple and unrealistic for most applications. Additionally, due to it only considers static networks, at least it has to be replaced with a dynamic model, when moving to a dynamic setting.

The first short-coming was addressed by Hansen and Probst in [HP05]. They proposed an extension of the framework, which considers dynamic networks. This thesis aims to develop further on this framework.

The second shortcoming is the main focus of this thesis. A more realistic trust model will be developed, which is better able to respond to the challenges of dynamic, distributed systems. Introducing partial and recommended trust will allow users to express trust more accurately, as well as allow trust to be propagated in a sound manner. These improvements will increase the usability and security of Secure Program Partitioning. Additionally, extending Secure Program Partitioning to dynamic networks allows for new interesting applications, *grid systems*, *online transactions*, etc. The work on trust models, has also been published as an article [SPJH06] at the *Formal Aspects of Security and Trust 2006* workshop in Hamilton, Canada.

The developed framework will be customizable and extendable. This flexibility makes the framework better suited for practical use, as the individual applications can customize the framework for their individual use. The flexibility will be achieved by supplying the trust model and optimization criteria as pa-

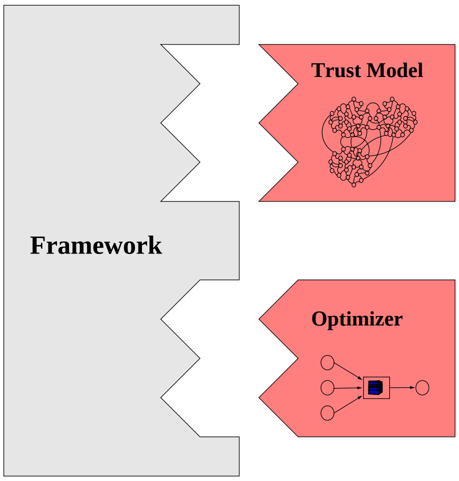


Figure 1.1: The proposed framework allows new trust models and optimizations to be plugged in.

rameters. Allowing applications to use their own trust model means that the framework can be incorporated into an already existing trust infrastructure, or a new custom trust model, which suit the application's specific purpose can be made.

Secure Program Partitioning, as mentioned, distributes program execution onto multiple hosts. Often, several valid splits exist. This allows for optimizations. By introducing the optimization criteria as a parameter, the individual application can introduce its own optimizations.

- **Security** – All the possible splits are guaranteed to be valid under the requirements of Secure Program Partitioning. However, an application might have its individual security optimizations.

- **Performance** – Applications might want to optimize performance, for instance by minimizing network traffic or schedule hard computations to strong hosts.

Figure 1.1 illustrates how new trust models and optimization criteria can be plugged in to the framework.

The improvements mentioned above will make Secure Program Partitioning more secure and better suited for realistic applications. To demonstrate the capabilities and soundness of the proposed improvements a prototype will be developed as part of this thesis.

1.1 Structure of this Thesis

This thesis is structured as follows. The next chapters are a thorough presentation of Secure Information Flow (Chapter 2) and Secure Program Partitioning (Chapter 3).

In Chapter 4 the *sflow* language is introduced. Hereafter, Secure Program Partitioning is extended to dynamic networks in Chapter 5. The design of the prototype system is presented in Chapter 6, followed by a chapter on implementation of this design (Chapter 7).

The framework and implementation is evaluated in Chapter 8. A discussion on future work can be found in Chapter 9. Finally, a general conclusion is presented in Chapter 10.

The thesis also contains a few appendices that can be consulted, if additional details are needed on the testing and implementation of the prototype.

In addition to this thesis a CD-ROM is enclosed. It contains the developed program, and the source code. The examples used in this thesis can also be found on the CD-ROM.

Chapter 2

Information Flow

Information wants to be free... Or does it?
– Anonymous Coward, Slashdot.org,
October 8, 2005

The traditional, and still predominant approach to information security is to (only) restrict access to data. Access is restricted using methods like access control and encryption. Even though it is the standard in modern information systems, it has some critical shortcomings. One of these is the lack of control of data after it has been released. Once released to another principal data might be leaked to a third party, intentionally or by mistake.

Secure Information Flow is an end-to-end security policy, where not only data access is restricted but also the use of data. This allows for specifying much stronger security policies than the traditional approach.

This chapter introduces Secure Information Flow by looking at several different approaches. First, Denning's approach [Den76] is considered (Section 2.2). Secondly, type systems are examined by looking at Volpano's approach [VSI96] (Section 2.3). Finally, the Decentralized Label Model will be investigated [ML97] (Section 2.4).

The motivation for these approaches is to maintain *non-interference*. Before Denning's approach is dealt with, non-interference is introduced.

2.1 Non-interference

Non-interference is a security property, which guarantees that no data is ever leaked to a lower security class. By upholding this the propagation of data can be controlled at all times.

[GM82] defines non-interference as:

Definition 2.1 (Non-interference) A program preserve *non-interference*, iff all low level output is independent of all high level input.

The property of non-interference is a very strong security guarantee, but in many cases it is also too restrictive. In later sections, a sensible method for circumventing the non-interference requirement will be introduced.

2.2 Secure Information Flow

Secure Information Flow was introduced by Denning in 1976 [Den76]. It introduces a security model consisting of *information receptacles* (N) (for instance variables in a program). A *security class* is assigned to each receptacle (SC). Additionally, operators exist for combining security classes ($\sqcup$) and checking whether information flow between two classes is allowed ($\sqsubseteq$):

$$FM = \langle N, SC, \sqcup, \sqsubseteq \rangle$$

Denning arranges the security classes in a *lattice model*, partially ordered by the $\sqsubseteq$-operator.

- Flow of information into a target object is only allowed if the data assigned has a *lower or equal* security level:

$$SC_1 \sqsubseteq SC_2$$

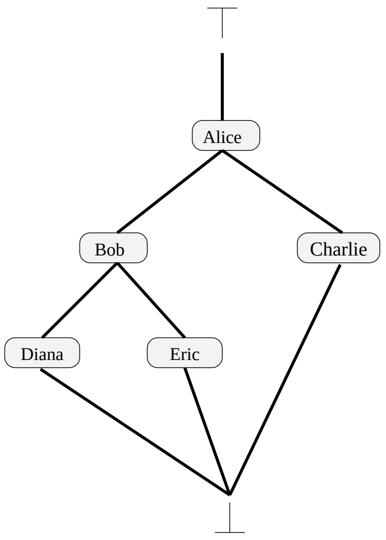


Figure 2.1: Simple lattice model

I.e. data with security class SC_1 can flow to SC_2.

- When objects are combined (e.g. arithmetic operation on two variables), the resulting security class is the *Least Upper Bound* (or LUB) of the two objects' security classes:

  $$SC_1 \sqcup SC_2$$

  The least upper bound is the lowest security class, where

  $$SC_1 \sqsubseteq SC_1 \sqcup SC_2 \quad \wedge \quad SC_2 \sqsubseteq SC_1 \sqcup SC_2$$

- The lattice also has an operator for *Greatest Lower Bound* (or GLB).

  $$SC_1 \sqcap SC_2$$

  The greatest lower bound is the highest security class, where

  $$SC_1 \sqcap SC_2 \sqsubseteq SC_1 \quad \wedge \quad SC_1 \sqcap SC_2 \sqsubseteq SC_2$$

```
var H : bool
var L : bool

if H then
    L := true
else
    L := false
```

Listing 2.1: Implicit flow example

Example 2.1 (A simple lattice model) In Figure 2.1 a simple lattice model is depicted. The persons in the graph could for instance be users on a computer (administrators, super-users, users, etc.) or security clearances in a company (managers, accountants, secretaries etc.). These cases have a hierarchical structure which can be reflected using a lattice model.

If data owned by Bob is combined with data owned by Charlie, only Alice is able to read the resulting data:

$$\begin{aligned} Bob \sqcup Charlie &\not\sqsubseteq Bob \\ Bob \sqcup Charlie &\sqsubseteq Alice \end{aligned}$$

Data owned by Diana or Eric is not accessible to Charlie, as they only trust Bob (and consequently Alice).

$$Diana \not\sqsubseteq Charlie$$

The greatest lower bound are derived in a similar matter. For instance:

$$Bob \sqcap Charlie = \bot$$

□

2.2.1 Implicit flow

In languages containing conditions (e.g. if and while) the problem of *Implicit Flow* arises. Implicit flow is best illustrated with an example. In Listing 2.1 the program has two boolean variables L and H (Low and High). H belongs to a higher security class, $L \sqsubseteq H$ and $L \neq H$, so the assignment $L := H$ is not allowed. Nevertheless, using an if-condition the equivalent of an assignment can be achieved without using a direct assignment, as the program shows.

This small example illustrates how an implicit flow occurs when conditional statements are used. To prevent this kind of information leak, the security class of the condition statement must be considered as well. Secure Information Flow can be achieved by assigning a block label to each *basic block*. This label is then combined with the label in any assignment using the $\sqcup$-operator.

If this approach is applied to the program in Listing 2.1, the program will be considered insecure as the block label in the if-then-else statement will be H. The assignment will fail because $H \not\sqsubseteq L$.

2.2.2 Enforcing security using static analysis

Secure information flow can be ensured using *static analysis*. Every flow of information must be checked in the program. If no explicit or implicit leaks occur, the program is considered secure, and is allowed to be run. The check of an assignment is shown in Figure 2.2.

The fact that secure information flow can be checked statically, allow programs to be verified at *compile time*. Compared to *run-time* security checks this offers:

- Increase of performance. The security check is only done when the program is compiled. Afterward it can run indefinitely without any further security verification needed.
- The program is validated before being run. i.e. the compiler assists the programmer in writing secure programs. In fact it will not compile if it is not secure.
- Implicit flow is hard to monitor at run-time. Branching in the program is easier to analyze statically [Pob04].

While Denning used *abstract syntax trees* to verify Secure Information Flow,

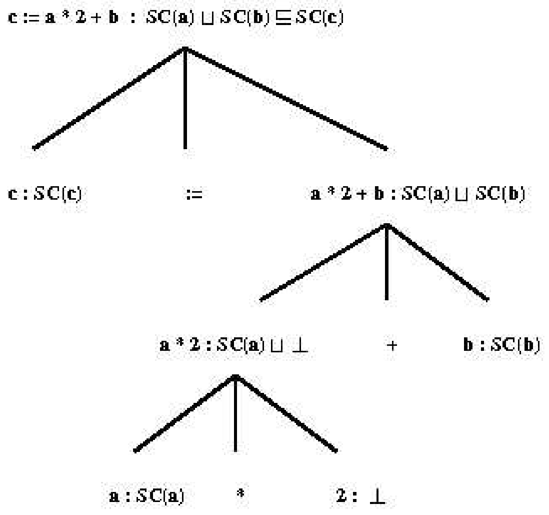

c := a * 2 + b : SC(a) ⊔ SC(b) ⊑ SC(c)
c : SC(c)
:=
a * 2 + b : SC(a) ⊔ SC(b)
a * 2 : SC(a) ⊔ ⊥
+
b : SC(b)
a : SC(a)
*
2 : ⊥

Figure 2.2: Example of how an abstract syntax tree can be used to check information flow. SC is used to find the security class of a variable. The assignment is allowed if $SC(a) \sqcup SC(b) \sqsubseteq SC(c)$.

$$\frac{}{\Gamma, bl \vdash val : \bot} \qquad \frac{\Gamma(x) = l}{\Gamma, bl \vdash x : l}$$

$$\frac{\Gamma, bl \vdash e_1 : l_1 \qquad \Gamma, bl \vdash e_2 : l_2}{\Gamma, bl \vdash e_1 \ \texttt{op}\ e_2 : l_1 \sqcup l_2} \quad \frac{\Gamma, bl \vdash e : l \qquad l \sqcup bl \sqsubseteq \Gamma(x)}{\Gamma, bl \vdash x := e}$$

$$\frac{\Gamma, bl \vdash e : l \qquad \Gamma, l \sqcup bl \vdash c_1 \qquad \Gamma, l \sqcup bl \vdash c_2}{\Gamma, bl \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2}$$

Figure 2.3: Typing rules for secure information flow

Volpano used type systems instead. Volpano's approach is introduced in the next section.

2.3 Type systems

Volpano et al. took a different approach to secure information flow [VSI96]. Instead of using static analysis like Denning, they use a type system which ensures Secure Information Flow. While still being static it incorporates the secure information flow model into a type system. The type system of Volpano will only be described briefly here. Type systems will be dealt with in more detail, when the sflow-language is introduced in Chapter 4.

In Figure 2.3 an example of typing rules for secure information flow is shown. The environment consists of a security class store:

$$\Gamma : \text{VAR} \rightharpoonup \text{SC}$$

and the block label *bl*.

Incorporating Secure Information Flow into the type system has clear advantages from a compilation perspective. Also from a theoretical view, using a type system is a more elegant solution. In fact one of the main contributions of Volpano's article is the *soundness* argument, and the proof of non-interference.

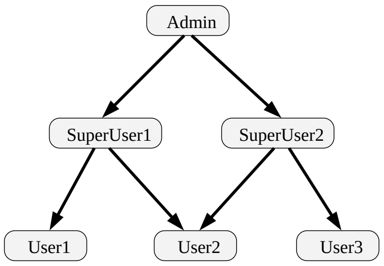


Figure 2.4: Example of principal hierarchy

2.4 Decentralized Label Model

The information flow model used in this thesis is the *Decentralized Label Model* (DLM) introduced by Myers and Liskov [ML97, ML00]. In this model, information is owned or read by *principals*. A principal can be:

- A user
- A group of users
- A process

Basically anyone or anything that has a relation to the information.

Principals can be ordered into hierarchies. The term *act for* is used when a principal is allowed to act for another principal. If principal p is allowed to act for q, we write $q \preceq p$. In Figure 2.4 an example of a principal hierarchy is depicted. The hierarchy reflects the security clearances on a computer system. For instance *SuperUser1* can act for *User1* and *User2*, but not for *User3*. *Admin* can act for all principals in the hierarchy.

A label consists of owners and readers. Each owner specifies the principals who are allowed to read the data:

$$\{o : r_1, ..., r_n\}$$

A label can have multiple owners, which each have a set of readers:

$$\left\{o_1 : \underline{R_1}; \ldots ; o_n : \underline{R_n}\right\} \tag{2.1}$$

where $\underline{R_i}$ denotes the set of readers allowed by owner o_i.

When information has multiple owners, it contains multiple policies, one for each owner. When accessing the data, all the policies must be obeyed. This means that a principal must be present as a reader in all policies, if he is to be allowed to read the data. The function *readers* is used to find the allowed readers:

$$readers(\left\{o_1 : \underline{R_1}; \ldots ; o_n : \underline{R_n}\right\}) = \bigcap_{i=1..n} \left(\{o_i\} \cup \underline{R_i}\right) \tag{2.2}$$

As this function shows, owner are implicit readers of their own policies.

Readers for a specific owner can also be extracted:

$$readers(\left\{o_1 : \underline{R_1}; \ldots ; o_n : \underline{R_n}\right\}, o_k) = \underline{R_k}\ ,\ 1 \leq k \leq n \tag{2.3}$$

Finally, the function *owners* returns the owners of a label:

$$owners(\left\{o_1 : \underline{R_1}; \ldots ; o_n : \underline{R_n}\right\}) = \{o_1, \ldots, o_n\} \tag{2.4}$$

Example 2.2 (Owners and Readers) The three functions applied to the label $\{A : C\ ;\ B : A, C\}$ results in the following labels:

$$\begin{array}{rcl} readers(\{A : C\ ;\ B : A, C\}) & = & \{A, C\} \\ readers(\{A : C\ ;\ B : A, C\}, A) & = & \{C\} \\ owners(\{A : C\ ;\ B : A, C\}) & = & \{A, B\} \end{array}$$

2.4.1 Lattice Model

Labels in the Decentralized Label Model can be ordered in a lattice, ordering according to restrictiveness of the labels.

- Each added owner makes the label *more restrictive.*

- Each added reader makes the label *less restrictive*.

This means the ordering of two labels can be expressed as:

Definition 2.2 (Less restrictive) Label L_1 is *less restrictive* than L_2, $L_1 \sqsubseteq L_2$, iff

$$\begin{aligned} &(owners(L_1) \subseteq owners(L_2)) \land \\ &(\forall p \in owners(L_1) : readers(L_2, p) \subseteq readers(L_1, p)) \end{aligned} \tag{2.5}$$

This states that all owners in L_2 must be present in L_1. And for all owners in the less restrictive L_1, all readers of L_2 must be included in L_1.

When two labels are combined the *least upper bound* (LUB) operator is used.

Definition 2.3 (Least upper bound) The *least upper bound* of two labels L_1 and L_2 is

$$\begin{aligned} L_1 \sqcup L_2 = \{o_k : \underline{R_k} \quad | \quad & o_k \in owners(L_1) \cup owners(L_2) \land \\ & \underline{R_k} = \varphi_{cr}(L_1, L_2, o_k)\} \end{aligned} \tag{2.6}$$

where,

$$\varphi_{cr}(L_1, L_2, o) = \begin{cases} readers(L_1, o) & \text{if } o \notin owners(L_2) \\ readers(L_2, o) & \text{if } o \notin owners(L_1) \\ readers(L_1, o) \cap readers(L_2, o) & \text{if } o \in owners(L_1) \\ & \land\ o \in owners(L_2) \end{cases}$$

i.e. in the least upper bound all policies must be included, and if two labels share a policy, then the intersection of the two reader-sets is the resulting reader-set.

2.4.2 Relabeling Rules

In the Decentralized Label Model a relabeling (replacing one label with another) is considered legal when:

$$L_1 \rightarrow L_2, \text{ if } L_1 \sqsubseteq L_2 \tag{2.7}$$

Intuitively this means you can always add owners and remove readers.

The *non-interference*-requirement has proved to be too restrictive for most practical applications. The Decentralized Label Model introduces the notion of *non-interference until declassification.* A principal is allowed to weaken its own policies, if this is done using the *declassify* expression. Declassify takes an expression e and changes its label to L.

To relax a policy of an owner o, the authority of o is of course needed. In the following the predicate *authority* will be used to check if the the operation has the authority of the principal. E.g.:

$$\{o_1 :;\ o_2 : r_1\} \overset{declassify}{\longrightarrow} \{o_1 : r_1; o_2 : r_1\},\ \text{if}\, authority(o_1)$$

o_1 can also remove his policy all together:

$$\{o_1 :;\ o_2 : r_1\} \overset{declassify}{\longrightarrow} \{o_2 : r_1\},\ \text{if}\, authority(o_1)$$

Example 2.3 (A Bank) In this example some of the processes in a bank will be modeled using the Decentralized Label Model. A bank has sensitive information, to which several principals need access: customers, bank employees, tax officials, etc.

For instance the label of an account balance can be modelled as:

$$\text{Balance} \quad : \quad \{Bank : Cust;\ Cust : Bank\}$$

The bank and customer co-own the account balance data, which means that it cannot be released to a third party, without both the bank and customer's permission.

We now introduce a function ATM, which is used when a customer withdraw money from an ATM.

```
function ATM(AmountWithdrawn : {Cust:Bank})
     Balance := Balance − AmountWithdrawn
```

The validity of the assignment can be resolved using the following type inference rule:

```
InsuranceQuote : {Ins:Cust}

function InsuranceQuote()
    if Balance > 10000 then
        InsuranceQuote := 1000
    else
        InsuranceQuote := 2000
```

Listing 2.2: Illegal Insurance quote program

$$\frac{\begin{array}{l}\text{Balance} : \{\mathit{Bank} : \mathit{Cust};\ \mathit{Cust} : \mathit{Bank}\}\\ \text{AmountWithdrawn} : \{\mathit{Cust} : \mathit{Bank}\}\\ \{\mathit{Bank} : \mathit{Cust};\ \mathit{Cust} : \mathit{Bank}\} \sqcup \{\mathit{Cust} : \mathit{Bank}\} \sqsubseteq \{\mathit{Bank} : \mathit{Cust};\ \mathit{Cust} : \mathit{Bank}\}\end{array}}{\text{Balance} := \text{Balance} - \text{AmountWithdrawn}}$$

The assignment is valid as the type-check is succesful.

Assume the bank has a strategic alliance with an insurance company *Ins*. If the customer gives his or her permission, the account balance will be shared with the insurance company. Based on the balance, the insurance company will give the customer a quote.

In Listing 2.2 a program which performs this task is listed. However, this program contains an illegal *implicit flow*. The block label for the if-then-else statement is $bl = \{Bank : Cust;\ Cust : Bank\}$. The check done by the type system (see Figure 2.3) fails:

$$\frac{\begin{array}{l}1000 : \bot\\ \bot \sqcup \{\mathit{Bank} : \mathit{Cust};\ \mathit{Cust} : \mathit{Bank}\} \sqsubseteq \{\mathit{Ins} : \mathit{Cust}\}\end{array}}{\text{InsuranceQuote} := 1000}$$

Because the label check $\{Bank : Cust;\ Cust : Bank\} \not\sqsubseteq \{Ins : Cust\}$ fails.

The program, however, can be corrected using a *declassify* statement. The program in Listing 2.3 does exactly this. Notice that the method needs the authority of both *Cust* and *Bank*, as their policies are made less restrictive (in fact they are removed all together).

```
InsuranceQuote : {Ins:Cust}
QuoteTemp : {Bank:Cust; Cust:Bank; Ins:Cust}

function InsuranceQuote() authority({Bank,Cust})
    if Balance > 10000 then
        QuoteTemp := 1000
    else
        QuoteTemp := 2000

    InsuranceQuote := declassify(QuoteTemp,{Ins:Cust})
```

Listing 2.3: Correct Insurance quote program

The new program will reveal information about the account balance to the insurance company, but only whether the balance is over or under 10000. From the customer's perspective this is a better solution, than having to disclose his account balance to the insurance company. □

2.4.3 Integrity

The information flow model used in this thesis also supports integrity. Integrity policies are the dual of privacy/confidentiality policies. Where privacy policies ensure that data is *read* properly, integrity policies make sure data is *written* properly. Integrity labels keep track of who has influenced the data. Using this information a principal can specify a policy, where he only allows principals he trusts to affect the data.

Unlike confidentiality labels, the integrity label does not have an owner (the ?-character is used to indicate an integrity label), it simply specifies who trusts the integrity of the data. Whenever two integrity labels are combined, the resulting integrity is the intersection of the two.

Definition 2.4 (Integrity – Least upper bound) The least upper bound of two integrity labels is defined by:

$$\{? : \underline{P_1}\} \sqcup \{? : \underline{P_2}\} = \{? : \underline{P_1} \cap \underline{P_2}\} \tag{2.8}$$

Intuitively, a principal needs to have trust in the integrity of the originating data in order to have trust in the resulting data.

Similarly, the lattice ordering operator, $\sqsubseteq$, is also the dual of its confidentiality counterpart.

Definition 2.5 (Integrity – Less restrictive) The less restrictive predicate of two integrity labels is:

$$\{? : \underline{P_1}\} \sqsubseteq \{? : \underline{P_2}\} \equiv \underline{P_2} \subseteq \underline{P_1} \tag{2.9}$$

The labels used in this thesis will support both confidentiality and integrity. The following notation is used:

$$\{o_1 : \underline{R_1}; \ldots; o_n : \underline{R_n};\ ? : \underline{P}\}$$

The ordering ($\sqsubseteq$) and meet ($\sqcup$) can be extended to the combined label in a straightforward manner:

$$L_1 \sqsubseteq L_2 \quad \equiv \quad C(L_1) \sqsubseteq C(L_2) \wedge I(L_1) \sqsubseteq I(L_2) \tag{2.10}$$
$$L_1 \sqcup L_2 \quad = \quad (C(L_1) \sqcup C(L_2)) \cup (I(L_1) \sqcup I(L_2)) \tag{2.11}$$

Where C and I extract the confidentiality label and integrity label, respectively.

Confidentiality policies can be made less restrictive using the declassify function. Similarly, integrity policies can be loosened using the endorse function. Endorse has to be used when principals are added to the integrity policy.

Example 2.4 (A Bank with Integriy) We now return to the bank example. Now the bank and customer wish to ensure the integrity of the balance. So an integrity label is added.

$$\text{Balance} \quad : \quad \{\mathit{Bank} : \mathit{Cust};\ \mathit{Cust} : \mathit{Bank};\ ? : \mathit{Bank}, \mathit{Cust}\}$$

This, however, will result in the ATM-function, from the previous example, becoming invalid.

$$\frac{\begin{array}{l}\text{Balance} : \{\mathit{Bank} : \mathit{Cust};\ \mathit{Cust} : \mathit{Bank};\ ? : \mathit{Bank}, \mathit{Cust}\} \\ \text{AmountWithdrawn} : \{\mathit{Cust} : \mathit{Bank}\} \\ \{\mathit{Bank} : \mathit{Cust};\ \mathit{Cust} : \mathit{Bank};\ ? : \mathit{Bank}, \mathit{Cust}\} \sqcup \{\mathit{Cust} : \mathit{Bank}\} \\ \sqsubseteq \{\mathit{Bank} : \mathit{Cust};\ \mathit{Cust} : \mathit{Bank};\ ? : \mathit{Bank}, \mathit{Cust}\}\end{array}}{\text{Balance} := \text{Balance} - \text{AmountWithdrawn}}$$

The label check will fail, because

$$\{Bank : Cust;\ Cust : Bank;\ ? : Bank, Cust\} \sqcup \{Cust : Bank\} = \{Bank : Cust;\ Cust : Bank\}$$

and

$$\{Bank : Cust;\ Cust : Bank\} \not\sqsubseteq \{Bank : Cust;\ Cust : Bank;\ ? : Bank, Cust\}$$

Nevertheless, this can be taken care of by both the bank and customer *endorsing* the amount to be withdrawn. The authority of the two principals is needed to do this. The corrected function becomes:

```
function ATM(AmountWithdrawn : {Cust:Bank})
        authority ({Bank, Cust})
    Balance := Balance -
               endorse(AmountWithdrawn, {?:Bank, Cust}\})
```

This resembles how an ATM works in practice. The bank through its trust in the ATM, the credit card, and the integrity of the PIN, endorse the withdrawal. The customer endorses the withdrawal by using his credit card and PIN code. As they both have trust in the integrity of the process, they will have trust in the integrity of the resulting balance.

Confidentiality ensured that the account balance would never be leaked to any third party. The introduction of integrity will give the bank and principal assurance that the balance is always correct. □

2.5 Covert Channels

Covert channels is a term for all indirect data flow. If any information is leaked in anyway, this is considered a covert channel. We have already seen the example of implicit flow. In this section other types of covert channels will be discussed briefly.

Timing channel The execution time is monitored. An example of this is the older implementation of OpenSSL, where the time of response changed, if a correct user name was typed. Additionally, for the password, each correct character would result in the response time changing. This is of course a serious flaw, which allow people to break the OpenSSL without any prior knowledge [Ope].

Network traffic The communication over the network is monitored, and based on the communications, branching in the program can be monitored.

Power monitoring Power consumption goes up when hard calculations are being performed, e.g. while-loops. It can also be monitored if a principal is active, by looking at its power consumption.

Storage monitoring The storage used can change during the execution. Thus the used storage, reveals information about the program.

Etc. Only imagination sets the limit for clever attacks using covert channels, therefore this list is not in any way complete.

Covert channels, besides implicit flow, will not be dealt with in our design. We assume perfect communication channels, where no eavesdropping is possible. It is also impossible to retrieve any information during execution on the individual hosts.

Now that Secure Information Flow and the Decentralized Label Model have been introduced, we look at how this can be used in distributed systems.

Chapter 3

Secure Program Partitioning

A leak? You call what's going on here a leak? Last time we had leak like like this, Noah built himself a boat!
– James A. Wells, from the movie 'Absence of Malice'

Secure Program Partitioning is a language based approach for protecting confidential data in distributed systems. The approach is based on Secure Information Flow and the Decentralized Label Model [ZZNM01].

In this approach a program, annotated with constraints for information flow, is distributed over the network. A unique feature of this approach is that it allows programs to be executed even in a scenario with mutual distrust.

When we move into a distributed setting, several new issues have to be handled:

- Trust between principals and hosts becomes very important, as the principals will only allow their data to be stored and manipulated by hosts they trust.
- Communication between hosts. How to make sure that the data transferred on the network is not intercepted by a third party.

- Synchronization is important. The program needs to be executed in the right order, and the synchronization mechanisms must be sufficiently robust not to allow any interference with the execution.

Secure Program Partitioning, like most current information flow systems, does not support multiple threads. This is still an open research area [SM03, SV98, RS06]. Hence, this approach does not support parallel execution on multiple hosts. Hopefully, future research will solve this issue, and thereby be better at utilizing the capabilities of distributed systems.

The original approach [ZZNM01] only deal with static networks. This thesis extends Secure Program Partitioning to dynamic networks, where hosts are allowed to leave and enter the network. This raises some important questions:

- Handling changes in the network. When principals leave the network, the program might need to be re-split. When principals join the network, a more optimal split might be possible.
- A trust model with support for dynamic behavior must be implemented.
- Handling data stored on a host who is leaving the network.

But before these questions are addressed, Secure Program Partitioning will be described.

3.1 Partitioning programs

Before any partitioning of a program takes place, the program needs to be verified. i.e. the program maintain *non-interference until declassification* (cf. section 2.4). This task is performed by a compiler with Secure Information Flow support, e.g. the JIF compiler [Mye99].

Once the program has been verified, the partitioning can take place. The splitter takes a program and a trust relation, and produces a split (if possible).

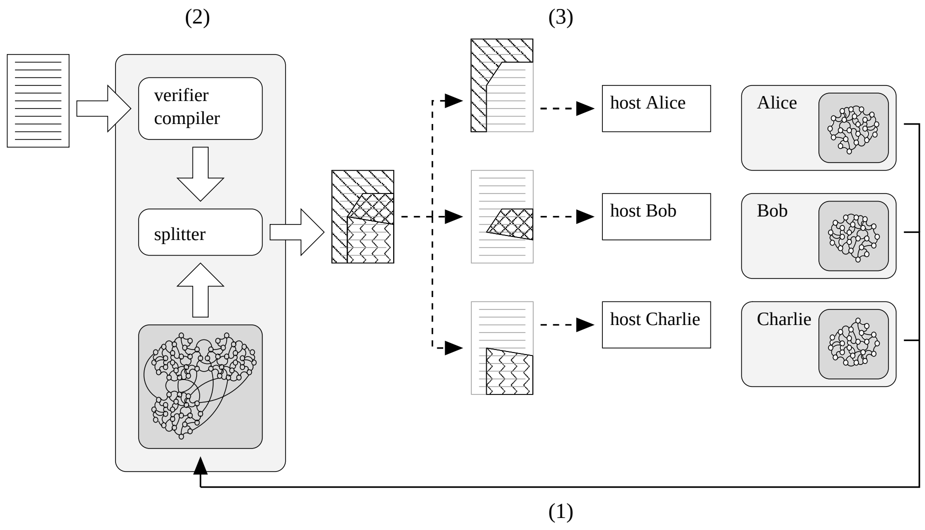


Figure 3.1: The splitting framework. Each principal has a local set of trust declarations. The splitter combines these into a global trust graph (1), and uses this to generate a split (2). Finally, the program is distributed across the network (3).

In the split program, each field and statement has been assigned to a host. In Figure 3.1 a general overview of the splitting process is depicted.

3.1.1 Assigning fields

The following security constraints must be satisfied for a field f to be assigned to a host h:

$$C(L_f) \sqsubseteq C_h \quad \text{and} \quad I_h \sqsubseteq I(L_f) \tag{3.1}$$

It states that the host h has at least as much confidentiality as the field f. Additionally, the host must have at least as much integrity as the field.

Example 3.1 (Assigning Fields) Consider a field x with the label $L = \{A : B; B :; ? : A, B\}$. For the field to be assigned to a host h, the following require-

```
if h then
    x := a
else
    x := 0
```

Listing 3.1: Read channels

ments must be met:

$$\{A : B; B :\} \sqsubseteq C_h \text{ and } I_h \sqsubseteq \{? : A, B\}$$

This states that h must have the confidentiality and integrity of both A and B. For instance if h has:

$$C_h = \{A :; B :\} \quad \text{and} \quad I_h = \{? : A, B\}$$

The requirements are met as C_h is more restrictive than $C(L)$, and I_h is less restrictive than $I(L)$.

Note that the Distributed Label Model was used to express the trust relation of h and the principals A and B. The trust model of Secure Program Partitioning will later be dealt with more thoroughly.

Requirement (3.1) is not sufficient if we want to prevent *read channels*. The host to whom the field is assigned to, can reason about the location of the program counter. Suppose the field a in the program in Listing 3.1 is assigned to a principal p. If the program is being executed on another host, information about h is implicitly leaked to principal p, based on the read request. □

To avoid read channels the requirements are extended. The confidentiality of the *block label* (cf. Section 2.2.1) in each use of the variable must be less restrictive than the confidentiality of the host. Combined with the previous constraint this becomes:

$$C(L_f) \sqcup \text{Loc}_f \sqsubseteq C_h \tag{3.2}$$

Loc_f is the least upper bound of all block labels where f is read.

$$\text{Loc}_f = C(bl(u_1) \sqcup bl(u_2) \sqcup \cdots \sqcup bl(u_n))$$

where, $u_1, u_2, \ldots, u_n$ are locations of uses of f, and the function bl extracts the block label at a particular location.

```
a : int {Alice: ; ?:Alice}
b : int {Bob: ; ?:Bob}
sum : int {Alice: ; Bob: ; ?:}

sum := a + b;
```

Listing 3.2: Assigning statements

3.1.2 Assigning statements

A statement S can be assigned to a host h if the host has at least the confidentiality of all values used in the statement. Additionally the host must have the integrity of all values defined. To ensure this, two constraint labels are introduced

$$L_{in} = \bigsqcup_{v \in U(S)} L_v \quad \text{and} \quad L_{out} = \bigsqcap_{l \in D(S)} L_l$$

where, $U(S)$ denotes all values used in S, and $D(S)$ denotes all definitions in S.

For a host h to be able to execute S, the following constraints must be satisfied:

$$C(L_{in}) \sqsubseteq C_h \quad \text{and} \quad I_h \sqsubseteq I(L_{out}) \tag{3.3}$$

Example 3.2 (Example: Assigning statements) The small program in Listing 3.2 will now be looked at. In the program two values have to be added, Bob and Alice do not trust the host of each other. However, both of them trust host T.

$$\begin{aligned} L_A &= \{\text{Alice} :; ? : \text{Alice}\} \\ L_B &= \{\text{Bob} :; ? : \text{Bob}\} \\ L_T &= \{\text{Alice} :; \text{Bob} :; ? : \text{Alice}\} \end{aligned}$$

The two labels L_{in} and L_{out} are now generated for the sum statement.

$$\begin{aligned} L_{in} &= \{\text{Alice} :; ? : \text{Alice}\} \sqcup \{\text{Bob} :; ? : \text{Bob}\} = \{\text{Alice} :; \text{Bob} :; ? :\} \\ L_{out} &= \{\text{Alice} :; \text{Bob} :; ? :\} \end{aligned}$$

Only host T is able to execute the statement.

$$C(L_{in}) = \{\text{Alice} :; \text{Bob} :\} \sqsubseteq \mathrm{C(L_T)}$$
$$I(L_T) \sqsubseteq \{? :\} = I(L_{out})$$

□

3.1.3 Declassification

A special case arises when declassification is performed. All the principals whose authority is needed to declassify a label, need to be sure that the program arrived at the declassification properly.

The block label contains the information about the principals who trust the integrity of the program execution at a particular point. In order to perform a declassification, we make sure that all principals P, whose authority is needed, trust the execution.

$$I(\underline{bl}) \sqsubseteq I_P \tag{3.4}$$

where I_P is the label $\{? : \underline{P}\}$.

3.2 Optimal split

Splitting programs might produce multiple solutions. To minimize execution time, some sort of optimization algorithm can be employed. The main focus is to minimize the remote control transfers and field accesses. In [ZZNM01] dynamic programming and a weighted control graph (e.g. statement in loops are weighted higher) is used to perform optimizations.

In Chapter 5, optimizations with regard to security will be investigated, as partial trust is introduced.

```
getField  : Host × FieldId → val
setField  : Host × FieldId → val
forward   : Host × VarId × val → void
rgoto     : Host × PC × Token → void
lgoto     : Token → void
sync      : Host × PC × token → Token

Token     : { Host × PC × HashVal × Nonce }[k_h]
```

Listing 3.3: Run-time interface

3.3 Run-Time Interface

Once the program has been partitioned, the Secure Program Partitioning framework adds synchronization statements to allow data exchange, and to ensure the integrity of the program execution. The run-time interface in listing 3.3 consists of:

getField Get value of a field from a remote principal.

setField Set value of a field on a remote host.

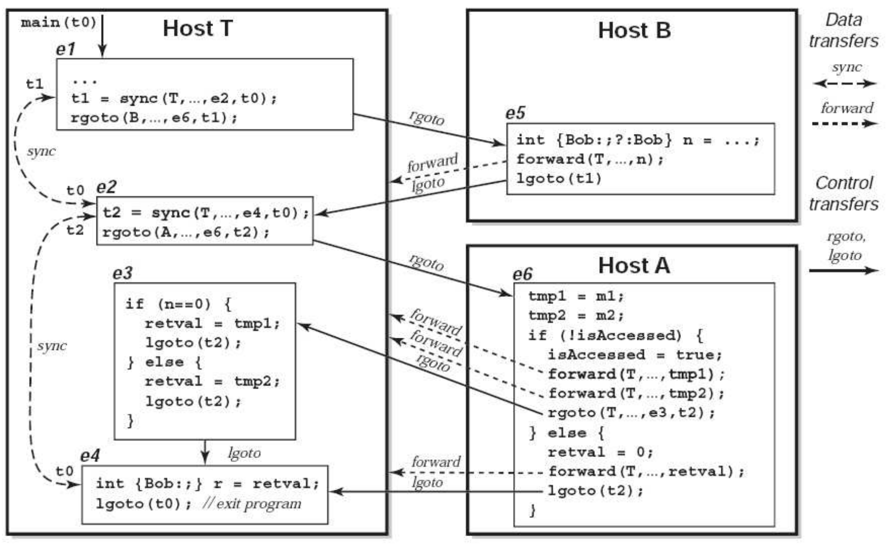
forward A field and its value is forwarded to another host.

rgoto Regular goto. The code at PC is invoked on a remote host. The host doing the *rgoto* must have a combined integrity as high as the code being invoked.

lgoto Linear goto. Transfers control from one code segment, to one with higher integrity. The host transferring control must present a *capability token* (see below), in order to perform the operation.

sync Synchronize execution. Generates the needed *capability token* for entering the following code segment.

The *capability tokens* are used to verify that the principal is allowed to invoke the code segment. All principals share an *integrity control stack*, where the



```
...
t1 = sync(T,...,e2,t0);
rgoto(B,...,e6,t1);
```

```
int {Bob:;?:Bob} n = ...;
forward(T,...,n);
lgoto(t1)
```

```
t2 = sync(T,...,e4,t0);
rgoto(A,...,e6,t2);
```

```
if (n==0) {
  retval = tmp1;
  lgoto(t2);
} else {
  retval = tmp2;
  lgoto(t2);
}
```

```
int {Bob:;} r = retval;
lgoto(t0); //exit program
```

```
tmp1 = m1;
tmp2 = m2;
if (!isAccessed) {
  isAccessed = true;
  forward(T,...,tmp1);
  forward(T,...,tmp2);
  rgoto(T,...,e3,t2);
} else {
  retval = 0;
  forward(T,...,retval);
  lgoto(t2);
}
```

Figure 3.2: Control Flow Graph

principal being invoked can check the token sent to it. The stack must then be popped in the right order, in order for the program to be executed.

In Figure 3.2 (Taken from [ZZNM01]) a control flow graph can be seen.

The run-time interface does not change when moving to the dynamic setting. Therefore, it will not be investigated further.

3.4 Trust relation

In [ZZNM01] a very simple trust model is used. Trust is between principals and hosts, and divided into integrity and confidentiality. The principals who trust a host h are expressed as:

$$C_h = \{p_1 :; ...; p_n :\} \quad \text{and} \quad I_h = \{q_1 :; ...; q_n :\} \tag{3.5}$$

$p_1, ..., p_n$ trust h to protect the *confidentiality* of their data, while $q_1, ..., q_n$ trust h to protect the *integrity* of their data. The advantage of this trust model is its simplicity, as it uses labels of the Distributed Label Model to denote trust.

The disadvantage of this model is that it is static and too simple for many applications. For a dynamic distributed system a dynamic model is needed. In Chapter 5 a more suiting and dynamic model is presented.

The trust model only deals with trust between principals and hosts. This is an unrealistic model, and unnecessarily complicated. In realistic scenarios you express your trust in principals, like

- User: If a principal trust a user, he implicitly trusts the user's host (or hosts).
- Server process: A principal can have trust in a process running on a server.

The division between hosts and principals is even superfluous, as hosts can be viewed as principals. In this thesis only trust between principals will be considered.

Chapter 4

The sflow Language

A language that doesn't have everything is
actually easier to program in than some that do.
– Dennis M. Ritchie

In this chapter a simple language with Decentralized Label Model support is presented. The language must support the dynamic extensions proposed in the next chapter. The language is a small, simple, customized language. It is inspired by the While language from [NNH99], but has been extended to support the Decentralized Label Model. It is kept small, so the verifier and splitter can be developed without having to consider the details of a complex language with information flow support, such as JIF [Mye99].

The lightweight *sflow* (short for *simple flow language*) can be used to exhibit all the important properties of *Secure Program Partitioning* .

4.1 Grammar

The language is defined by the grammar in Figure 4.1. In the figure security labels are denoted with l. Labels consist of a confidentiality (cl) and an integrity (il) part.

$$\begin{array}{rcl} e & ::= & n \mid x \mid e_1 \ \texttt{op} \ e_2 \mid \texttt{declassify}(e, cl) \mid \texttt{endorse}(e, il) \\ c & ::= & \texttt{skip} \mid \texttt{int}\ l\ x \mid x := e \mid c_1; c_2 \\ & & \mid \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{while}\ e\ \texttt{do}\ c \end{array}$$

Figure 4.1: Grammar for the *sflow* language

The grammar does not contain methods or classes, as these are not needed to experiment with the framework. A simple, sequential language is sufficient.

4.2 Operational semantics

The semantics for the language can be seen in Figure 4.2. The ψ function in the figure evaluates arithmetic expressions.

$$\begin{array}{rcl} \langle M, x := e\rangle & \Rightarrow & \langle M[x \mapsto \psi(e)], \texttt{skip}\rangle \\ \langle M, \texttt{skip}; c\rangle & \Rightarrow & \langle M, c\rangle \\ \langle M, c_1; c_2\rangle & \Rightarrow & \langle M', c_1'; c_2\rangle \quad (\textit{if}\ \langle M, c_1\rangle \Rightarrow \langle M', c_1'\rangle) \\ \langle M, \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2\rangle & \Rightarrow & \langle M, c_1\rangle \qquad (\textit{if}\ \psi(e) \neq 0) \\ \langle M, \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2\rangle & \Rightarrow & \langle M, c_2\rangle \qquad (\textit{if}\ \psi(e) = 0) \\ \langle M, \texttt{while}\ e\ \texttt{do}\ c\rangle & \Rightarrow & \langle M, c;\ \texttt{while}\ e'\ \texttt{do}\ c\rangle \quad (\textit{if}\ \psi(e) \neq 0) \\ \langle M, \texttt{while}\ e\ \texttt{do}\ c\rangle & \Rightarrow & \langle M, \texttt{skip}\rangle \quad (\textit{if}\ \psi(e) = 0) \end{array}$$

Figure 4.2: Operational semantics for the *sflow* language

$$\frac{}{\Gamma, bl \vdash n : \bot} \qquad \frac{\Gamma(x) = l}{\Gamma, bl \vdash x : l}$$

$$\frac{\Gamma, bl \vdash e_1 : l_1 \qquad \Gamma, bl \vdash e_2 : l_2}{\Gamma, bl \vdash e_1 \ \texttt{op} \ e_2 : l_1 \sqcup l_2}$$

$$\frac{I(bl) \subseteq \{? : P_{declassify}\} \qquad \mathit{authority}(P_{declassify})}{\Gamma, bl \vdash \texttt{declassify}(e, l)}$$

$$\frac{I(bl) \subseteq \{? : P_{endorse}\} \qquad \mathit{authority}(P_{endorse})}{\Gamma, bl \vdash \texttt{endorse}(e, l)}$$

Figure 4.3: Typing rules for expressions.

When variables are declared, the security environment Γ is updated.

$$\langle M, \Gamma, \texttt{int}\ l\ x\rangle \Rightarrow \langle M[x \mapsto 0], \Gamma[x \mapsto l], \mathit{skip}\rangle$$

The operational semantics define how the language will be interpreted. The type system presented in the coming section, will ensure *secure information flow.*

4.3 Type system

The type system ensures that no illegal information flow occurs. The type system is inspired by [VSI96, MSZ04].

The type rules for expressions can be seen in Figure 4.3. The sets $P_{declassify}$ and $P_{endorse}$ denote the principals, whose authority (i.e. owners of the policies) is needed to downgrade the policy. The predicate *authority* checks if the principal performing the downgrade has the authority needed. In our case this will be determined by the trust model.

The other premise, $I(bl) \subseteq \{? : P_{declassify}\}$, ensures the owners that the program arrived at the downgrade statement correctly, i.e. they have trust in the integrity of the program execution (cf. Section 3.1.3).

$$\frac{}{\Gamma, bl \vdash \texttt{skip}} \qquad \frac{\Gamma, bl \vdash e : l \qquad l \sqcup bl \sqsubseteq \Gamma(x)}{\Gamma, bl \vdash x := e}$$

$$\frac{\Gamma, bl \vdash c_1 \qquad \Gamma, bl \vdash c_2}{\Gamma, bl \vdash c_1; c_2} \quad \frac{\Gamma, bl \vdash e : l \qquad \Gamma, l \sqcup bl \vdash c}{\Gamma, bl \vdash \texttt{while}\ e\ \texttt{do}\ c}$$

$$\frac{\Gamma, bl \vdash e : l \qquad \Gamma, l \sqcup bl \vdash c_1 \qquad \Gamma, l \sqcup bl \vdash c_2}{\Gamma, bl \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2}$$

Figure 4.4: Typing rules for statements.

The typing rules for statements are listed in Figure 4.4. The assignment typing rule will ensure that no illegal flow occurs. By taking the *least upper bound* of the expression label and block label, we ensure that no direct or implicit flow occurs. The block label is updated in the conditional statements (if and while).

Listing 4.1 contains a valid *sflow* program. The program will be discussed later, but is included here to illustrate the syntax of *sflow* programs.

This concludes the chapter on the *sflow* language. The next chapter presents a design that uses this languages to perform *Secure Dynamic Program Partitioning*.

```
int{Alice:; ?:Alice} m1;
int{Alice:; ?:Alice} m2;
int{Alice:; ?:Alice} isAccessed;
int{Bob:} n;
int{Alice:; Bob:; ?:Alice} val;
int{Bob:} return_val;

if isAccessed then {
    val := 0;
}
else {
    isAccessed := 1;
    if endorse(n,{?:Alice}) = 1 then {
        val := m1;
    }
    else {
        val := m2;
    };
};
return_val := declassify(val,{Bob:});
```

Listing 4.1: Oblivious Transfer in *sflow*

Chapter 5

Extending SPP to Dynamic Networks

Security is an attempt to try to make
the universe static so that we feel safe.
– Anne Wilson Schaef

In this chapter the Secure Program Partitioning framework is extended to also deal with dynamic networks. Moving to the dynamic setting will introduce a set of new challenges which must be addressed.

When principals are allowed to enter and leave the network, the split might no longer be valid or optimal. Hence, the splitter must be able to respond to the changing network.

Furthermore, in dynamic distributed systems, the static trust model described in Section 3.4 is insufficient. Complete trust and inability to propagate trust make the model unsuitable for most realistic trust scenarios. Additionally, the inability to handle dynamic networks makes the static trust model insufficient for dynamic distributed systems.

In order to make the Secure Program Partitioning applicable to more realistic scenaros, we extend the trust model to accept changes in the network. The issue of trust propagation will be handled by introducing recommended trust. Finally the model will be extended to support partial trust. A probabilistic approach will be used to model this.

The last part of this chapter will address how to handle data when a principal leaves the network. The problem arises when data is stored on, but not owned by the leaving principal. In that case the data should become unavailable for the leaving principal.

5.1 Dynamic Splitter

In the dynamic scenario the set of available principals and the trust graph are no longer static. Principals are allowed to join and leave the network. Hereby partitioning becomes a dynamic process, where the splitter continuously must make sure that the split is valid.

5.1.1 Joining the Network

When a principal p joins the network it is added to the set of active principals P_{active}. It is obvious that any previous partitioning is still valid. The splitter, however, might want to repartition the program, if a more optimal split exists. The splitter will base its decision on two criteria :

- Security: A more secure split exists.
- Performance: Lower execution time under an alternative split.

The first criteria requires a measure of security. In the coming sections such a measure will be introduced. Figure 5.1 illustrates the process of a principal joining the network.

Performance is not the focus of this thesis, so it will not be dealt with in detail. Nevertheless, a performance measure could be introduced by evaluating the performance of each host, the network latency and bandwidth. Using this, as well as an analysis of remote calls in the split program, an approximation of execution time could be introduced. By minimizing the approximated execution time, the performance could be improved significantly in many cases.

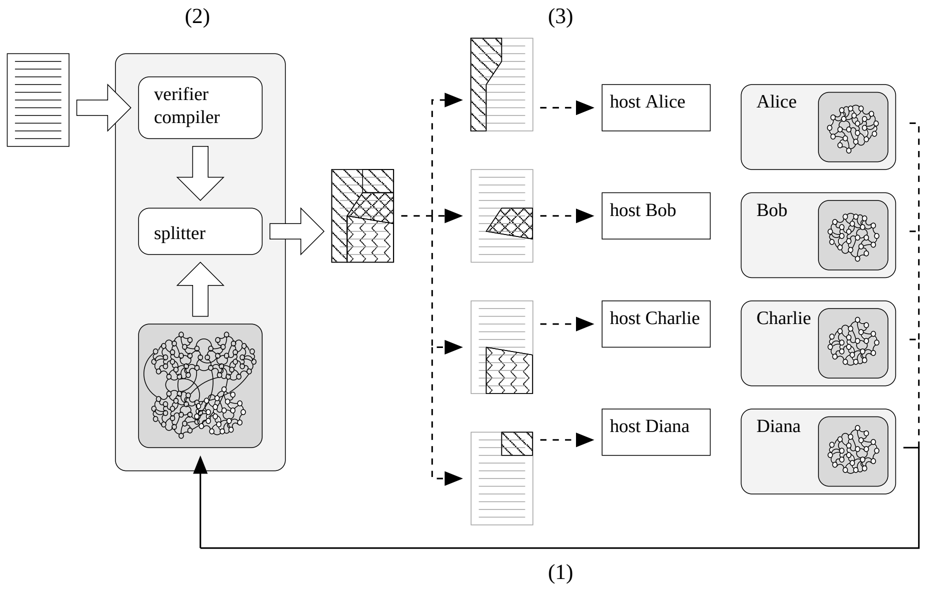


Figure 5.1: Principal Diana joins the network. The trust graph is updated (1), the program is re-split (2), and redistributed across the network (3). Compared to figure 3.1, a part of Alice's subprogram is scheduled to run Diana's host.

When a principal joins the network the splitter has to decide if the program should be resplit. In the case where program execution has already begun, several options exist. The execution could simply just continue until it is done. In many distributed systems, however, executions are continuous. In that case the execution could be halted, the variables and statement can be rescheduled, and then the program execution could be resumed. Its important that the variables are not altered or corrupted during the rescheduling, so some verification mechanism should be employed by the splitter.

The work presented in [ZCMZ03] uses hashing to ensure the integrity of the data. While only one principal has the real data, several principals can maintain hash values of the data, and thereby ensure that no corruption of the data has taken place. This could be applied to the case of rescheduling an already running program.

5.1.2 Leaving the Network

When a principal p leaves the network, p is removed from the set of active principals P_{active}. If the principal is part of the split, $p \in P_{split}$, all data and statements needs to be reassigned to other principals. If using the principals in P_{active} can not produce a valid partitioning, execution is halted. The process of leaving the network is illustrated in Figure 5.2.

If execution already has begun and the principal is part of the partitioning, it might be possible to recover the state. First of all, we make a distinction between *clean exiting*, and *dirty exiting*.

Clean exiting means that the principal is leaving the network intentionally. It notifies the splitter, and the splitter can thereafter take corresponding action, and repartition any statements or variables to other hosts.

A dirty exit is where a principal suddenly becomes unavailable (network connection is lost, computer shuts down, etc.). In this case, a repartitioning of data cannot be achieved. The data stored on the principal is forever lost. Instead of starting all over, one could roll-back to a state, where the data of the leaving principal has not been modified yet. This is only possible if the splitter stores intermediate states.

Storing intermediate states would hurt performance. Execution would have to be halted, and the splitter would have to probe each principal for its data and program counter. Nevertheless, in scenarios where a program is running for a long time, this could be a necessary precautionary step. Examples are large clusters and grid systems working on a computational intensive problem, where calculations take several days. It would be unsatisfactory to have to start over, each time a host breaks down.

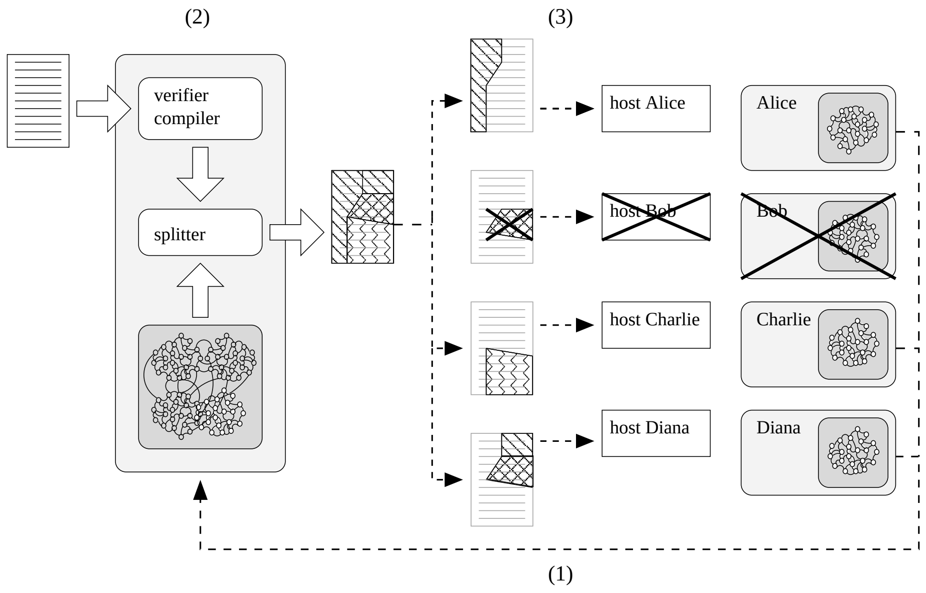


Figure 5.2: Principal Bob leaves the network. The splitter is notified (1), the program is re-split (2), and redistributed across the network (3). Compared to figure 5.1, Bob's subprogram is scheduled to run on Diana's host.

These issues are left to future work, and will not be dealt with further. The focus will instead be on finding a fitting trust model for the Secure Dynamic Program Partitioning.

5.2 Trust Model

In the coming sections, different trust models are investigated. We start with the simple model used by [ZZNM01], and then continue by making a series of motivated modifications to the model, to make it better suited for the dynamic scenario.

First, some basic concepts and assumptions are introduced.

5.2.1 Basics of Trust

Trust is belief in a principal's ability to perform a particular action correctly. Key properties of trust are [ARH97]:

- Trust is subjective.
- Trust affects those actions, which cannot be monitored.
- Trust is *non-transitive.*

In this thesis the focus will be on the propagation of trust in a distributed system. Fundamental to trust is that it is *non-transitive* (e.g. [Jøs96]), i.e. if Alice trusts Bob, and Bob trusts Charlie, then Alice does not automatically trust Charlie. This is due to the fact that trust is subjective. Alice needs to have a relation to Charlie, in order for her to trust him.

Lots of systems in use (especially authentication protocols) make the erroneous assumption that trust *is* transitive. This is a convenient assumption, but in direct violation with the basic concept of trust.

In this thesis trust is always:

- Between two principals.
- Non-symmetrical.
- Non-transitive.

In later sections, *recommended trust* will be introduced to allow propagation of trust through the network without violating the property of *non-transitivity.*

5.3 Static Trust

In [ZZNM01] trust is complete and direct. Direct because no trust propagation exists, and complete because a principal either trusts another principal completely or not at all.

Trust is divided into confidentiality and integrity. To recap, maintaining confidentiality is not to leak data, while integrity is a principals ability to handle data correctly as specified in the program.

We have already seen that this trust model can be expressed using the Decentralized Label Model.

$$C_p = \{p_1 :; \ldots ; p_n :\} \quad \text{and} \quad I_p = \{q_1 :; \ldots ; q_n :\} \tag{5.1}$$

where $p_1, \ldots, p_n$ have trust in p's confidentiality, and $q_1, \ldots, q_n$ trust that p will maintain integrity of their data.

5.4 Dynamic Trust

In the dynamic trust model principals are allowed to update their trust declarations, thus the trust graph is no longer static. Joining principals must also inform the splitter of their trust relations.

A trust declaration consists of two sets of principals.

$$TC_p = \{p_1, \ldots, p_n\} \quad \text{and} \quad TI_p = \{q_1, \ldots, q_n\} \tag{5.2}$$

TC_p is the set of principals who p trusts with regards to confidentiality, and TI_p is the set of principals whose integrity p trusts.

Using these trust declarations, the trust labels in the global trust graph are updated by the splitter.

Example 5.1 (Joining) Principal C joins the network. C has the following trust declarations:

$$TC_C = \{A, B\} \quad \text{and} \quad TI_C = \{A\}$$

The trust labels in the global trust relation are updated:

$$C_A := C_A \cup \{C :\}, \quad C_B := C_B \cup \{C :\}, \quad I_A := I_A \cup \{C :\}$$

□

In this model a principal p is responsible for maintaining his own local trust graph/relations. No one else can modify p's trust relations. This is an important improvement, as trust is not a static notion. Trust will change based on the information and experience a principal receives, so it is important that the principal is able to update his or her trust relations.

5.5 Recommended Trust

Recommendations are fundamental to most trust scenarios in our society. Trust is being propagated between people by recommendation.

The way we deal with organizations is an example of this. Because it is not possibly to know all the people we are dealing with, trust is instead put in organizations (banks, governmental institutions, etc.). We are allowing organizations to recommend individuals to handle our data, money, cases, etc. The trustworthiness of an organization is directly dependent on the trustworthiness of its individuals.

We gain trust in an organization by recommendations from others (e.g. people or other organizations). Also previous actions of the organization will have direct impact on our trust in the organization.

Building trust between principals in the real world is complex, dynamic, subjective, and not fully understood. Consequently, it is impossible to model it precisely. A model that allows propagation through recommendation can be achieved, though. Such a model will be better at handling dynamic distributed systems, than the simple, static model proposed in [ZZNM01].

Some previous work has been done in this area. [BBK94, YKB93] work with recommended trust in distributed systems, while [Mau96] works with propagation of certificates in *Public-Key Infrastructures* (PKIs) using recommendations.

This work is used as inspiration, to develop a model that supports recommendations.

In the recommended trust model, a principal A can allow another principal B to recommend principals. Recommendations are also annotated with a distance $n \geq 1$, which states how many edges away from B a recommended principal can be. If $n = 1$, only principals directly trusted by B (neighbors in the trust graph) can be recommended.

A recommendation is a statement of the form:

$$Rec_{A,B,dist}$$

Expressing that A trust B to recommend principals with a maximum distance of $dist$ from B. Trust can be declared using a similar notation:

$$Trust_{A,B}$$

Meaning A trusts B directly.

Trust graphs can be constructed as sets of these two components.

Definition 5.1 (Trust Graph) In a network with a set of principals P, a trust graph TG can be constructed from a set of statements

$$Trust_{p,q} \quad \text{and} \quad Rec_{r,s,dist} \tag{5.3}$$

where, $p, q, r, s \in P$ and $dist \in \mathbb{N}$.

Only one recommendation edge is allowed in TG for each pair of principals p, q. $\square$

A principal trusts another principal, if trust between two principals can be derived by traversing recommendation edges in the trust graph. Following definition states how trust can be derived.

Definition 5.2 (Recommended Trust) A principal p_1 trusts another principal p_n, iff

p_1 has direct trust in p_n:

$$TP_{p_1,p_n} = \langle Trust_{p_1,p_n} \rangle \tag{5.4}$$

Or, one or more recommendation paths exist

$$TP_{p_1,p_n} = \langle Rec_{p_1,p_2,d_1}, Rec_{p_2,p_3,d_2}, \ldots, Rec_{p_{n-2},p_{n-1},d_{n-2}}, Trust_{p_{n-1},p_n} \rangle \tag{5.5}$$

Where, all distances $d_i \geq n - i - 1$. $\square$

This definition states that trust can be derived, if a path of recommendations exists to a neighboring principal of the destination principal. Additionally, all the recommendation statements in the path must have sufficient recommendation distance. This is experienced in:

Corollary 5.3 *Trust to principal p_n can be derived from all principals in a trust path p_i, $i \in \{1, \ldots, n-1\}$.*

Before discussing this definition of trust, an example is presented to illustrate how the model works in a concrete scenario.

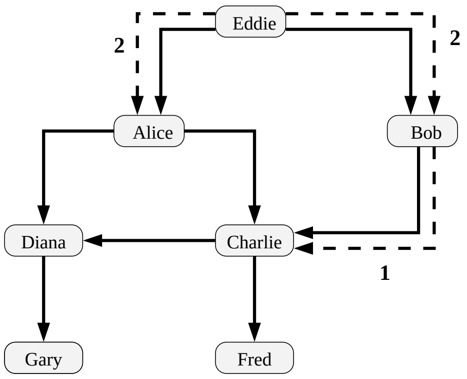


Figure 5.3: Trust graph with recommended trust. Solid lines are trust, while dashed lines are recommended trust. The numbers denote recommendation distance.

Example 5.2 (Recommended Trust) In Figure 5.3 a trust graph with recommended trust is depicted. Using Definition 5.1 the trust graph can be expressed as (only the first letter of the principal id is used):

$$\begin{aligned} TG &= \{Trust_{A,C}, Trust_{A,D}, Trust_{B,C}, Rec_{B,C,1}, Trust_{C,D}, Trust_{C,F}, \\ &\quad Trust_{D,G}, Trust_{E,A}, Rec_{E,A,2}, Trust_{E,B}, Rec_{E,B,2}\} \end{aligned}$$

Trust paths can be derived for this trust graph, for instance:

$$TP_{E,F} = \langle Rec_{E,B,2}, Rec_{B,C,1}, Trust_{C,F} \rangle$$

Eddie trusts all principals in the graph except Gary. We write $TP_{E,G} = \emptyset$ to express lack of trust.

Several trust paths might exist to the same principal, for instance from Eddie to Diana:

$$TP_{E,D} = \{\langle Rec_{E,A,2}, Trust_{A,D} \rangle, \langle Rec_{E,B,2}, Rec_{B,C,1}, Trust_{C,D} \rangle\}$$

□

5.5.1 Discussion of the Recommended Trust Model

The model proposed introduces recommendation annotated with an upper limit for the distance to the recommended principal. Two alternative approaches could be considered:

- A system-wide distance n, which all recommendation paths must obey. The central splitter would be the obvious choice for selecting n. For instance the splitter could only allow neighbors to be recommended by setting $n = 1$.
- Another approach is to have no restriction of the length of recommendation paths ($n = \infty$).

The first option seems like a reasonable choice. We choose, however, the individually specified recommendation distances, as this allows principals to manage their own level of trust. A principal might have different levels of trust in other principals ability to recommend principals, hence, the distances should be specified for each principal.

The second approach has some obvious shortcomings. When the paths become too long, the chance of the principal actually being trustworthy becomes significantly smaller. It only takes one bad recommendation edge, and a multitude of bad principals might have been recommended. The approach propagates trust too loosely, and is therefore unsuited for larger distributed systems.

These choices reflect the fundamental property that trust is individual (cf. Section 5.2.1). As a general design principle, trust should be individually specified.

It is important to realize that the principal of *non-transitivity* has not been violated. The propagation of trust is completely controlled by the individual principal, as he explicitly declares who he trusts to recommend principals, as well as the allowed distance. Thereby he is indirectly declaring trust to the recommended principals.

5.6 Confidentiality and Integrity

In the *Decentralized Label Model* there are two types of security properties, confidentiality and integrity. *Secure Program Partitioning* keeps this distinction by incorporating it into its trust model.

Our trust model also supports this two-dimensional notion of trust. The two trust types are independent, thus the trust graph will contain both confidentiality and integrity edges. Recommendation edges are also divided into confidentiality and integrity. Hence, the combined trust graph consists of two independent trust graphs, one specifying confidentiality and one specifying integrity.

As we continue developing the trust model, trust will still be dealt with as one-dimensional, as there is no difference in the way we derive trust for confidentiality and integrity.

5.7 Probabilistic Trust

The introduction of recommended trust addresses some of the challenges faced for a trust model in a dynamic distributed system. However, it still operates with complete trust. Complete trust is, however, not well-suited for real world scenarios. In the real world no one is completely trustworthy [Zim95], and different levels of trust exist.

To trust someone is a subjective choice, based on known information (has he or she been trustworthy in the past, recommendations, etc.) and risk (what are the consequences of being wrong). The higher the risk, the more trust is needed.

To model this behavior, we introduce probabilistic trust. Each trust declaration is annotated with a probability (as usual in the range from 0 to 1), which is an assessment of the probability that the trust statement holds. Both direct trust and recommendations have probabilities.

Our model has been inspired by the probabilistic model introduced in [Mau96]. Maurer deals with certifying certificates in public key infrastructures, while our model deals with trust in a distributed system. Thus the two models differ in a few important aspects. These differences will be discussed after the model has been introduced.

The probabilistic model extends the recommended trust model. Each edge/ statement now also has a probability ϕ.

Definition 5.4 (Probabilstic Trust Graph) In a network with a set of principals P, a trust graph TG can be constructed from a set of statements

$$Trust_{a,b} \quad \text{and} \quad Rec_{c,d,dist} \tag{5.6}$$

where $a, b, c, d \in P$, $dist \in \mathbb{N}$. Each statement S has a probability:

$$p(S) = \phi \tag{5.7}$$

where $\phi \in [0, 1]$.

Only one trust edge is allowed in TG for each principal pair a, b. TG can contain several recommendation edges for a principal pair a, b, one for each recommendation distance $dist$. If $Rec_{a,b,dist}$ is valid, so is $Rec_{a,b,dist'}$, $dist' < dist$. $\Box$

Trust paths have the same structure as in the deterministic model. They can either represent direct trust

$$TP_{p_1,p_n} = \langle Trust_{p_1,p_n} \rangle$$

or a recommendation path:

$$TP_{p_1,p_n} = \langle Rec_{p_1,p_2,d_1}, Rec_{p_2,p_3,d_2}, \ldots, Rec_{p_{n-2},p_{n-1},d_{n-2}}, Trust_{p_{n-1},p_n} \rangle$$

where all distances $d_i \geq n - i - 1$.

The probability of a path being valid is the probability of all statements in the path being valid. As the statements are disjoint, the probability can be calculated as:

$$P(TP_{p_1,p_n} \subseteq TG) = \prod_{S \in TP_{p_1,p_n}} p(S) \qquad (5.8)$$

Several paths can exist between two principals, p_a, p_b. Intuitively each extra path should increase the probability of p_a trusting p_b. However, some paths are subset of other paths, and in that case only the shortest of the two paths is included. If several recommendation distances exists between two nodes, the shortest recommendation distance is selected.

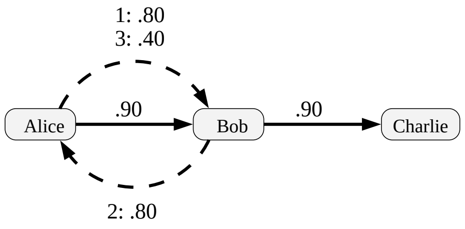


Figure 5.4: Probabilistic trust graph with multiple recommendations.

Example 5.3 (Minimal paths) In Figure 5.4 a trust graph is depicted. Several trust paths exist between Alice and Charlie:

$$\begin{aligned}\{ \quad & \langle Rec_{A,B,1}, Trust_{B,C} \rangle, \langle Rec_{A,B,3}, Trust_{B,C} \rangle, \\ & \langle Rec_{A,B,3}, Rec_{B,A,2}, Rec_{A,B,1}, Trust_{B,C} \rangle \qquad \}\end{aligned}$$

However, only one of these paths is considered minimal.

The first two paths are identical except for the recommendation distance. As stated in Definition 5.4, $Rec_{A,B,3}$ implies $Rec_{A,B,1}$. So it is clear that they should not both be included. We write:

$$\langle Rec_{A,B,1}, Trust_{B,C}\rangle \subset \langle Rec_{A,B,3}, Trust_{B,C}\rangle$$

Even more obvious is that:

$$\langle Rec_{A,B,1}, Trust_{B,C}\rangle \subset \langle Rec_{A,B,3}, Rec_{B,A,2}, Rec_{A,B,1}, Trust_{B,C}\rangle$$

So only the path $\langle Rec_{A,B,1}, Trust_{B,C}\rangle$ is minimal.

According to (5.8) the probability of this path is:

$$P(TP_{A,C} \subseteq TG) = \prod_{S \in TP_{A,C}} p(S) = p(Rec_{A,B,1}) \cdot p(Trust_{B,C}) = 0.72$$

□

Based on the previous example, a minimal trust path will be defined as:

Definition 5.5 (Minimal Trust Path) Consider two principals p_a, p_b in a trust graph TG. Then $\nu = TP_{p_a,p_b} \subseteq TG$ is considered a minimal path, iff

$$\nexists TP_{p_a,p_b} \subseteq TG : TP_{p_a,p_b} \subset \nu \tag{5.9}$$

□

When calculating confidence, only minimal trust paths will be included. However, it has still not been addressed what to do when several minimal trust paths exist between two principals.

The confidence will be calculated as the probability that one of the trust paths is valid. This can be achieved by applying probability logic.

Definition 5.6 (Confidence) Let $\nu_1, \ldots, \nu_k$ denote all minimal paths in the trust graph TG from which $Trust_{p_a,p_b}$ can be derived. Then the confidence in a trust statement $Trust_{A,B}$ is given by:

$$conf(Trust_{p_a,p_b}) = P(\bigvee_{i=1}^{k}(\nu_i \subseteq TG)) \tag{5.10}$$

□

In probability logic the probability that at least one of two events will occur, can according to [Hai84] be calculated as :

$$\begin{aligned} P(A \vee B) &= P(A) + P(B) - P(A \wedge B) \\ &= P(A) + P(B) - P(B|A) \end{aligned} \tag{5.11}$$

For whole series of k events, the *inclusion-exclusion principle* can be applied [Mau96]. This is equivalent to expanding (5.11) for all k events. The confidence in the validity of one of the paths can be calculated from the sum of probabilities of the k events, subtracting the $\binom{k}{2}$ events from intersecting 2 paths, adding the $\binom{k}{3}$ events from intersecting 3 paths, etc. This can be written as:

$$\begin{aligned} conf(Trust_{A,B}) &= \sum_{i=1}^{k} P(\nu_i \subseteq TG) \\ &- \sum_{1 \leq i_1 < i_2 \leq k} P((\nu_{i_1} \cup \nu_{i_2}) \subseteq TG) \\ &+ \sum_{1 \leq i_1 < i_2 < i_3 \leq k} P((\nu_{i1} \cup \nu_{i_2} \cup \nu_{i_3}) \subseteq TG) \\ &- \cdots \end{aligned} \tag{5.12}$$

The complexity of calculating the confidence for a statement $Trust_{p_a,p_b}$ is of the order 2^k, where k is the number of minimal paths. So for large graphs with many trust paths, the calculation becomes infeasible. In that case sensitivity analysis should be applied, to find paths which have only marginal influence on the final confidence value, and discard them. Doing this is *safe* as it can never *increase* the confidence.

In the probabilistic model, trust becomes a confidence value. So instead of being binary (trust or no trust), it becomes a continuous scale. To determine a sufficient confidence value, each user must specify the minimum confidence level. For instance a user might require a confidence level of 0.95 to trust someone.

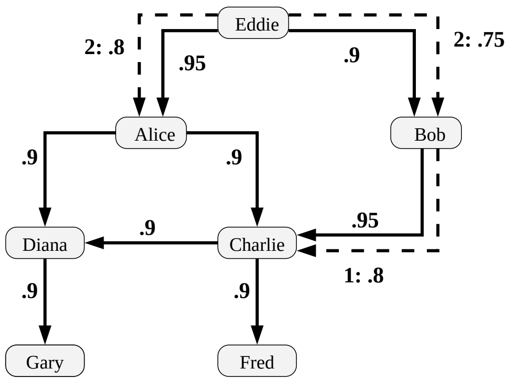


Figure 5.5: Probabilistic trust graph.

Allowing each user to specify his or her own trust level follows the design principle of individually specified trust (cf. Section 5.5.1). We require all principals in the trust graph to have a threshold of confidence. The confidence thresholds will be represented as a mapping,

$$[p_1 \mapsto \tau_1, p_2 \mapsto \tau_2, \ldots, p_n \mapsto \tau_n] \tag{5.13}$$

where $\{p_1, p_2, ..., p_n\} = P$ (i.e. all principals are included).

It is assumed a function *threshold* exists, which finds the threshold mapping for a principal.

$$\mathit{threshold}(p) = \tau \tag{5.14}$$

For a principal p_a to trust a principal p_b, then

$$\mathit{conf}(Trus_{p_a,p_b}) \geq \mathit{threshold}(p_a) \tag{5.15}$$

Example 5.4 (Probabilistic Trust) We now return to the example with recommended trust. Trust probabilities have been added to the edges, as seen in Figure 5.5.

First, Eddie's confidence in Fred is calculated. As in Example 5.2 only one path is identified:

$$\langle Rec_{E,B,2}, Rec_{B,C,1}, Trust_{C,F} \rangle$$

As only path exists, the confidence becomes the probability of this path being valid:

$$conf(Trust_{E,F}) = p(Rec_{E,B,2}) \cdot p(Rec_{B,C,1}) \cdot p(Trust_{C,F}) = .75 \cdot .8 \cdot .9 = 0.54$$

More interesting is to find the confidence value from Eddie to Charlie.

$$\nu_1 = \langle Rec_{E,A,2}, Trust_{A,C} \rangle \quad , \quad \nu_2 = \langle Rec_{E,B,2}, Trust_{B,C} \rangle$$

The probabilities of the two paths are:

$$P(\nu_1 \subseteq TG) = p(Rec_{E,A,2}) \cdot p(Trust_{A,C}) = .8 \cdot .9 = .72$$

and

$$P(\nu_2 \subseteq TG) = p(Rec_{E,B,2}) \cdot p(Trust_{B,C}) = .75 \cdot .95 = .7125$$

The intersection[1] of the two events is:

$$\begin{aligned} p((\nu_1 \cup \nu_2) \subseteq TG) &= p(Rec_{E,A,2}) \cdot p(Trust_{A,C}) \cdot p(Rec_{E,B,2}) \cdot p(Trust_{B,C}) \\ &= .72 \cdot .7125 = .513 \end{aligned}$$

Using formula (5.12), the confidence is calculated:

$$\begin{aligned} conf(Trust_{E,C}) &= P(\nu_1 \subseteq TG) + P(\nu_2 \subseteq TG) - p((\nu_1 \cup \nu_2) \subseteq TG) \\ &= .9195 \end{aligned}$$

Finally, confidence in $Trust_{E,D}$ will be calculated. The following minimal paths are identified:

$$\nu_1 = \langle Rec_{E,A,2}, Trust_{A,D} \rangle \quad , \quad \nu_2 = \langle Rec_{E,B,2}, Rec_{B,C,1}, Trust_{C,D} \rangle$$

Resulting in:

$$\begin{aligned} conf(Trust_{E,D}) &= P(\nu_1 \subseteq TG) + P(\nu_2 \subseteq TG) - p((\nu_1 \cup \nu_2) \subseteq TG) \\ &= 0.8712 \end{aligned}$$

As a result, if Eddie has a minimal confidence threshold of 0.9, the following trust statements can be derived:

$$\{Trust_{E,A}, Trust_{E,B}, Trust_{E,C}\}$$

□

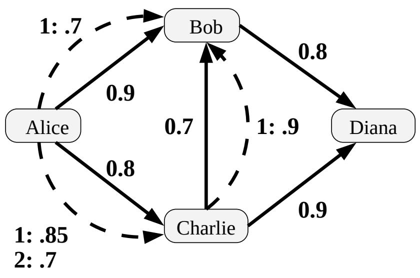


Figure 5.6: Probabilistic trust graph.

Example 5.5 (Overlapping Paths) In this example the trust graph in Figure 5.6 is considered. The confidence in $Trust_{A,D}$ will be calculated. However, it is slightly more advanced than the previous example, as there are three minimal trust paths, and the paths overlap each other:

$$\begin{aligned}\nu_1 &= \langle Rec_{A,B,1}, Trust_{B,D}\rangle \\ \nu_2 &= \langle Rec_{A,C,1}, Trust_{C,D}\rangle \\ \nu_3 &= \langle Rec_{A,C,2}, Rec_{C,B,1}, Trust_{B,D}\rangle\end{aligned}$$

Using (5.12) the confidentiality is calculated as:

$$\begin{aligned}conf(Trust_{A,D}) &= P(\nu_1 \subseteq TG) + P(\nu_2 \subseteq TG) + P(\nu_3 \subseteq TG) \\ &\quad -P((\nu_1 \cup \nu_2) \subseteq TG) - P((\nu_1 \cup \nu_3) \subseteq TG) \\ &\quad -P((\nu_2 \cup \nu_3) \subseteq TG) + P((\nu_1 \cup \nu_2 \cup \nu_3) \subseteq TG)\end{aligned}$$

As $\nu_2 \cup \nu_3 = \langle Rec_{A,B,1}, Rec_{A,C,2}, Rec_{C,B,1}, Trust_{B,D}\rangle$, we get

$$\begin{aligned}P((\nu_1 \cup \nu_3) \subseteq TG) &= p(Rec_{A,B,1}) \cdot p(Rec_{A,C,2}) \cdot p(Rec_{C,B,1}) \cdot p(Trust_{B,D}) \\ &= .7 \cdot .7 \cdot .9 \cdot .8 = .3528\end{aligned}$$

The rest of the calculations are done in a similar manner as before. The confidence becomes:

$$conf(Trust_{A,D}) = .932132$$

□

[1] We use the union of two paths, two find the probability of the probability that both events occur

Level	Probability
Full	1.00
Very high	0.95
High	0.90
Medium	0.80
Low	0.40
None	0.00

Table 5.1: Example of trust levels

5.7.1 Discussion of the Probabilistic Model

The probabilistic trust model introduces several concepts intended to make the trust model more realistic. Outside the boundaries of distributed systems, trust is often gained by recommendations from other principals. The model incorporates this by introducing recommendation edges, where a principal explicitly can declare trust in another principal's ability to recommend principals.

As discussed in the previous model (cf. Section 5.5.1), an upper limit for the recommendation distance is introduced to limit the propagation of trust in the trust network.

In the real world trust is almost never complete. However, most trust models do not reflect this. In our model, partial trust is modeled using probabilities like in [Mau96]. Alternatives are for example *Fuzzy Logic* [JS97], or *Network Flow* from *Graph Theory* [BM76]. However, probability was chosen because it better resembles the way we approach trust. Trust is always an assessment, and probabilities are a precise way of defining how likely it is that your assessment is correct. Probabilistic trust also allows an elegant and mathematical founded way of propagating trust through the network.

From a user's perspective, assigning probabilities to other principals is not an easy task. Instead trust levels should be defined, where each trust level has a probability. An example of a set of trust levels can be seen in Table 5.1.

Our model differs significantly from the model presented in [Mau96]. Maurer deals with certificates, while we deal with trust with regards to confidentiality and integrity. Certificates have to be issued by an authority. In order for a principal to have trust in a certificate, the principal must have trust in both the issuing principal, and the certificate holder.

In our model the recommended principal can be trustworthy, even though the recommender is not. So the dependency that exists for certificates, does not exist in our scenario. In practice this means only recommended paths are traversed when recommendation paths are constructed. E.g. a trust path

$$\langle Rec_{A,B,1}, Trust_{B,C} \rangle$$

would be calculated by taking

$$conf(Trust_{A,C}) = p(Rec_{A,B,1}) \cdot p(Trust_{B,C})$$

In the case of certificates, the principal B also needs to be trustworthy, so the confidentiality in all intermediate principals should be calculated:

$$conf(Trust_{A,C}) = conf(Trust_{A,B}) \cdot p(Rec_{A,B,1}) \cdot p(Trust_{B,C})$$

Propagation of trust in our model is less sensitive to malicious principals. One malicious principal in Maurer's model means that all certificates issued by this principal is invalid. In our model, principals are not automatically considered untrustworthy when they are recommended by an untrustworthy principal.

There are several possible extensions to this model. A few are discussed below. These extensions are orthogonal to our approach, and could be added to the framework, without making any changes to the original framework.

In a running version of Secure Dynamic Program Partitioning, it would be practical to use a Public Key Infrastructure as the underlying structure for identifying principals. However, this will be left as future work, but the design must support such an extension.

As mentioned, in our approach trust is explicitly declared either as direct trust edges or recommendation edges. However, you could introduce *automated trust negotiation* [WSJ00, WYS$^+$02]. In this approach trust is negotiated automatically between principals. A common approach is to use credentials. A principal's credentials might contain information about the principals identity, who trusts the principal, and the digital rights the principal has. Based on this, trust can be inferred.

The trust models in this thesis only deal with confidentiality and integrity. However, there is another aspect to trust, *availability*. Availability is the chance that a principal or host is available [ZH99]. In our framework the splitter always has full overview of the availability of the principals, so the aspect of availability has been left out. But in more realistic scenarios it might be beneficial to introduce availability. In our model this can easily be modeled as edges with probabilities.

5.8 Quality of a Partitioning

Often programs have several valid partitionings. These partitionings obey all security policies in the program, by construction. However, they do not have identical properties, so the splitter must decide which partitioning that best fits the applications purpose. This can be viewed as a matter of optimization. Several possibilities for what to optimize exist:

- Minimal network communication as proposed in [ZZNM01].
- Optimal performance by scheduling hard computations to strong principals.
- Spread sensitive data on multiple principals, to limit consequences of a principal being compromised.

Applications have different preferences of what to optimize. The framework should be parameterized with an optimization component, so the individual application can specify his or her own optimization method.

In this thesis two optimization methods that utilize the probabilistic model will be implemented, namely:

- Highest confidence.
- An individual metric for judging the split.

The two methods will be introduced in the coming sections.

5.8.1 Highest Confidence

The highest confidence method is fairly simple. For a statement or a field, the principal that all the owners can best agree on is selected.

Suppose O is the set of principals that owns the statement S (owner of data being read or written). The statement can be assigned to principals in the set A. The optimal assignment is the one where the lowest confidence of any principal in O, is the highest. This can be expressed as:

$$optimal(A, O) = max_{\forall q \in A}(min_{\forall p \in O}(conf(Trust_{p,q})) \tag{5.16}$$

This optimization method is simple and efficient. It uses the already computed confidence value to find the principal which the principals best agree on. However, another optimization method that considers the interdependencies of a data leak has been conceived.

5.8.2 The Metric

This section introduces a metric to judge the quality of a partitioning from the view point of a principal A. The idea is to judge the risk of a set of principals violating the confidentiality of data that is owned by A and has been partitioned to B. This work was presented first time in [SPJH06].

Assume that a piece of data or some code owned by principal A should be scheduled to B. How would A evaluate whether or not B is well-suited to host A's data. Again there are two components to this question. First of all, A will want to inspect which other principals might allow the splitter to partition their code to B. Any other code running on the host might result in an increased risk of A's data being leaked. Another principal might find a way to tamper with the execution platform, and hereby leak the data owned by A.

Tthe execution platform is considered part of the *trusted code base* (TCB), it is difficult to guard against all attacks. For instance if you have full access to a computer, you can circumvent the execution platform by reading directly from memory (cf. Section 2.5 on *covert channels*). Preferably, a principal would schedule its data and code to a principal it trusts fully, and no other principals will schedule their code or data to the same principal.

The confidence that principal A has in a principal B not leaking his data, is the confidence value:

$$\mathit{conf}(\mathit{Trust}_{A,B})$$

This can be computed by applying equation (5.12). Obviously, the risk that B leaks the data is the inverse of the confidence value, namely $1 - \mathit{conf}(\mathit{Trust}_{A,B})$.

In computing the probability that *any* of the principals who trust B will leak A's data, we face the problem that these events are certainly not independent, so that it is hard to compute the probability for the case that this happens. However, the metric just needs to compare different scenarios, but does not need to be a probability. So the first part of the metric is to find the average of the inverse of all *conf* values as defined above

$$\mathit{leak}(A,B) = \frac{\sum_{p \in T_B}(1 - \mathit{conf}(\mathit{Trust}_{A,p}))}{N} \tag{5.17}$$

where T_B is the principals who can schedule their data and statements to B, and N is the cardinality of the set T_B.

The higher the value of $\mathit{leak}(A,B)$, the less confidence A has in other principals that might schedule data and statements to B.

The second component is the confidence that principal A has in principal B, a measure for how likely it is that B leaks data owned by A.

The metric we suggest is defined as the quotient of these two numbers:

$$M(A,B) = \frac{\mathit{leak}(A,B)}{\mathit{conf}(Trust_{A,B})} \tag{5.18}$$

The higher the metric, the higher A judges the likelihood that B will leak its data if stored on B, thus preferably the metric should be low. As a principal always have full trust in him- or herself, the metric $M(A,A)$ is 0.

5.8.3 Applying the Metric

The metric can be calculated for both confidentiality and integrity. However, which of the two to choose depends on the situation. Principals whose data is read, want the confidentiality metric to be high, while principals whose data is defined, would like the integrity to be high (similar to the constraints in Secure Program Partitioning, see Section 3.1). So the metric is calculated, based on whether data is being defined or used.

When assigning fields, both the metric for integrity and confidentiality will be calculated for all owners of the field.

The metric is an individual measurement of another principal's trust worthiness, based on the probabilities. However, as data can have multiple owners, it has to be considered which principal to choose when the owners do not have the same preference.

Several options exist for selecting the principal:

- The average metric should be as low as possible.
- The highest metric of any of the principals is kept as low as possible.
- Weighing for instance, confidentiality higher than integrity.
- Weighing principals differently.

However, we want to maintain *fairness*, meaning no principal is considered more important than any other. This is best satisfied by the second option, as no principal (or group of principals) has precedence over another.

If A is the set of principals the statement can be assigned to, and O is the set of owners of the data, then the optimal metric becomes

$$optimal(A, O) = min_{\forall p \in A}(max_{\forall q \in O}(M(q, p))) \tag{5.19}$$

For each of the possible assignments, the highest metric is computed. The statement will be assigned to the principal who has the lowest of the highest metrics.

Example 5.6 (Calculating the Metric) Consider the following program:

```
int{A:} a; int{B:} b;
a := b;
```

The program shall be split under the trust graph in Figure 5.7. Both *Alice* and *Bob* have a trust threshold of 0.80 (the minimal trust probability they accept).

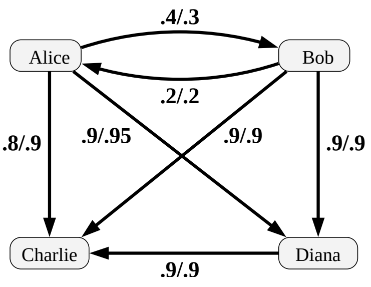


Figure 5.7: Trust graph for the metric example. Each edge is annotated with both confidentiality and integrity (C/I).

It is fairly obvious that the statement *a := b* must be assigned to either *Charlie* or *Diana*. However, to decide which one to choose the metrics is calculated for both *Alice* and *Bob*. From Alice to Charlie the leak is calculated from:

$$leak(A, C) = \frac{\sum_{p \in T_C} (1 - conf(TrustI_{A,p}))}{N}$$

TrustI denotes trust in the integrity.

Bob and *Diana* might also schedule their data to *Charlie*, thus T_C is $\{B, D\}$. So the *leak* becomes:

$$leak(A, C) = \frac{(1 - 0.3) + (1 - 0.95)}{2} = 0.375$$

To find the metric, we divide with *Alice*'s confidence (integrity) in *Charlie*,

$$M(A, C) = \frac{0.375}{0.9} = 0.417$$

Finding the metric for *Diana* is easier, as only *Bob* trusts her, besides from *Alice.*

$$M(A,D) = \frac{0.7}{0.95} = 0.737$$

Alice prefers *Charlie*, because $M(A,C) < M(A,D)$. But before the statement is assigned, we also need to find the metrics for *Bob.*

$$\begin{aligned} M(B,C) &= \frac{((1-0.2)+(1-0.9))/2}{0.9} = 0.5 \\ M(B,D) &= \frac{1-0.2}{0.9} = 0.889 \end{aligned}$$

The highest metric for the two possibilities is compared. The lowest of the two is selected. If assigned to *Charlie* the value becomes 0.5, while for *Diana* it is 0.889. The statement is assigned to *Charlie.* □

5.9 Erasure Policies

Another important issue to consider when moving to a dynamic setting, is how to handle data on unavailable hosts, what happens to data when it gets rescheduled to another principal.

The motivation is that data should only be available on one principal, and become unavailable when principals leave the network. Otherwise a principal might find a way to access data, which is not intended for him. To make this less likely, data will only be available at a principal, when it is strictly necessary. Data should be made unavailable, when

- Data is rescheduled.
- Principal leaves the network.
- Program execution ends.

As proposed in [HP05], *Erasure Policies* [CM05] could be employed to control availability of data. Erasure policies have the form:

$$l_1 \, c\!\!\nearrow\! l_2 \tag{5.20}$$

where l_1 and l_2 are security labels, $l_1 \sqsubseteq l_2$ and $l_1 \neq l_2$. When condition c is fulfilled, the policy l_2 applies, and otherwise l_1.

The article [CM05] introduces a type system, which has support for these dynamic policies. The SDPP framework should be extended to support these policies, as this would allow us to make data unavailable when principals leave, and when data is rescheduled.

Data which is scheduled to a principal p, but not solely owned by p, must be made unavailable when the principal is not available any longer. To achieve this the framework must support the two predicates:

- $active(p)$ – Checks if the principal p is currently active in the network.
- $scheduledTo(var, p)$ – Checks if the variable var is scheduled to the principal p.

The following Erasure Policy will make sure that the data can only be accessed as long as the principal is active on the network.

$$var \ : \ l \ _{\neg active(p) \nearrow} \top \tag{5.21}$$

Relabeling the data to the top element $\top$ is in effect equivalent to making the data unavailable, as it cannot be read by any principal.

Similarly, the data should be made unavailable, when data is rescheduled to another principal.

$$var \ : \ l \ _{\neg scheduledTo(var,p) \nearrow} \top \tag{5.22}$$

The splitter will during partitioning add erasure policies to the variables. If the principal is not the sole owner of the data scheduled to him or her, erasure policies are added.

Example 5.7 (Adding erasure policies) A variable n is owned by both A and B. It is scheduled to A, but the splitter adds an erasure policy, if A becomes unavailable, or if the variable is rescheduled.

```
int {A:;B:}c↗⊤ n;
```

where $c = \neg active(A) \vee \neg scheduledTo(n, A)$. $\square$

For the erasure policies to work they have to be build into the middle ware (*trusted code base*). The middle ware must perform checks of the two predicates, each time the data is read or written to. These run-time checks are necessary as the availablity of a principal can never be guaranteed.

5.10 Decentralized Splitting

In the framework proposed so far, it has been assumed that a central splitter, which all principals have complete trust in, exists. However, this might not always be the case. If not, decentralized splitting can be applied.

Decentralized splitting means that every principal participating has a splitter. They can then verify any split suggested by any other principal using this splitter. The trust graph is also decentralized and every principal can access it. This can be achieved by each principal maintaining the global trust graph. The trust graph must be synchronized, so every principal has the same trust graph at all times.

Example 5.8 (Decentralized Splitting) Consider two principals, an online store and a customer. The customer does not trust the store, and vice versa. Using decentralized splitting they are still able to make a transaction.

The store has a program, which transfers money from the customer's account to the store's account. This program is split by the store. The customer now uses his or her own splitter to check if the split is valid. If the split is valid, the customer knows that his personal information will be kept safe, as the split program will obey the policies imposed on the information. The transaction can then be carried out.

Compared to current scenarios for online trade, this allow you to engage in transactions, with stores you do not trust to protect your information. A significant improvement to the uncertainty of todays online transactions. $\square$

This concludes the theoretical description of the Secure Dynamic Program Partitioning framework. In the next chapter, a design for a prototype is presented, which implements the concepts from this chapter.

CHAPTER 6

Design

Never put off till run-time what you can do at compile-time.
– D. Gries

Based on the theoretical concepts presented in previous chapters, a design will be developed, which shows how Secure Dynamic Program Partitioning can work in practice.

The design is a prototype, and is intended for experimenting with the concepts. The design should, however, be kept as generic as possible, so it allows for future improvements and extensions.

The design will not be specific to any language. Modeling of the design, however, will follow the functional paradigm. Functional modeling is concise, precise, easy to understand, and can fairly easy be extended to imperative and object-oriented languages.

Before the design is specified, the design requirements will be listed.

6.1 Requirement Specification

In this section the requirements for our design are listed. The requirements are high level, and they leave room for design issues to be dealt with later on. Some of the terms and concepts are not explained when they are introduced. They will, however, be explained when the design is presented.

6.1.1 Information flow

An independent library with support for the Decentralized Label Model must be developed.

- Labels consist of a confidentiality and an integrity part.
- The operators *lowest upper bound* ($\sqcup$), *greatest lower bound* ($\sqcap$), and less restrictive ($\sqsubseteq$) must be supported.
- Labels must always be minimal, i.e. no superfluous policies exist.
- Principal identification must be extend-able. I.e. new id types can be introduced: number, certificate, etc.

6.1.2 Parser

- The parser must be able to parse sflow programs. Only well-typed sflow programs will be accepted.
- The parser produces an *abstract syntax tree* (AST) and *symbol table.*
- The symbol table contains: type, label, and a *Definition-Use chain.*

6.1.3 Verifier and Splitter

- The verifier can detect illegal data flow, both explicit and implicit.

- The splitter splits programs based on the AST, symbol table, trust model, and the principals currently active in the system.
- The trust model must be parameterized, i.e., different trust models can be used.
- If more than one possible split exist, the splitter must find the optimal split. The optimization component is also parameterized.

6.1.4 Trust model

- The trust model must support both confidentiality and integrity.
- The simple trust model from [ZZNM01] (cf. Section 5.3) and the probabilistic trust model from [SPJH06] (cf. Section 5.7) must be implemented.

6.1.5 Optimizer

- The optimizer must determine the optimal split, when multiple possible splits exist.
- Each optimizer component defines what is optimal.
- Specifically, optimization based on the metric from section 5.8.2 must be implemented.

6.1.6 System manager

A system manager tool will be developed, which is able to respond to changes in the network. Additionally, it must provide an interface for interaction with the user.

- Principals should be able to join and leave the network. When joining the trust graph must be updated. If necessary the program is re-split using the Splitter.

- Programs can be submitted either by principals, or loaded from a file.
- For testing purposes, it should be possible to load trust graphs from files.
- It should be possible to change trust model and optimizer.

6.1.7 Erasure policies

- The splitter must have the ability to add erasure policies to data.
- The *middleware* must support the two predicates *active* and *assignedTo*.
- Erasure policies must be added to data that is not assigned to its owner (only owner).

Based on these requirements, the system will be designed.

6.2 Assumptions

The design is intended as a prototype, and its purpose is to illustrate the concepts of Secure Dynamic Program Partitioning. To limit the implementation work required, the following assumptions are made about the principals and the *middleware*:

- All principals have complete trust in the splitter.
- All principals have the same middleware, which is used when executing the program. The middleware ensures that no leaks occur, and the program is run under the constraints of the *decentralized label model*. The middleware is part of the *Trusted Code Base* (TCB).
- Network communication is done using secure channels, where no data is leaked.

Based on these assumptions, the design will be developed.

```
CLabel : Principal × Principal-set
ConfLabel : CLabel-set

IntegrityLabel : Principal-set

SecurityLabel : ConfLabel × IntegrityLabel
```

Listing 6.1: Decentralized Label Model: Data types

```
lub : SecurityLabel × SecurityLabel → SecurityLabel
glb : SecurityLabel × SecurityLabel → SecurityLabel

lessRestrictive : SecurityLabel × SecurityLabel → bool

simplify : SecurityLabel → SecurityLabel
```

Listing 6.2: Decentralized Label Model: Functions

6.3 Decentralized Label Model

Labels in the decentralized label model can be represented using the data types in Listing 6.1. The data types assume that the data type Principal exists, which uniquely identifies a principal. Additionally, it is assumed that a set data type exists.

Note that a confidence label is made up from multiple policies, each with an owner and a set of readers ($\{o_1 : \underline{r_1}, ..., o_n : \underline{r_n}\}$).

The integrity label has no owner, so it can be represented simply as a principal set.

The basic operations in the decentralized label model are shown in Listing 6.2. Below the operations are described briefly. Refer to Section 2.2 for the theoretical background.

- **lub** – Least upper bound, $\sqcup$, of two labels.

```
AST : Node
Node : NodeId × Type × Node* × Leaf*
Leaf : Type × LeafData

FlowGraph : (NodeId × NodeId)−set

SymbolTable : Var $\overset{m}{\to}$ Type × SecurityLabel × DUChain

DUChain : (NodeId × NodeId−set)−set
```

Listing 6.3: Abstract syntax

- **glb** – Greatest lower bound, $\sqcap$, of two labels.

- **lessRestrictive** – Check if the first label is less or as restrictive, $\sqsubseteq$, as the second label.

- **simplify** – Simplifies a label. If a label contains multiple policies with the same owner, the label can be simplified. E.g.:

$$\{Alice : Bob, Charlie; Alice : Charlie\} \equiv \{Alice : Charlie\}$$

The last property can also be used as an invariant for all labels, to ensure a label never contains multiple policies with the same owner.

In practice this can be achieved by applying simplify each time the label is modified.

6.4 Abstract Syntax

Listing 6.3 shows the internal/abstract representation of programs. The different data structures are explained below.

6.4.1 Abstract syntax tree

Parsing the program produces an *abstract syntax tree* (AST) [ASU86]. The AST is made up from nodes, which have an id, a type, sub-nodes and leafs. The node id uniquely identifies each node.

The types are used to classify nodes. Two nodes with the same type have the same structure, i.e., same number of sub-nodes and leafs, and same order.

The following node types will be used (cf. Chapter 4):

- **Statement** – Top-most node type. Used to construct the tree.
- **Var declaration** – Declares variable, including label.
- **Assignment** – Expression is assigned to variable.
- **If-then-else** – Conditional statement based on an expression.
- **While** – Conditional statement based on an expression.
- **Expression** – Evaluates to a value.
- **Declassify** – Declassifies an expression.
- **Endorse** – Endorses an expression.

Each node is made up from a list of sub-nodes and leafs. While the nodes are used to reflect the structure of the program, the leafs contain the actual program components, i.e. values, variable names, variable types, labels and operators.

In Figure 2.2 an example of an abstract tree can be seen.

6.4.2 Flow graph

The flow graph reflects the possible execution flows of a program [ASU86]. The flow graph can be constructed by examining the AST, as the AST reflects the branching of the program. The flow graph is constructed from the function:

$$flowgraph : \text{AST} \rightarrow \text{FlowGraph}$$

The *flowgraph*-function uses the following function to find the flow of any node in the AST [NNH99]:

$$flow : \text{Node} \rightarrow \text{FlowGraph}$$

$$\begin{array}{rcl}
flow([\mathit{type}\ x]^l) & = & \emptyset \\
flow([x := a]^l) & = & \emptyset \\
flow([\mathit{skip}]^l) & = & \emptyset \\
flow(S_1; S_2) & = & flow(S_1) \cup flow(S_2) \cup \\
 & & \{(l, \mathit{init}(S_2)) | l \in final(S_1)\} \\
flow(\text{if } [e]^l \text{ then } S_1 \text{ else } S_2) & = & flow(S_1) \cup flow(S_2) \cup \\
 & & \{(l, \mathit{init}(S_1)), (l, \mathit{init}(S_2))\} \\
flow(\text{while } [e]^l \text{ do } S) & = & flow(S) \cup \{(l, \mathit{init}(S))\} \cup \\
 & & \{(l', l) | l' \in final(S)\}
\end{array}$$

The l's are unique identifiers of nodes.

The auxiliary function *init* finds the first statement in any node.

$$\mathit{init} : \text{Node} \rightarrow \text{NodeId}$$

$$\begin{array}{rcl}
\mathit{init}([\mathit{type}\ x]^l) & = & l \\
\mathit{init}([x := a]^l) & = & l \\
\mathit{init}([\mathit{skip}]^l) & = & l \\
\mathit{init}(S_1; S_2) & = & \mathit{init}(S_1) \\
\mathit{init}(\text{if } [e]^l \text{ then } S_1 \text{ else } S_2) & = & l \\
\mathit{init}(\text{while } [e]^l \text{ do } S) & = & l
\end{array}$$

And, $final$ retrieves the last statements of any node (note that an if-then-else node produces two final nodes):

$$final : \text{Node} \rightarrow \text{NodeId} - \textbf{set}$$

```
int{Alice:} n;
int{} a;

if a then
    n := n + 3;
else
    n := n + 2;

while n > 10 do
    n := n - 1;

a := 0;
```

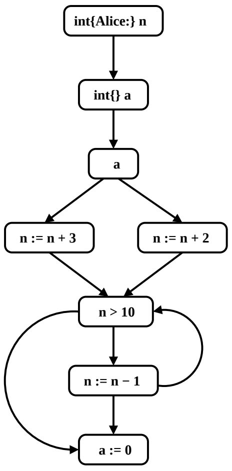


Figure 6.1: Example of a flowgraph

$$
\begin{aligned}
final([type\ x]^l) &= \{l\} \\
final([x := a]^l) &= \{l\} \\
final([skip]^l) &= \{l\} \\
final(S_1; S_2) &= final(S_2) \\
final(\text{if } [e]^l \text{ then } S_1 \text{ else } S_2) &= final(S_1) \cup final(S_2) \\
final(\text{while } [e]^l \text{ do } S) &= \{l\}
\end{aligned}
$$

Figure 6.1 depicts a small program and its flow graph

6.4.3 Symbol table

The symbol table is used to store data for variables type, value, etc. [ASU86]. In our analysis type, security label, and Definition-Use chain are needed for each variable.

So the symbol table becomes a mapping from variables to these values. The symbol table will be created during parsing.

```
1  int{Alice:} n;
2  int{} a;
3  a := 3;
4  n := 2;
5  if a then
6      n := 3 + a;
7  else
8      a := 3 + n;
9  a := a + n;
```

$$\begin{aligned}
\mathbf{a} \quad : \quad & (\mathit{def}: 3,\ \mathit{uses}: 5,6,9),\\
& (\mathit{def}: 8,\ \mathit{uses}: 9),\\
& (\mathit{def}: 9,\ \mathit{uses}:)\\
\\
\mathbf{n} \quad : \quad & (\mathit{def}: 4,\ \mathit{uses}: 8,9),\\
& (\mathit{def}: 6,\ \mathit{uses}: 9)
\end{aligned}$$

Figure 6.2: Example of Definition-Use chains

6.4.4 Definition-Use chains

With Definition-Use (or just DU) chains all locations where a variable is written and read can be represented. The function *duChain* finds the DU-chains for a variable by examining the flow graph.

$$\mathit{duChain} : \text{Var} \times \text{FlowGraph} \rightarrow \text{DUChain}$$

Listing 6.3 specifies the DUChain data type

For a variable x the DU chain is generated by traversing the flow graph. Each time a definition (e.g. assignment) is encountered, a new chain is created. From this definition all possible uses, without x being redefined, are found. This will normally result in several chains. [ASU86, NNH99] has algorithms for finding DUChains.

Figure 6.2 shows an example of DU chains.

6.5 Parser

The parser translates a file, containing sflow code, into the abstract syntax.

$$\mathit{parse} : \text{File} \rightarrow \text{AST} \times \text{SymbolTable}$$

```
prog ::= vardecl c

vardecl ::= var ";" vardecl  |  var ";"

var ::= ID "{" seclabel "}" ID

c ::= s ";" c | s ";"

s ::= ID ":=" e
    | "if" e "then" "{" c "}" "else" "{" c "}"
    | "while" e "do" "{" c "}"

e ::= VAL | ID | e OP e
    | "declassify" "(" e "," "{ cl "}" ")"
    | "endorse" "(" e "," "{ il "}" ")"

seclabel: ϵ | cl il | cl | il

cl ::= ID ":" ";" | ID ":" ";" cl
     | ID ":" idlist ";" | ID ":" idlist ";" cl

il ::= "?" ":" idlist

idlist: ID | ID "," idlist
```

Listing 6.4: BNF grammar for the *sflow* language

The parser will be generated using a parser generator. The most common approach is to use a LALR parser generator[ASU86]. Widely used LALR parser generators include Berkeley Yacc [BYA06], GNU Bison [BIS06], and CUP [CUP06].

Common for these LALR parsers is that they accept *context-free grammars*, most often in Backus-Naur form. They accept ambiguities, and will deal with them according to precedence rules. However, in general ambiguities should be avoided.

In Listing 6.4 the BNF grammar for the sflow language is shown. The grammar contains no ambiguities.

To make validation more simple, all variables must be declared initially. This small change does not change the typing rules, as the grammar still accepts only valid sflow programs (it is a subset of the original sflow grammar).

Using this grammar, programs can be parsed. As mentioned, parsing should produce both an abstract syntax tree and a symbol table.

After the program has been parsed, the program must be checked for illegal information flow.

6.6 Verifier

As mentioned checking can be done both using typing rules and static checking of the abstract syntax tree. In general typing rules is the better choice, but due to our simple language, the checking can be done relatively simple by statically checking the AST.

In Listing 6.5 the verifier is shown. Not all functions used are defined explicitly, but their names should reveal their purpose.

- **analyze** – Checks the AST for illegal flow. It applies the `checkNode` function to the root node. If illegal flow exists a DLMException is thrown.
- **checkNode** – Checks current node, and all its subnodes for illegal flow. Every assignment is checked. Conditional statements (if and while) will result in the block label being updated for all subnodes.
- **exprLabel** – Overloaded function, which finds the label for the expression.

If no exception occurs when processing the AST, the program is secure from a information flow perspective.

Compared to implementing a type system for the language, this is a quicker solution, and makes the same guarantees as the typing rules would. However, if the system is used for a more complex language, for instance containing methods and objects, implementing a type system is recommended.

```
analyze : AST × SymbolTable → void
  throws DLMException
analyze(AST ast, SymbolTable st) =
  checkNode(root(ast),⊥,st)


checkNode : Node × SecurityLabel → void
  throws DLMException
checkNode(Node n, SecurityLabel bl, SymbolTable st) =
  case type(n) of
    STMT:
      forall n' ∈ subnodes(n) do checkNode(n',bl,st)
    ASSIGN:
      let l1 = assignLabel(n),
          l2 = lub(exprLabel(getExpr(n),st),bl) in
        if not lessRestrictive(l2,l1) then
          throw DLMException
    IF ∨ WHILE:
      let l = exprLabel(getExpr(n),st) in
        forall n'∈ subnodes(n) do checkNode(n',lub(l,bl),st)


exprLabel : Node × SymbolTable → SecurityLabel
exprLabel(Node n, SymbolTable st) =
  case type(n) of
    OP:
      let subnodes = subnodes(n) in
        lub(subnodes[1],subnodes[2])
    DECLASSIFY:
      let l = exprLabel(getExpr(n),st) in
        setConfLabel(l,declassifyLabel(n))
    ENDORSE:
      let l = exprLabel(getExpr(n),st) in
        setIntegrLabel(l,endorseLabel(n))


exprLabel : Leaf × SymbolTable → SecurityLabel
exprLabel(Leaf l, SymbolTable st) =
  case type(l) of
    VAL: ⊥
    VAR: getLabel(st,getVarName(l))
```

Listing 6.5: Verifier

```
split : AST × SymbolTable × TrustModel × OptimalSplit ×
        Principal-set → SplitAST × SplitSymbolTable

SplitAST : SplitNode
SplitNode : NodeId × Type × SplitNode* × Leaf* × Principal

SplitSymbolTable : Var -m-> Type × SecurityLabel × DUChain ×
                             Principal

/* Auxiliary functions used by the split function */
assignField : Var × SymbolTable × TrustModel →
              Principal-set

assignStatement : Node × SymbolTable × TrustModel →
                  Principal-set
```

Listing 6.6: Splitter

6.7 Splitter

The splitter partitions verified sflow programs. Each field and statement will be assigned to a principal, based on the trust graph. The splitter also needs to know which principals are currently active in the system, as sub-programs can only be scheduled on active principals.

Additionally, the splitter has an optimization component, which is used to find the optimal split, when more than one solution exists. This will be dealt with later.

Both the trust model and the optimization component are supplied as parameters. This means the splitter can be set up with any trust model or optimization component, just as long as they have the correct interface. This will be dealt with further as the individual components are introduced in the coming sections.

The split function uses two auxiliary functions, one for assigning fields and one for asssigning statements.

- **assignField** – As described in section 3.1.1, the requirement for a field f to be assigned to a principal p is:

  $$C(L_f) \sqcup \mathrm{Loc}_f \sqsubseteq C_p \quad \text{and} \quad I_p \sqsubseteq I(L_f) \tag{6.1}$$

  Loc_f is the least upper bound of all block labels where f is read.

  $$\mathrm{Loc}_f = C(bl(u_1) \sqcup bl(u_2) \sqcup \cdots \sqcup bl(u_n))$$

  These constraints will result in a set of principals, to which the field can be assigned. This set is the result of the function. If the set is empty, the program can not be split.

- **assignStatement** – For a statement S to be assigned to a principal p, p must have at least the confidentiality of all values used in the statement. Additionally p must have the integrity of all values defined (cf. section 3.1.2.

  $$C(L_{in}) \sqsubseteq C_p \quad \text{and} \quad I_p \sqsubseteq I(L_{out}) \tag{6.2}$$

  Where,

  $$L_{in} = \bigsqcup_{v \in U(S)} L_v \quad \text{and} \quad L_{out} = \bigsqcap_{l \in D(S)} L_l$$

  $U(S)$ denotes all values used in S, and $D(S)$ denotes all definitions in S. The *assignStatement* function also return a set of principals, which the statement can be assigned to.

Using these two auxiliary functions, the splitter finds all the principals, which each field and statement can be assigned to. The optimal split is then found, using the optimization component.

The splitter only deals with how to perform the actual split. Keeping track of active principals and updating the trust graph is done by the system manager, and will be described later.

6.8 Trust Model

The trust model is a central component in the partitioning. It contains all trust declarations of the principals. Using these trust declarations trust between principals can be derived.

```
type : TrustModel → int

trustsConf   : TrustModel × Principal × Principal → bool
trustsIntegr : TrustModel × Principal × Principal → bool

updateTrustModel : TrustModel × TrustDecl → TrustModel

trustLabels : TrustModel → TrustLabel-set

TrustLabel : Principal × SecurityLabel

principals : TrustModel → Principal-set
```

Listing 6.7: Trust model interface

The basic framework is shown in Listing 6.7.

The trust model from [ZZNM01] can be implemented as a set of trust labels:

```
TrustModel : TrustLabel-set
```

The probabilistic trust model will be designed using a graph approach. Each edge in the graph has a probability, and the minimal trust paths are found by traversing the trust graph.

The design is shown in listing 6.8.

- **pathProb** – Calculate probability of a single path:

$$P(TP_{A,B} \subseteq TG) = \prod_{S \in TP_{A,B}} p(S)$$

- **confTrust** – Calculate the confidence principal A has in another principal B with regards to confidentiality. This is done by first finding all minimal confidentiality paths, then finding all the combinations, and finally the accumulated trust:

$$\begin{aligned} conf(Trust_{A,B}) &= \sum_{i=1}^{k} P(\nu_i \subseteq TG) \\ &- \sum_{1 \leq i_1 < i_2 \leq k} P((\nu_{i_1} \cup \nu_{i_2}) \subseteq TG) \\ &+ \sum_{1 \leq i_1 < i_2 < i_3 \leq k} P((\nu_{i1} \cup \nu_{i_2} \cup \nu_{i_3}) \subseteq TG) \\ &- \cdots \end{aligned}$$

- **integrTrust** – Like confTrust, except that it finds minimal paths with regards to integrity.
- **calculateConfidences** – Find all confidence values, for all combinations of principals. This results in two mappings, one for confidentiality and one for integrity. Using this and the thresholds, the trust labels can be derived. If the threshold is lower or equal to the confidence, the principal trusts the principal sufficiently.

Storing the calculated confidences has some obvious performance advantages. Otherwise the paths and probabilities would have to be found, each time trust between two principals has to be found.

The minimal trust paths can be found by taking all direct trust edges from the starting node. If a direct edge to the destination principal exists, it is added to the minimal trust paths.

Hereafter all the recommended edges are checked by investigating if a recommendation path exists from that node. Each time a recommendation edge is taken, the recommendation distance is decremented. The algorithm can be seen in Listing 6.9. The '@'-operator appends an element to the list.

When several recommendation edges exist between two principals, the minimal trust path will always be the one with the lowest recommendation distance where the recommendation path is still valid.

This concludes the description of the trust model framework, and the two trust models, which will be implemented.

```
TrustModel : TrustEdge-set × Thresholds

Thresholds : Principal $\xrightarrow{m}$ Probability

TrustEdge : ConfTrustEdge | IntegrTrustEdge
          | RecConfEdge    | RecIntegrEdge

Probability : real ∈ [0,1]

ConfTrustEdge   : Principal × Principal × Probability
IntegrTrustEdge : Principal × Principal × Probability

RecConfEdge     : Principal × Principal × Probability ×
                  Distance
RecIntegrfEdge  : Principal × Principal × Probability ×
                  Distance

/* Probability of a path */
pathProb : TrustEdge-set → Probability

/* Calculate confidence between two principals */
confTrust    : TrustModel × Principal × Principal →
               Probability
confTrust(TrustModel tm, Principal from, Principal to) =
  trust(minTrustPathsConf(tm,from,to))

integrTrust  : TrustModel × Principal × Principal →
               Probability
integrTrust(TrustModel tm, Principal from, Principal to) =
  trust(minTrustPathsIntegr(tm,from,to))

/* Calculate the confidence from the minimal paths */
trust : (TrustEdge-set)* → Probability

/* Minimal trust paths */
minTrustPathsConf   : TrustModel × Principal × Principal →
                      (TrustEdge-set)*
minTrustPathsIntegr : TrustModel × Principal × Principal →
                      (TrustEdge-set)*

/* Find all confidences */
calculateConfidences : TrustModel →
              ( (Principal × Principal) $\xrightarrow{m}$ Probability ) ×
              ( (Principal × Principal) $\xrightarrow{m}$ Probability )
```

Listing 6.8: Probabilistic Trust Model

```
minTrustPaths(TrustModel tm,
              Principal at,
              Principal dest,
              Principal-set visitedNodes,
              TrustEdge-set traversedEdges,
              int recdist) =
if (at∈visitedNode ∨ recdist = 0)
  ∅
else
  directTrust(tm,at,dest,traversedEdges) @
  recTrust(tm,at,dest,visitedNodes,traversedEdges,recdist)

directTrust(TrustModel tm,
            Principal at,
            Principal dest,
            TrustEdge-set traversedEdges) =
  if (directTrustExists(tm,at,dest))
    traversedEdges ∪ getDirectEdge(at,dest)

recTrust(TrustModel tm,
         Principal at,
         Principal dest,
         Principal-set visitedNodes,
         TrustEdge-set traversedEdges,
         int recdist) =
  forall recedge = recommendationEdges(tm,at)
    minTrustPath(tm,dest(recedge),dest,visitedNodes ∪ at,
      traversedEdges ∪ recedge,
      min(recdist - 1, recDist(recedge) )
```

Listing 6.9: Minimal trust paths

6.9 Optimal Split

In many cases several valid splits exist. This leaves room for optimizations. However, what to optimize might differ from one user to the next. Several scenarios exist:

- Minimal network traffic.
- Optimal performance by scheduling hard computations to strong principals.
- Spread sensitive data on multiple principals in order to limit consequences in case of a bad principal.

To support this, a framework for optimizers is developed. Based on the abstract syntax tree, symbol table, trust model and the possible assignments, it finds the optimal split.

```
findOptimalSplit : AST × SymbolTable × TrustModel ×
                   (Node $\xrightarrow{m}$ Principal-set) → SplitAST
```

Three optimizers will be developed. A very simple one, which takes the first Principal in the set of possible principals. Secondly, optimization of confidence in the probabilistic trust graph (Section 5.8.1). The third is based on the metric from Section 5.8.2.

6.9.1 Highest Confidence

Optimization of confidence is fairly simple. For each statement all possibilities are considered. The statement will be scheduled to the principal, in which all the owners of the data used in the statement has most confidence in.

As already discussed in Section 5.8.1, the statement is assigned to the principal where the lowest confidence is as high as possible.

$$optimal(A, O) = max_{\forall q \in A}(min_{\forall p \in O}(conf(Trust_{p,q}))$$

```
metricConf   : ProbabilisticTrustModel × Principal ×
Principal → real
metricIntegr : ProbabilisticTrustModel × Principal ×
Principal → real

leakConf   : ProbabilisticTrustModel × Principal ×
Principal → real
leakIntegr : ProbabilisticTrustModel × Principal ×
Principal → real
```

Listing 6.10: Optimizer using the Metric

where A is the principals the statement can be assigned to, and O is the set of owners of the statement.

6.9.2 The Metric

The metric is somewhat similar to the highest confidence. For each statement the metric is calculated from all owners to all the principals for which the statement can be assigned. A principal is considered an owner if data he owns is included in the statement.

For owners of defined variables, the integrity metric is calculated. The confidentiality metric is used for owners of used variables. If a principal owns both defined and used variables, both metrics will be calculated.

The node will be assigned to the principal, where the highest metric for any of the owners is the lowest.

$$optimal(A, O) = min_{\forall p \in A}(max_{\forall q \in O}(M(q, p)))$$

where A is the principals the statement can be assigned to, and O is the set of owners of the statement.

The functions that are used to find the metric are shown in Listing 6.10.

```
State : SplitAST × SplitSymbolTable × TrustModel ×
        OptimalSplit × Principal-set //active principals

principalJoin  : Principal × TrustDecl × State → State
principalLeave : Principal × State → State

loadProgram : File × State → State

loadTrustGraph : File × State → State

submitProgram : Program × State → State
Program : String

split : State → State

resplit : State → bool
```

Listing 6.11: System manager

6.10 System Manager

The system manager is the central component in Secure Dynamic Program Partitioning. Using this component, principals can join or leave the network. They can update their trust graph, and the trust model and optimization model can be selected.

Additionally, the system manager can load programs and principals can submit programs, which will then be partitioned. Listing 6.11 contains the functions that makes up the system manager.

The system manager is the front end for Secure Dynamic Program Partitioning. Its interface must support all aspects of the framework. It is designed to be extendable by any user interface; graphical, terminal-based, web-based, etc.

The design is based on the state of the system manager. The functions will then manipulate that state. Each function is described below:

- **principalJoin** – Principal is added to the list of active principals. The trust graph is updated with the trust declarations of the principal joining. The program is resplit, if a better split exists.

- **principalLeave** – The principal is removed from the list of active principals. If the principal is part of the current split, the program is resplit.
- **loadProgram** – Program is loaded from file and parsed. Program is not split straight away.
- **loadTrustGraph** – Trust graph loaded from file.
- **submitProgram** – A program is submitted, parsed, and split across the network.
- **split** – The program is split using the Splitter.
- **resplit** – Predicate to decide if a program should be resplit.

Deciding whether a program should be resplit is a rather complicated task. If a principal leaves the network and it is not in the current split, the program should of course not be resplit. If it is in the current split, the program must be repartitioned.

However, when a principal joins, several possibilities exist. If a better split exists according to the optimizer, the program should be resplit. If execution already started, it should not be resplit (cf. Section 5.1).

6.10.1 Trust Declarations

When a principal joins it has a trust declaration. A trust declaration is a list of principals it trusts. Trust declarations differ from one trust model to another. In the simple trust model, the principal lists the principals it trust with regards to confidentiality, and the ones it trust with regard to integrity.

In the probabilistic model, recommendations also exist, and each statement has to be annotated with a probability.

Each trust model must also define a syntax for loading trust graphs from files. For the simple trust model the following syntax is used:

```
[ [C|I] [a–zA–Z]+ [a–zA–Z]+ ]*

//Example
C Alice Bob
```

And the probabilistic model:

```
[ [C|I|RC|RI] [a–zA–Z]+ [a–zA–Z]+ [0|1].[0 −9]* ]*

//Example
RC Alice Bob 0.90
```

Loading trust graphs is only used for testing, as it has some obvious security flaws. Normally trust graphs should only be generated by assembling the individual trust graphs of the principals.

6.11 Erasure Policies

For the erasure policies to work, they must be built into the *middleware*, and thereby be part of the *trusted code base.* Additionally, the middleware must contain the two predicates:

- *active(p)* – Decides if principal p is active in the distributed system.
- *scheduledTo(var,p)* – Checks if the variable *var* is scheduled on p.

The predicates always produce the correct result. If a principal suddenly exits the network, the predicates automatically become *false.*

Each time the variable is read or written, the condition $c = \neg active(A) \vee \neg scheduledTo(n, A)$ has to be checked by the type system during run-time. If the condition becomes true, the variable becomes unavailable on the principal.

This concludes the description of the design. In the next chapter the implementation of the design will be presented.

CHAPTER 7

Implementation

Paranoia is the only sane approach. In this business, you would be crazy not to be paranoid.
– Unknown

In this chapter the implementation of the design is presented. The chapter will not cover details about all implementation aspects, only non-trivial extensions from our design will be dealt with. For more extensive documentation, refer to the source code and *JavaDoc* on the enclosed CD-ROM, and the description of the program library in Appendix B.

The design has been implemented in Java. Java is an object-oriented, imperative programming language. Java has a large collection of programming libraries (or API), which provides a lot of the functionality needed for our framework.

Compared to a functional implementation (like ML and Haskell), an imperative, object-oriented implementation makes it easier for others to benefit from this work. This is mainly due to its more wide-spread use.

Java has been chosen over other object-oriented languages because of its platform independence, its large API, and its wide-spread use. Additionally, JIF extends Java, so this allows for easier integration of the developed libraries at a later stage.

Going from a functional specification to an imperative implementation is fairly straightforward. Data types can be implemented as objects, and functions can be declared as static methods. Nevertheless to better utilize the object design in Java, the methods will most of the time not be static. Instead they will manipulate already instantiated objects.

Erasure Policies (see Section 5.9) have not been included in the prototype, due to the limited amount of time available. Implementing Erasure Policies is left as future work.

7.1 Collection Framework

As the design specification states in the last section, data types which represent sets and maps are needed. The collection framework in the `java.util`-package has support for this. The following classes are used:

TreeSet Tree set is an ordered set, where the elements are sorted according to a *comparator*. The comparator is used to decide where in the tree to put an added element.

HashSet The hash set is essentially a HashMap. Elements are identified using their hash value. The HashSet is faster than the TreeSet due to its simple nature. However, in the case where element are constantly being manipulated and compared (as the case is with labels), the hash value can no longer be used.

TreeMap Similarly to the TreeSet, it uses a comparator. The keys are sorted in a tree according to the comparator. The values can be any object.

HashMap For each hash value an object can be stored. Similarly to the HashSet, the HashMap is faster, but will only be used in simple cases.

Vector In many cases, vectors will be used instead of arrays. This is because they are dynamic (size changes when elements are added), and part of the Collection framework, so conversion is easy, for instance to a set.

The collection framework is fast and versatile, so it is an obvious choice when implementing the before mentioned data types.

Additionally, since of Java 2, SE 5.0 support *generics* are supported, where the classes can now be instantiated with a type. E.g.

```
Vector<Principal> principals = new Vector<Principal >();
```

This eliminates the need for *type casting*, which in earlier versions resulted in inelegant code and a large risk of type cast errors.

7.2 Parser

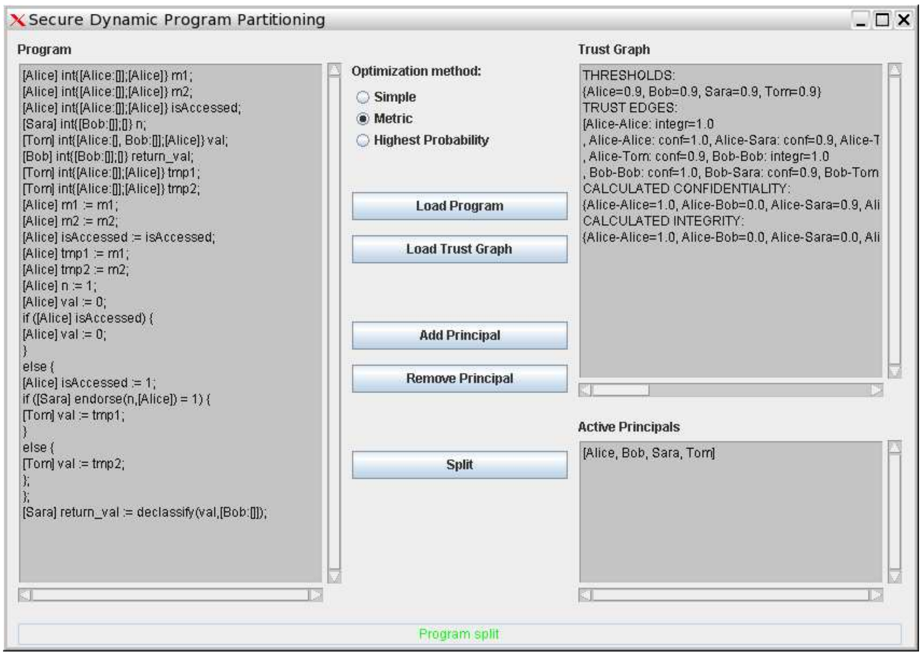
Parsing is done by using the JFlex 1.4.1 lexical analyzer to scan the file, and the BYACC/J 1.13 parser generator (Berkeley YACC 1.8 with Java support). The grammar from Section 6.5 is written, and using this grammar the abstract syntax tree is constructed. This is fairly straight-forward, and will not be dealt with further.

The two components, the lexer and the parser, are compiled into a Java class, which can then be called when parsing *sflow* programs.

7.3 User Interface

The system manager class provides an abstract interface, which can be used by any concrete interface: command prompt, web based, or traditional window GUI.

A simple graphical user interface (or GUI) was made as part of the implementation work in this thesis. The GUI can be seen in Figure 7.1. The GUI consists of:



Optimization method: Simple [✓ Metric] Highest Probability

```
[Alice] int{[Alice:[]];[Alice]} m1;
[Alice] int{[Alice:[]];[Alice]} m2;
[Alice] int{[Alice:[]];[Alice]} isAccessed;
[Sara] int{[Bob:[]];[]} n;
[Tom] int{[Alice:[], Bob:[]];[Alice]} val;
[Bob] int{[Bob:[]];[]} return_val;
[Tom] int{[Alice:[]];[Alice]} tmp1;
[Tom] int{[Alice:[]];[Alice]} tmp2;
[Alice] m1 := m1;
[Alice] m2 := m2;
[Alice] isAccessed := isAccessed;
[Alice] tmp1 := m1;
[Alice] tmp2 := m2;
[Alice] n := 1;
[Alice] val := 0;
if ([Alice] isAccessed) {
[Alice] val := 0;
}
else {
[Alice] isAccessed := 1;
if ([Sara] endorse(n,[Alice]) = 1) {
[Tom] val := tmp1;
}
else {
[Tom] val := tmp2;
};
};
[Sara] return_val := declassify(val,[Bob:[]]);
```

```
THRESHOLDS:
{Alice=0.9, Bob=0.9, Sara=0.9, Tom=0.9}
TRUST EDGES:
[Alice-Alice: integr=1.0
, Alice-Alice: conf=1.0, Alice-Sara: conf=0.9, Alice-T
, Alice-Tom: conf=0.9, Bob-Bob: integr=1.0
, Bob-Bob: conf=1.0, Bob-Sara: conf=0.9, Bob-Tom
CALCULATED CONFIDENTIALITY:
{Alice-Alice=1.0, Alice-Bob=0.0, Alice-Sara=0.9, Ali
CALCULATED INTEGRITY:
{Alice-Alice=1.0, Alice-Bob=0.0, Alice-Sara=0.0, Ali
```

```
[Alice, Bob, Sara, Tom]
```

```
Program split
```

Figure 7.1: Graphical user interface

- **Program text area** – Here the program is listed, and when split, principal ids are added before each statement (as seen in Figure 7.1).
- **Optimization method** – A radio group, where one of the three different optimization methods in this thesis can be selected.
- **Trust graph text area** – Here the trust relations are listed. In the case of the probabilistic model, the calculated confidences are also listed.
- **Active principals text area** – List of active principals.
- **Load program and trust graph** – Get program and trust graph from file.
- **Add principal** – Open dialog where the principal id and the trust declarations can be written.
- **Remove principal** – Open dialog where one of the active principals can be selected.
- **Split** – Split the program based on trust graph, optimizer and active principals.
- **Status line** – In the bottom of the window, where error and status messages will be displayed.

The interface is the *front-end* for the implemented prototype. It can be used to demonstrate the splitting process for various programs, trust models, and optimizations methods. Additionally, it allows users to update the network (add or remove principals) through a simple interface.

7.4 Generic Design

In the design specification, the splitter is parameterized with the trust model and optimization component. This is achieved by declaring interfaces, which the trust models and optimization components must use. The interfaces have the same form as presented in the Design chapter (Sections 6.8 and 6.9).

Chapter 8

Evaluation

Building technical systems involves a lot of hard work and specialized knowledge: languages and protocols, coding and debugging, testing and refactoring.
– Jesse J. Garrett

In this chapter the implemented system is evaluated. In the first part, the correctness of the implementation is tested. In the second part, the capabilities of the system are discussed, specifically performance and security. In the third part, the system is evaluated in two case studies. The cases are realistic scenarios which intend to show how *Secure Dynamic Program Partitioning* works in practice. The cases are:

- Insurance quotes
- Oblivious transfer

The focus will be on how the added support for dynamic networks provides new possibilities. Additionally, differences from the original *Secure Program Partitioning* will be discussed.

8.1 Test Strategy

The developed system will be thoroughly tested to ensure its quality. The test strategy has the following objectives.

- Ensuring the integrity of the developed system, hereby, also ensuring the integrity of the results arrived at.
- To make any future work easier, the central libraries developed should be thoroughly tested.

If these objectives are met, the quality of the system will be considered sufficiently high. To achieve this quality level, testing should test individual components, as well as ensure the quality on a larger scale. The following test methods will ensure this:

- High-level functional test. The system will be tested on the examples from this thesis. Additionally, some special cases will be constructed to test specific parts of the program.
- Unit testing all non-trivial classes. This will make it easier to work on the code in future work. Unit testing will be discussed further in the coming section.

This test strategy implies that non-essential modules, like the parser and user interface, will not be tested as thoroughly as the core modules, like the splitter and trust graphs.

The quality of the program will be considered sufficiently high, if none of the test cases fail.

8.2 Unit Testing

During development, unit test classes were constructed to test the implemented functionality. Whenever a key class was developed, a unit test class was created to test the functionality. For more information on unit testing see Appendix C.

The unit test approach achieved several things:

- The code is tested as soon as it is created (preferably at least). Testing is easier when the development process is still fresh in memory. Additionally, the code will to be more complete, as you are forced to conceive test cases, and hereby you realize where problems might occur at an earlier stage.
- Unit testing is modular, which means each module is tested individually before being put into the larger context. This results in fewer bugs, as the functionality of each module has been verified.
- The unit test can be run each time a change is made. So when future development creates bugs in the old code, this can quickly be identified and corrected.
- Unit tests serve as documentation. Future developers can look at the unit tests, to see how the system works.

In this thesis unit testing has been carried out by using the JUnit framework [JUn06a, JUn06b]. For a list of test cases see Appendix C.

8.3 Functional Testing

Functional testing is carried out by testing the System Manager. Both the abstract interface and the concrete interface (graphical interface) is tested. As the modules have been tested individually, these tests show that the program works at the top level.

Some of the test cases are:

- Several examples have been tested if they are split correctly, including the examples in this thesis.
- How the system responds, when the trust model is incompatible with the optimization method.

- Try splitting a program containing illegal flow.
- Try splitting a program which contains undefined variables.
- Trust model with illegal probabilities, i.e. not in the range $[0, 1]$.
- Principal leaving and joining the network.

For a complete list of test cases see Appendix C.

As neither the unit tests nor the functional tests have shown any errors, we conclude that the objectives from the test strategy (cf. Section 8.1) have been achieved, and the system fulfills our quality requirements.

8.4 Performance

A matter that is not directly related to the correctness of the program, is its performance. As mentioned earlier performance was not the main concern of this design. This section, however, will briefly discuss performance as it certainly has relevance for any future implementation work.

The performance of the compilation/verification phase can be done in quadratic execution time, $O(n^2)$, where n is lines of code. This is because the definition-use chains need to be constructed.

Calculating the confidence in the probabilistic trust graph takes exponential time (see Section 5.7)., Because of this, any implementation of a distributed system with a large trust graph, have to consider this. In our implementation, all confidences are recalculated each time the trust graph is modified. Future implementations might consider:

- Reusing those already calculated confidences, which are not affected by the change in the graph.
- Sensitivity analysis of paths, so paths with only marginal influence could be left out.

The optimization of splits (e.g. using the metric) can be done fairly quickly. For a statement, the number of owners and the possible principals it can be scheduled to, tends to be low. If the confidences have already been calculated, the splitting process is not a time consuming process.

Therefore, performance optimizations of the framework should focus on calculating the confidences in the probabilistic trust graph. However, it should be pointed out, that the examples in this thesis are not affected by this, as they contain a maximum of 3-4 minimal paths.

Another issue worth considering is network traffic. The main focus of the thesis has been security. But for computation intensive problems, the splitter might take factors like latency and bandwidth into account when partitioning the program. The original Secure Program Partitioning framework employs optimization methods to keep network traffic low. Our framework has support for these optimizations, as it allows users to introduce new optimization parameters.

8.5 Security

Security is, of course, a central issue in the framework. In this thesis the main concerns were information flow in the system and the trustworthiness of the principals. This was enabled by a few assumptions about the infrastructure:

- **Secure channels** – Network communication between two principals cannot be intercepted by any third party.
- **Principal identification** – Each principal is uniquely identified, and it is not possible to impersonate another principal.
- **Central splitter** – Every principal has trust in the splitter, and the integrity of the splitter can never be compromised.

These assumptions, however, do not hold in the real world. Using encryption the actual data can be protected sufficiently, but the network traffic might be monitored. This would leak information about the program execution (cf. Section 2.5 on covert channels).

Certificates and an underlying *public key infrastructure* (PKI) would provide a way of identifying principals. This, however, introduces a whole series of new security issues. An obvious approach would be to combine the trust graph in our framework with certificates and a PKI.

If a central splitter was employed in a running implementation, this would need to have a high level of security. The consequences of a compromised splitter are far reaching, as the data of all principals are essentially compromised. Alternatively a decentralized splitter could be used as described in Section 5.10.

Introducing realistic network infrastructure, certificates, and decentralized splitting is orthogonal to this work, and is left as future work.

8.6 Case Study: Insurance Quotes

Security is a central issue when using the Internet. Often users are asked to submit sensitive data. The users, however, have no control over how the data is used. The proposed framework allows users to annotate their data with security policies, which make sure that the data is not mistreated. The users want to make sure that the host they submit their data to will obey their policies and not try to access their private data (e.g. using covert channels).

This example will show, how our improved trust model allows users to choose which online services to use, so their data remains as safe as possible.

Assume two web services, $S1$ and $S2$, which if provided some personal details, find the cheapest insurance quote from a number of insurance companies. However, customers using this service want their personal information to be safe, as this is highly sensitive data, for instance medical history, yearly income, or current insurance premium.

The two services are equally good, hence customers will choose the one they consider most secure. The two services use the same database, which they co-own.

The program that makes up the web service is shown in Listing 8.1. Principal

```
InsuranceDB {S1:S2; S2:S1; ?:S1,S2} db;
PersData {x:;?:x} data;

Quote {x:; S1:; S2:} quoteTemp;
Quote {x:;} quote;

quoteTemp := findCheapestQuote(db,data);
quote := declassify(quoteTemp,{x:});
```

Listing 8.1: Program which find the cheapest quote.

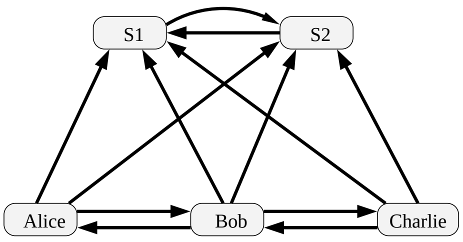


Figure 8.1: Simple trust graph for the insurance quote example

x can be any principal. For the web service to work, x must trust either $S1$ or $S2$. The *sflow* language does not contain any methods, so the purpose of the *findCheapestQuote* method is simply to illustrate the functionality of the program. The program will find the cheapest quote based on the data. The label of the resulting data will be the *least upper bound* of the data and the data base, as this, from an information flow perspective, is the same as:

```
    quoteTemp := db op data;
```

Using this small rewriting, the program can be split by the implemented system. We will now investigate how the program is split using the different trust models.

```
[S1]  InsuranceDB {S1:S2; S2:S1; ?:S1,S2} db;
[Bob] PersData {B:;?:B} data;

[S1]  Quote {B:; S1:; S2:} quoteTemp;
[Bob] Quote {B:;} quote;

[S1]  quoteTemp := findCheapestQuote(db,data);
[S1]  quote := declassify(quoteTemp,{B:});
```

Listing 8.2: Split program under the trust model in Figure 8.1.

8.6.1 The Simple Trust Model

In Figure 8.1 the initial trust graph is shown. It contains the two web servers providing the service, $S1$ and $S2$, and three potential customers, *Alice*, *Bob*, and *Charlie*. In the graph there is no distinction between confidentiality and integrity. If an edge exists, the principal trusts the other principal with regards to both confidentiality and integrity.

If *Bob* wants to use the insurance quote service, several possibilities exist for splitting the program. Data owned by only *Bob*, can be scheduled to any principal, while data owned by *S1* and *S2* must be scheduled to one of the two servers. A possible split can be seen in Listing 8.2.

8.6.2 The Probabilistic Model

The trust model is extended to support probabilities and recommendations. The adjusted trust graph is depicted in Figure 8.2.

The confidences can now be derived using the algorithm presented in Section 6.8. This has been done using the implemented system. The results are shown in Table 8.1.

As presented in Section 5.8.2 several possibilities exist when finding the optimal split. Two will be looked at here, the highest confidence and the metric optimization method.

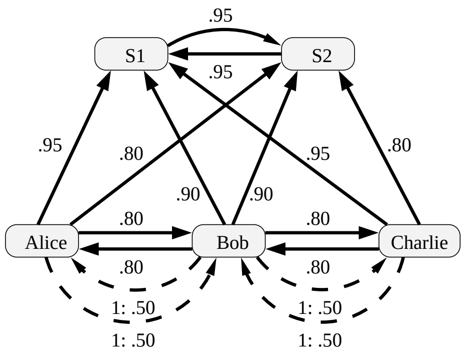


Figure 8.2: Probabilistic trust graph for the insurance quote example

Trust edge	**Confidence**
A-B	0.80
A-C	0.40
A-S1	0.97
A-S2	0.89
B-A	0.80
B-C	0.80
B-S1	0.97
B-S2	0.96
C-A	0.40
C-B	0.80
C-S1	0.97
C-S2	0.89

Table 8.1: Confidence table

```
[S1]   InsuranceDB {S1:S2; S2:S1; ?:S1,S2} db;
[Bob]  PersData {B:;?:B} data;

[S1]   Quote {B:; S1:; S2:} quoteTemp;
[Bob]  Quote {B:;} quote;

[S1]   quoteTemp := findCheapestQuote(db,data);
[S1]   quote := declassify(quoteTemp,{B:});
```

Listing 8.3: Split program using the *highest confidence* optimization method.

If the highest confidence method is used, *Bob* will prefer his data being scheduled on *S1*, and of course, if possible, at himself (cf. Table 8.1). The split is shown in Listing 8.3.

However, if we optimize with regard to the *metric*, we expect a different outcome. Now *Bob* considers the other principals *Alice* and *Charlie*, who may schedule their data on the server $S1$, but not on $S2$ (all trust thresholds are 0.90). *Bob* would prefer to schedule his data on a server, where none of the two principals *Alice* and *Charlie* will schedule their data, as he only trust them marginally.

Recall that the metric is defined as (cf. Section 5.8.2):

$$\begin{aligned} leak(A,B) &= \frac{\sum_{p\in T_B}(1 - conf(Trust_{A,p}))}{N} \\ M(A,B) &= \frac{leak(A,B)}{confidence(A,B)} \end{aligned}$$

Here, T_B is the set of principals who trusts B, and N is the cardinality of this set.

```
[S1]   InsuranceDB {S1:S2; S2:S1; ?:S1,S2} db;
[Bob]  PersData {B:;?:B} data;

[S2]   Quote {B:; S1:; S2:} quoteTemp;
[Bob]  Quote {B:;} quote;

[S2]   quoteTemp := findCheapestQuote(db,data);
[S2]   quote := declassify(quoteTemp,{B:});
```

Listing 8.4: Split program using the *metric* optimization method.

The metric for the two possibilities from *Bob*'s perspective are calculated as:

$$\begin{aligned} M(B,S1) &= \frac{(\mathit{conf}(\mathit{Trust}_{B,A}) + \mathit{conf}(\mathit{Trust}_{B,C}) + \mathit{conf}(\mathit{Trust}_{B,S2}))/3}{\mathit{conf}(\mathit{Trust}_{B,S1})} \\ &= \frac{(0.20+0.20+0.04)/3}{0.97} = 0.15 \\ M(B,S2) &= \frac{0.03}{0.96} = 0.03 \end{aligned}$$

From $M(B,S2) < M(B,S1)$ we get that *Bob* prefers his data to be scheduled on $S2$ (if not himself). Because the other owners $S1$ and $S2$, have no preference, the data should be scheduled on $S2$. The split is shown in Listing 8.4.

This case shows the significant difference between the two optimization methods.

- The highest confidence method, the service which the principal has the highest confidence in will be chosen.
- The metric, on the other hand, considers who else can schedule their data and statements at the service. The metric judges the likelihood that your data will be leaked to the other principals using the service by accumulating your distrust in the other principals and dividing this by your trust in the service. A principal would prefer to schedule its data to a service, which it fully trusts, as well as all other principal who can schedule their data to it.

It is hard to directly compare the two measures of splits, as their strength and weaknesses depend on the threat scenario. If it is realistic that other principals can gain access to your data by scheduling their programs to the service, then the metric is the better choice. However, if the other principals cannot interfere with the execution on the service, the service which you have the highest confidence in is preferable, because it is only the trustworthiness of the service that matters.

This small discussion illustrates the benefit of having the optimization component as a plugin. Threat scenarios are specific to the applications, thus the ability to customize the optimization parameters is a significant advantage. You could even imagine each principal in the system to have his or her own optimization requirements, and thereby be directly involved in how his or her data is scheduled.

8.7 Case Study: Oblivious Transfer

The second case considered is the *Oblivious Transfer* example [Rab81]. This example was also used in the original *Secure Program Partitioning* [ZZNM01]. It has been included to illustrate the added capabilities of *Secure Dynamic Program Partitioning*.

In the oblivious transfer example there are two principals, Alice and Bob, who do not trust each other. However, Alice will give up exactly one of two values to Bob, but will only do it once. Bob, on the other side, does not want Alice to know which of the two values he has requested. It is a well-established result that this can only be achieved if a trusted third party exist [DKS98].

Listing 8.5 contains the code for the Oblivious Transfer program. Alice has two integers *m1* and *m2*, where one of them will be transfered to Bob. The value *isAccessed* tells if Bob already received a value. *n* is either 1 or 2, depending on which value Bob is requesting. *val* is a temporary variable used to store the requested value, until it is declassified and returned to Bob (last line of the program).

It is important to notice that Alice needs to have trust in the integrity of the temporary value *val*, as she wants to make sure that the correct value is being declassified. This means that *n* has to be endorsed in the if-condition. This is

```
int{Alice:; ?:Alice} m1;
int{Alice:; ?:Alice} m2;
int{Alice:; ?:Alice} isAccessed;
int{Bob:} n;
int{Alice:; Bob:; ?:Alice} val;
int{Bob:} return_val;

if isAccessed then {
    val := 0;
}
else {
    isAccessed := 1;
    if endorse(n,{?:Alice}) = 1 then {
        val := m1;
    }
    else {
        val := m2;
    };
};
return_val := declassify(val,{Bob:});
```

Listing 8.5: Oblivious Transfer in *sflow*

experienced by:

$$L(endorse(n, \{? : Alice\})) \sqcup L(m1) = \{Alice : ;\ Bob : ;\ ? : Alice\}$$

8.7.1 Splitting under the Simple Trust Model

In the first part of the case study, the program is split using the simple trust model from [ZZNM01].

First we try to split the program when no third party exists. Alice and Bob only trust themselves:

$$\begin{aligned} L_{Alice} &= \{Alice : ;\ ? : Alice\} \\ L_{Bob} &= \{Bob : ;\ ? : Bob\} \end{aligned}$$

This program is not splittable under the constraints in equations (3.2) and (3.3). For instance, the field *val* cannot be assigned, as no label exists which is more restrictive than:

$$\{Alice : ;\ Bob :\}$$

So no valid split can be found by the Splitter.

However, suppose a principal Tom joins the network. Alice and Bob immediately declare their trust in Tom:

$$L_{Tom} = \{Alice : ;\ Bob : ;\ ? : Alice, Bob\}$$

The splitter will now try to split the program based on the new conditions, and this time a split exists. The split can be seen in Listing 8.6 (same resulting split as in [ZZNM01]). Alice is allowed to assign values to the common temporary value *val*, as she has as much integrity as the variable (variable has the integrity of Alice, and Alice trusts her own integrity).

However, endorsing n for Alice needs to be done by Tom, as she needs to have trust in the integrity of the principal downgrading the security.

Finally, Tom must store and declassify *val*, as this is the only way both Alice and Bob have trust in the declassification.

```
[Alice]  int{Alice:; ?:Alice} m1;
[Alice]  int{Alice:; ?:Alice} m2;
[Alice]  int{Alice:; ?:Alice} isAccessed;
[Bob]    int{Bob:} n;
[Tom]    int{Alice:; Bob:; ?:Alice} val;
[Bob]    int{Bob:} return_val;

[Alice]  if isAccessed then {
[Alice]      val := 0;
         }
         else {
[Alice]      isAccessed := 1;
[Tom]        if endorse(n,{?:Alice}) = 1 then {
[Alice]          val := m1;
             }
             else {
[Alice]          val := m2;
             };
         };
[Tom]    return_val := declassify(val,{Bob:});
```

Listing 8.6: Split of Oblivious Transfer program

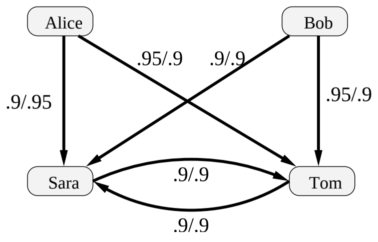


Figure 8.3: Trust graph for Oblivious Transfer. Trust edges are annotated with probabilities of the form *confidentiality/integrity*.

If Tom leaves the network, the split is no longer valid. As mentioned, our system does not support *erasure policies* yet, so the data stored on Tom's host will not automatically be deleted. This might result in a security breach, if Tom finds a way to circumvent the information flow policies.

The proposed *Secure Dynamic Program Partitioning* framework includes erasure policies. Applying erasure policies would, as mentioned in Section 5.9, address the issue of stored data on leaving principals. Due to time constraints this was, however, not included in the prototype.

8.7.2 Splitting under the Probabilistic Model

In this part of the case study, the probabilistic trust model is applied to the Oblivious Transfer example. The principal Sara has been added, which both Alice and Bob trust. Additionally, Sara and Tom trust each other. The probabilities can be seen in Figure 8.3.

We can now apply the two optimization methods Highest Confidence, and the Metric. As there are no recommendation paths, the confidences are the probability of the direct edge.

The values will here be calculated only for the statements which can not be assigned to either Alice or Bob. The others are trivial, and they will be scheduled as in Listing 8.6. The statements are:

```
int{Alice:; Bob:; ?:Alice} val
endorse(n,{?:Alice}) = 1
return_val := declassify(val,{Bob:})
```

Principals storing the field *val* need the confidentiality of Alice and Bob, and also the integrity of Alice. Both Sara and Tom fulfill this criteria.

The endorse statement need the confidentiality of Bob, as his value is being used. Additionally, the integrity of Alice is needed as her policy is being downgraded.

The declassify needs both the confidentiality of Alice and Bob. The integrity of Alice is needed because her policy is being downgraded, and since Bob's field *return_val* is being defined, his integrity is needed too.

Based on this, the metric and highest confidence can be calculated.

Applying the Highest Confidence optimization method

First, the confidence values are calculated. As only direct trust exists, this is very simple. For Sara the confidences are:

$$\begin{aligned} conf(TrustC_{A,S}) &= 0.90 \\ conf(TrustC_{B,S}) &= 0.90 \\ conf(TrustI_{A,S}) &= 0.95 \\ conf(TrustI_{B,S}) &= 0.90 \end{aligned}$$

Similarly, for Tom, the confidences become:

$$\begin{aligned} conf(TrustC_{A,T}) &= 0.95 \\ conf(TrustC_{B,T}) &= 0.93 \\ conf(TrustI_{A,T}) &= 0.90 \\ conf(TrustI_{B,T}) &= 0.90 \end{aligned}$$

If we first look at the field *val*. The field can be assigned to either Sara or Tom.

Statement	**Trust statements**	**Assigned to**
int{Alice:;Bob:;?:Alice} val	$TrustC_{A,x}, TrustC_{B,x}, TrustI_{A,x}$	Tom
endorse(n,?:Alice) = 1	$TrustC_{B,x}, TrustI_{A,x}$	Sara
return_val = **declassify**(val,Bob:)	$TrustC_{A,x}, TrustC_{B,x}, TrustI_{A,x}, TrustI_{B,x}$	Tom

Table 8.2: The table contains an overview of the critical statements in the Oblivious Transfer program. The table lists the statement, the trust statements, which influence assignment, and finally the principal, which it will be assigned too. (x refers to either Sara or Bob).

The trust statements which will determine, who it is assigned to are, for Sara

$$TrustC_{A,S}, TrustC_{B,S}, TrustI_{A,S}$$

and, for Tom

$$TrustC_{A,T}, TrustC_{B,T}, TrustI_{A,T}$$

As the lowest confidence is 0.90 in both cases, the highest *average* value will consider who will host the data. Thus, the field is assigned to Tom.

Table 8.2 lists all the statements, which need to be assigned to either Sara and Tom. The table also lists the trust statements, which will determine who will host the statement. Finally the principal which the statement is assigned to is listed.

The *endorse* statement is assigned to Sara, while Tom will host the *declassify* statement. In both cases it is determined by the principal with the highest average probability for the trust statements.

Applying the Metric Optimization Method

Recall, the definition of the Metric:

$$\begin{aligned} leak(A,B) &= \frac{\sum_{p \in T_B} (1 - conf(Trust_{A,p}))}{N} \\ M(A,B) &= \frac{leak(A,B)}{confidence(A,B)} \end{aligned}$$

```
[Alice]  int{Alice:; ?:Alice} m1;
[Alice]  int{Alice:; ?:Alice} m2;
[Alice]  int{Alice:; ?:Alice} isAccessed;
[Bob]    int{Bob:} n;
[Tom]    int{Alice:; Bob:; ?:Alice} val;
[Bob]    int{Bob:} return_val;

[Alice]  if isAccessed then {
[Alice]      val := 0;
         }
         else {
[Alice]      isAccessed := 1;
[Sara]       if endorse(n,{?:Alice}) = 1 then {
[Alice]          val := m1;
             }
             else {
[Alice]          val := m2;
             };
         };
[Tom]    return_val := declassify(val,{Bob:});
```

Listing 8.7: Split of Oblivious Transfer program using Highest Confidence optimization

We will calculate the metrics from Alice and Bob to Sara and Tom. First we calculate the metric from Alice to Sara. The principals Alice, Bob, and Tom might schedule their data and statements to Sara. So the metric for confidentiality from Alice to Sara becomes.

$$\begin{aligned} leak_C(A,S) &= \frac{(1 - conf(TrustC_{A,B})) + (1 - conf(TrustC_{A,T}))}{2} \\ M_C(A,S) &= \frac{leak(A,S)}{conf(TrustC_{A,S})} \\ &= \frac{((1-0)+(1-0.95))/2}{0.90} = 0.583 \end{aligned}$$

Here the C refers to confidentiality. I will be used to denote integrity.

All the values for the metric are listed below:

$$\begin{aligned} M_I(A,S) &= 0.579 \\ M_C(B,S) &= 0.594 \\ M_I(B,S) &= 0.611 \\ \\ M_C(A,T) &= 0.579 \\ M_I(A,T) &= 0.583 \\ M_C(B,T) &= 0.591 \\ M_I(B,T) &= 0.611 \end{aligned}$$

Based on these values the statements are assigned. The field *val* is, as before, stored at Tom, since the highest metric for Tom is 0.591, while for Sara the value is 0.594.

The endorse statement will be assigned to Tom, as this again gives the lowest metric. This is different from the Highest Probability method.

Finally, the declassify is also assigned to Tom. In this case the highest metric is the same in both cases, 0.611. But Tom will be used, because this will result in the lowest average:

$$\frac{0,583 + 0.579 + 0.594 + 0.611}{4} > \frac{0,579 + 0.583 + 0.591 + 0.611}{4}$$

The full split is listed in 8.8.

```
[Alice]  int{Alice:; ?:Alice} m1;
[Alice]  int{Alice:; ?:Alice} m2;
[Alice]  int{Alice:; ?:Alice} isAccessed;
[Bob]    int{Bob:} n;
[Tom]    int{Alice:; Bob:; ?:Alice} val;
[Bob]    int{Bob:} return_val;

[Alice]  if isAccessed then {
[Alice]      val := 0;
         }
         else {
[Alice]      isAccessed := 1;
[Tom]        if endorse(n,{?:Alice}) = 1 then {
[Alice]          val := m1;
             }
             else {
[Alice]          val := m2;
         };
         };
[Tom]    return_val := declassify(val,{Bob:});
```

Listing 8.8: Split of Oblivious Transfer program using the Metric optimization

In this example we have seen that it is possible for two principals to engage in a transaction, even when they do not trust each other. The security policies guarantee that no information, other than the strictly necessary, is leaked to the distrusted principal.

The case has been included to illustrate how information flow policies work in a distributed system. The example was also included in the original Secure Program Partitioning article [ZZNM01]..

CHAPTER 9

Future Work

The future belongs to those who see
possibilities before they become obvious.
– John Sculley

In this chapter, suggestions for future work are discussed. As in any project, the limited time available made it necessary to prioritize some areas over others. In the coming sections, useful and interesting extensions of the work in this thesis will be presented.

9.1 Execution Platform and Real Networking

The first extension that will be considered, is creating an execution platform where *sflow* programs can be executed. For distributed systems, the synchronization mechanisms presented in Section 3.3 should also be added when splitting the program. Finally, this should be tested when having several hosts connected by a network.

In this realistic setting, the security could be evaluated further. Requirements for the execution platform and network communication could be investigated. Eliminating covert channels would be an interesting research area.

9.2 Erasure Policies

Implementing *Erasure Policies* is an obvious extension of this work. Section 5.9 introduces Erasure Policies as part of our framework, while Section 6.11 presents a possible design, which would provide the functionality needed.

However, due to the limited time available, the implementation was not included. Especially if an execution platform was implemented, this would be interesting to investigate further.

9.3 Compatibility with JIF

Another useful extension would be to move from the simple *sflow* (cf. Chapter 4) to the fully-featured object-oriented *JIF* [Mye99]. JIF extends Java (except multiple threads) with information flow support. Moving to JIF would make it possible to test the framework on more realistic applications.

Moving to JIF would require that the splitter was updated to handle JIF code. The optimizer and trust model components are independent of the chosen language, so they could be used without any major modifications.

9.4 The Future of Secure Dynamic Program Partitioning

In this section we engage in a small discussion of where information flow has to go, to enter commercial use.

In recent years the *National Security Agency* has developed a version of the operating system *Linux* with *mandatory access control* [BL73] support. This is called *SELinux* (Security Enhance Linux). This shows that there is a rising trend to improve the inherently flawed execution platforms, which are currently in use. By the introduction of SELinux, the IT community has shown a will to rethink operating system design by introducing security primitives, in this case *Mandatory Access Control*. The next important step could be to make a version of Linux where the kernel supports information flow policies, like the *Decentralized Label Model*. Alternatively, a virtual machine, like the *Java Virtual Machine*, with information flow support could be constructed [vm06]. If such systems were developed, it would be a fundamental change to how we approach security.

Introducing secure information flow in distributed systems would be another milestone. This would finally achieve the objective of protecting people's sensitive data online.

It is clear that many obstacles still remain, but current security approaches suffer from more and more flaws, as complexity is increased [LSM$^+$98]. So maybe we need a fundamental change in the way we address security. Taking a low-level approach, like Secure Information Flow, could be the answer.

CHAPTER 10

Conclusion

At the end of the day, the goals are simple: safety and security.
– Jodi Rell

Applying *Secure Information Flow* to distributed systems has some promising perspectives. Information flow policies will allow users to better protect their data in distributed systems as these policies let users control access, integrity, and propagation.

The foundation of this work is *Secure Program Partitioning* [ZZNM01], which is a technique for distributing security-typed programs on the network, while obeying the information flow policies of the data, and the trust relation of the principals. Secure Program Partitioning, however, does not consider dynamic networks, and has a simple trust model that is not adequately suited for the complex trust relations of many distributed system.

Hansen and Probst addressed the first issue in [HP05]. This work has been build on further, by developing a full functioning framework for handling a dynamic network. The second issue, that is developing a suiting trust model, was the main focus of this thesis. By introducing recommended and partial (that is probabilistic) trust, we are better able to protect the users and express realistic trust scenarios in a dynamic network.

Another central issue in this thesis have been to resolve the ambiguity of splits. In most cases several splits exist for a given program and trust model. In the framework, the optimization component selects the split that will be used. Two optimization components have been presented in this thesis, both utilizing the probabilistic trust model.

- Assign data or statement to the principal in which the stakeholders have the highest confidence.
- The other method involves calculating a metric that captures the dependencies and nature of a data leak. The program parts are then assigned to the principals, which gives the optimal value for the metric.

These two methods purely consider the security of the split. Alternatively, a user might introduce his or her own optimization method, which would optimize performance.

The framework is parameterized with both the trust model and the optimization component. Compared to the original framework (Zdanewic et al.'s Secure Program Partitioning), this gives a higher level of flexibility. In our framework, applications can apply their own custom trust models and optimization criteria, and can thereby respond to the needs of the applications specific domain. In our view, this is a key factor if the Secure Program Partitioning approach are to be successful.

The proposed concepts have been proved to work through the implementation of a prototype. By using the prototype we are able to illustrate the capabilities of our framework.

In this thesis, a few examples have been investigated:

- Protecting credit card data, when shopping online.
- Protecting personal data, when using an insurance quote web-service.
- Securely transferring data in a scenario of mutual distrust.

These cases illustrate how the proposed framework, while supporting the original functionality, is able to handle dynamic networks. Additionally, the examples have shown how the framework is able to handle different trust models and optimization algorithms.

Appendix A

Definition of Terms

Confidence In probability theory, the conditional probability that a certain event, or series of events will happen.

Confidentiality Used in connection with trust. Having trust in a persons confidentiality, is having trust in his or her ability to protect your confident data.

Distributed System Decentralized system which uses multiple hosts connected by a network to perform computations. Distributed systems does in this definition not necessarily use parallel computation.

Erasure Policies Technique for automatically making data inaccessible, when certain conditions are fulfilled. See Section 5.9.

Integrity Used in connection with trust. Having trust in a persons integrity is having trust in his or her ability not to corrupt the data.

Metric Refers to a specific metric for evaluating program assignments. See Section 5.8.2.

Principal Entity in the trust graph. Includes persons, a group of users, and processes.

SDPP Secure Dynamic Program Partitioning.

sflow Simple, sequential language with information flow support. Will be used throughout this thesis. See Chapter 4.

SPP Secure Program Partitioning.

Trust Graph Data structure which contains all trust relations.

Appendix B

The sdpp Package

This appendix contains a description of the Java packages included in the framework. The appendix is intended to give a quick overview of the functionality of each class, and by using UML diagrams, we illutrate how each class fit into the context.

B.1 Basic Classes

Principal

Principal can be a user, process, etc. Anyone who uses data. Each principal has a unique PrincipalId. If new properties are needed for principals, extend this class.

PrincipalId

Unique identification of the principal. In this implementation the principal id is a string. Could also be integer, certificate, etc.

PrincipalSet

Set of principals. Extends BasicSet.

PrincipalComparator

Compare two principals. Implements the comparator interface, which allow it to be the ordering class of a TreeSet.

BasicSet

Basic set class. Super class of all sets in the sdpp package. Extends the TreeSet class. Tree sets are ordered sets, where each element is placed by applying the comparator interface.

B.2 Decentralized Label Model Package

An overview of the structure of the package is given in Figure B.1.

BasicLabel

Basic two component label consisting of a principal and a principal set: $(p, \{p1, p2, ..., pn\})$ This label type can be used for confidence label, trust relation, and declaring

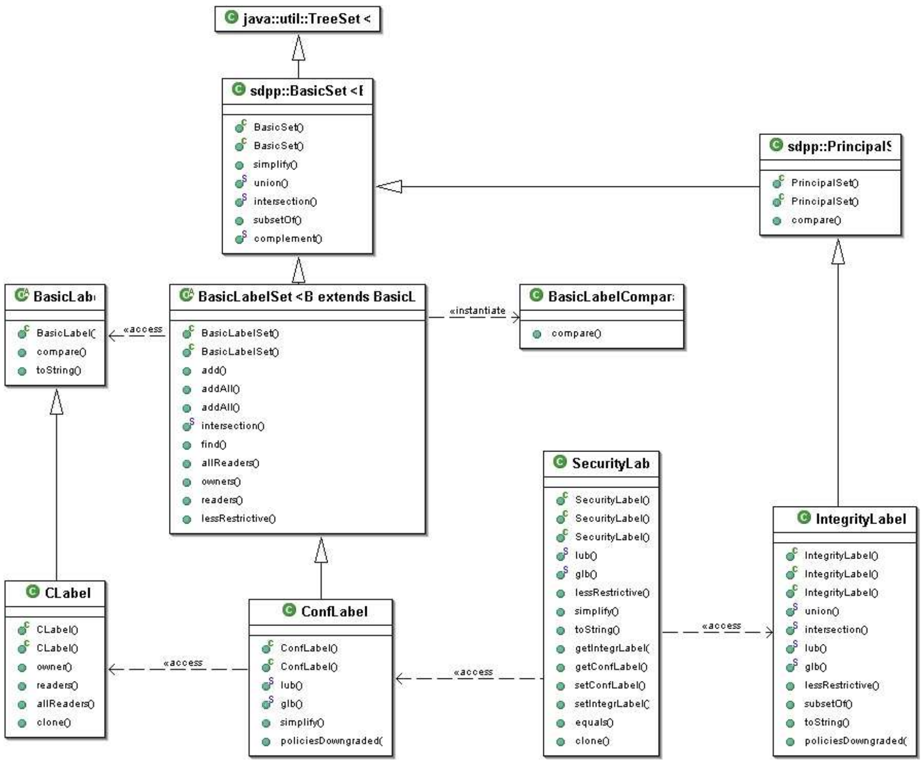


Figure B.1: UML diagram of the decentralized label model

ownership of hosts. Class can not be instantiated, declared abstract.

BasicLabelComparator

Class implementing the Comparator interface. Used to order BasicLabels in a BasicLabelSet.

BasicLabelSet

Set of BasicLabels. Extends BasicSet. It uses a generic type, however elements in set must extend BasicLabel. Elements are ordered using the BasicLabelComparator class.

Class cannot be instantiated; declared abstract.

CLabel

Confidence label component: $\{o : r1, ..., rn\}$, extends basic label.

ConfLabel

Confidence label. Is a set of CLabels:

$$\{o1 : r11, .., r1n; o2 : r21, ..., r2n; ...; on : rn1, ..., rnn\}$$

Extends BasicLabelSet.

IntegrityLabel

Integrity label: $\{? : r1, ..., rn\}$. Extends principal set. Integrity label contains the readers who trust the data. Everytime interference with other data occurs, the intersection of the two integrity labels are the resulting integrity. E.g. $\{? : r1, r2\} \sqcup \{? : r2, r3\} = \{? : r2\}$ Meaning only reader r2 trust the resulting data.

SecurityLabel

Security label contains both an confidence labels and an integrity label:

$$\{o1 : r11, ..., r1n; ...; om : rm1, ..., rml; ? : ri1, ..., rik\}$$

It is the the central class to the package, as it will be this object, which will be used to represents labels. It also contains methods for performing the operations:

- Least upper bound
- Greatest lower bound
- Less restrictive

B.3 Abstract syntax

This package contains classes for representing *sflow* programs. UML diagram for the package is depicted in Figure B.2.

AbstractSyntaxTree

Abstract syntax tree for sflow program. Contains reference to the top most node in the AST. Figure B.3 shows the abstract syntax tree for a while s A distinction are made between elements which has subnodes, Node, and those who has not, Leaf. Both extends the abstract class TreeElement.

Flow graph can also be retrieved using this class.

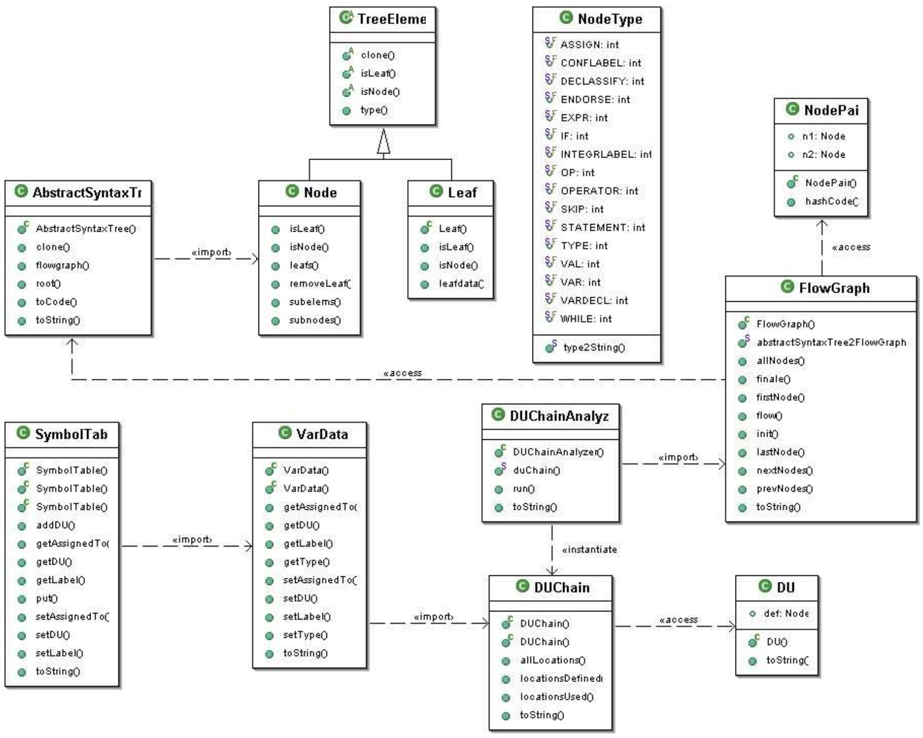


Figure B.2: UML diagram of the abstract syntax

```
while n do {
    n := n - 1;
}
```

```
   while
    /    \
  expr    S
  /       |
n      assign
        /    \
       n    expr
             |
            op
           / | \
         n  -  1
```

Figure B.3: Abstract syntax tree for a while statement

TreeElement

Super class for elements in the AST.

B.3.1 Node

Nodes are elements in the AST which can have both leafs and subnodes.

Node types:

STATEMENT Most basic node, has no leafs

ASSIGNMENT Expression (node) and variable (leaf)

EXPR Expression, can be a value, a var leafs.

OP 2 sub expr, and a leaf defining the operator (+,-,/,...).

DECLASSIFY expr node and label leaf.

ENDORSE expr node and label leaf.

Nodes can be assigned to principal, and contains block label. These are set by the splitter.

Leaf

Element in the AST. Leafs have no subnodes, but contain data. Types of leafs:

- Variable, e.g. 'x'
- Value, e.g. '3'
- Type, e.g. 'int'
- Confidentiality label, e.g. 'A:B,C; B:A'

- Integrity label, e.g. '?:A,B'
- Operator, e.g. '+'

NodeType

Node and leaf types in the AST.

NodePair

Two nodes, used in the flow graph.

SymbolTable

Contains information about the variables in the program.

Symbol table stores following data for each variable:

$$VarId \overset{m}{\to} Type \times Label \times DUChain \times AssignedTo$$

Type Is variable type, e.g. int.

Label SecurityLabel for variable

DUchain Definition-Use chain for variable

AssignedTo Used by the Splitter to assign variables to principals

The data is stored in VarData object for each variable.

VarData

Object which contains data for variables. Used by the SymbolTable

```
int{Alice:} n;
int{} a;

if a then
    n := n + 3;
else
    n := n + 2;

while n > 10 do
    n := n - 1;

a := 0;
```

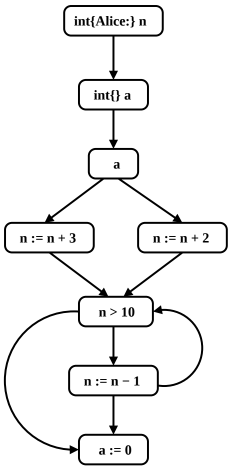


Figure B.4: Example of a flowgraph

FlowGraph

Construct directed graph of the program flow. An example of the flow graph can be seen in Figure B.4.

DU

Single definition and all its possible uses:

$$(l, \{l1, ..., ln\})$$

l is location (node) where variable is defined, and $l1, ..., ln$ are places where the variable might be used after this definition. Note a new definition will result in a new DU.

```
1 int{Alice:} n;
2 int{} a;
3 a := 3;
4 n := 2;
5 if a then
6       n := 3 + a;
7 else
8       a := 3 + n;
9 a := a + n;
```

$$\begin{array}{lcl}
\mathbf{a} & : & (\mathit{def}: 3,\ \mathit{uses}: 5, 6, 9), \\
 & & (\mathit{def}: 8,\ \mathit{uses}: 9), \\
 & & (\mathit{def}: 9,\ \mathit{uses}:) \\
 & & \\
\mathbf{n} & : & (\mathit{def}: 4,\ \mathit{uses}: 8, 9), \\
 & & (\mathit{def}: 6,\ \mathit{uses}: 9)
\end{array}$$

Figure B.5: Example of Definition-Use chains

DUChain

DUChains for a variable: { [L1: { L11,...,L1m }], ... , [Ln: { Ln1,...,Lno }] }
$L1, ..., Ln$ are locations where the variable is defined. E.g. `var := e;`

Example of DUChains for a program is shown in Figure B.5.

DUChainAnalyzer

Find all definiton-use chains for a variable, based on the flow graph for the program.

B.4 Parser

The parser is generated using JFlex 1.4.1 lexical analyzer to scan the file, and the BYACC/J 1.13 parser generator (Berkeley YACC 1.8 with Java support).

The grammar is show in Listing B.1.

Processing this grammar, will create a Java class Parser. The parser can parse files, strings, and java.io.Reader objects.

```
prog ::= vardecl c

vardecl ::= var ";" vardecl  |  var ";"

var ::= ID "{" seclabel "}" ID

c ::= s ";" c | s ";"

s ::= ID ":=" e
    | "if" e "then" "{" c "}" "else" "{" c "}"
    | "while" e "do" "{" c "}"

e ::= VAL | ID | e OP e
    | "declassify" "(" e "," "{ cl "}" ")"
    | "endorse" "(" e "," "{ il "}" ")"

seclabel: ε | cl il | cl | il

cl ::= ID ":" ";" | ID ":" ";" cl
     | ID ":" idlist ";" | ID ":" idlist ";" cl

il ::= "?" ":" idlist

idlist: ID | ID "," idlist
```

Listing B.1: BNF grammar for the *sflow* language

ParseException

Excpetion used when a parse error occurs.

B.5 Code verifier

The code verification is done by the sdpp.code.CodeVerifier class. It verifies parsed sflow code. The code must satisfy the property of non-interference until declassification. I.e. no high level data can flow to low level data.

B.6 Splitter

The splitter splits the program (consists of AST and SymbolTable) under the trust model. The trust model has to implement the TrustModel interface.

The split function uses two auxiliary methods, one for assigning fields, and one for asssigning statements.

- **assignField** –The requirement for a field f to be assigned to a principal p is:
$$C(L_f) \sqcup \mathrm{Loc}_f \sqsubseteq C_p \quad \text{and} \quad I_p \sqsubseteq I(L_f) \tag{B.1}$$
Loc_f is the least upper bound of all block labels where f is read.
$$\mathrm{Loc}_f = C(bl(u_1) \sqcup bl(u_2) \sqcup \cdots \sqcup bl(u_n))$$
- **assignStatement** – For a statement S to be assigned to a principal p, p must have at least the confidentiality of all values used in the statement. Additionally p must have the integrity of all values defined.
$$C(L_{in}) \sqsubseteq C_p \quad \text{and} \quad I_p \sqsubseteq I(L_{out}) \tag{B.2}$$
Where,
$$L_{in} = \bigsqcup_{v \in U(S)} L_v \quad \text{and} \quad L_{out} = \bigsqcap_{l \in D(S)} L_l$$

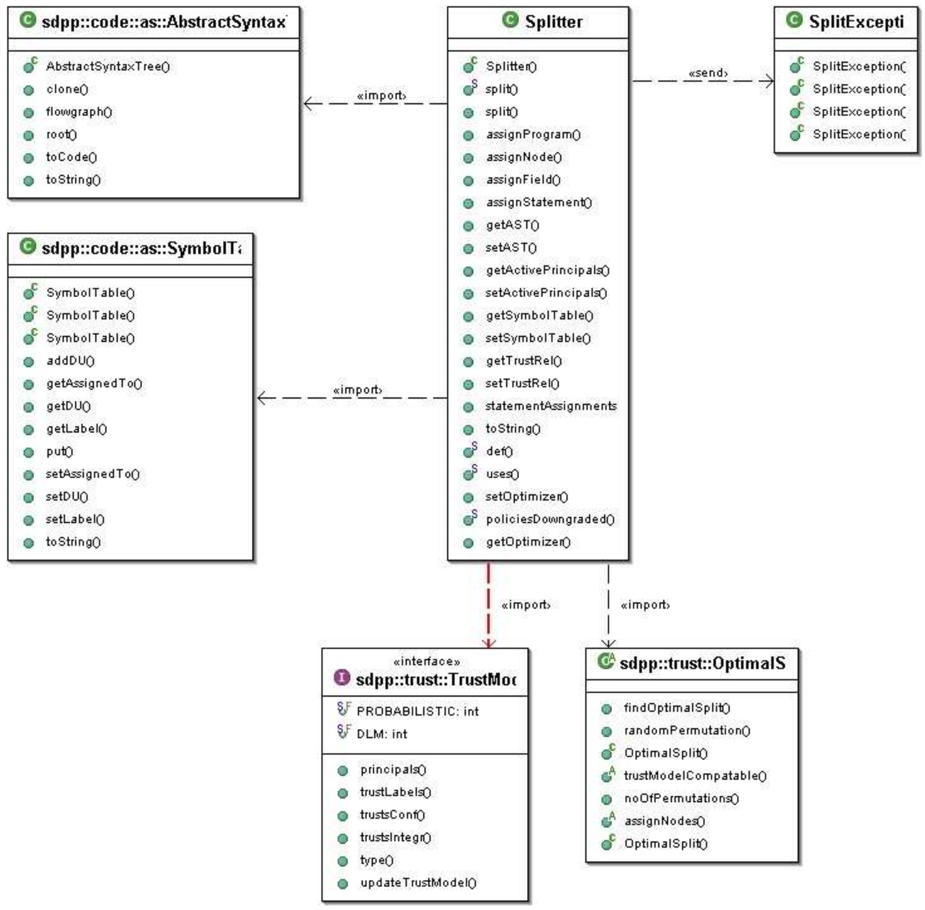


Figure B.6: UML diagram of the splitter

> $U(S)$ denotes all values used in S, and $D(S)$ denotes all definitions in S. The *assignStatement* function also return a set of principals, which the statement can be assigned to.

These constraints will result in a set of principals, which the field can be assigned to. This set is the result of the function. If the set is empty, the program can not be split.

When several possibilities exists for assigning a node to a host, the optimizer component is used to determined the optimal assignment. Optimizer must extend the abstract OptimalSplit class.

The splitter and the classes it depend on are depicted in Figure B.6.

SplitException

Exception which occurs when splitter cannot split the program.

B.7 Basic Trust Classes

Here follows a description of the basic classes which all trust model use.

TrustModel

Interface which all trust models must implement. Table B.1 lists the methods of the interface.

Here follows a description of each method in the interface.

principals All principals in the trust model.

trustLabels Convert trust derivations into trust labels.

Return type	Method header
PrincipalSet	principals()
TrustLabelSet	trustLabels()
boolean	trustsConf(Principal p1, Principal p2)
boolean	trustsIntegr(Principal p1, Principal p2)
int	type()
void	updateTrustModel(TrustDecl trustdecl)

Table B.1: Interface for TrustModel

trustConf Check if p1 trusts p2 with regards to confidentiality.

trustIntegr Check if p1 trusts p2 with regards to integrity.

type Type of trust model, e.g. DLM or PROBABILISTIC

updateTrustModel Update the trust model with trust declarations from a principal.

TrustDecl

Interface for trust declarations. Contains only two methods:

- **owner** – Principal who is declaring trust
- **type** – Type of trust declaration. Directly corresponds to the chosen trust model.

TrustLabel

Trust label is a security label with no readers. Additionally it has a owner which all the listed principals trust.

$$TL(O) = \{A :; B :; ? : A\}$$

A trust O with regards to confidentiality and integrity, while B trust O to maintain confidentiality.

TrustLabelSet

Set of trust labels. Extends BasicSet. Only one trust label can exist for each principal (owner). Uses the TrustLabelComparator to order trust labels.

BasicEdge

Basic edge in a trust graph. Contains an originating principal, destination and type.

ParseTrustGraph

Class for parsing trust graphs. Used for testing.

TrustException

Exception to be thrown when error is occured during trust computations.

B.8 DLMTrust

The DLM trust model is a simple trust model, which consists of a set of trust-labels. The UML diagram for the trust model is shown in Figure B.7.

DLMTrustModel

DLM trust model consists of a set of trust labels.

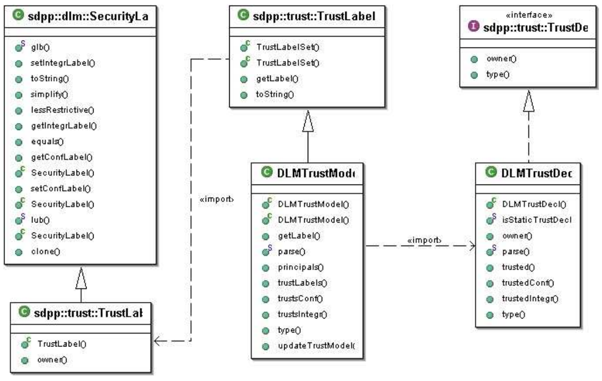


Figure B.7: UML diagram of the DLM trust model

If we have following trust labels,

$TL1(Alice) = \{Bob :; ? : Charlie\}TL2(Bob) = \{Charlie :\}TL3(Charlie) = \{\}$

A trust model could be

$$\{TL1, TL2, TL3\}$$

Meaning Bob trusts Alice with regards to confidentiality, and Charlie trusts Alice when it comes to integrity, and Bob with confidentiality. No one trusts Charlie.

The class contains methods for updating the trust model, as well as parsing a trust model.

DLMTrustDecl

Trust declarations for the DLM trust model.

Contains the principal which is declaring trust, and all the principals it trusts with regards to confidentiality and integrity.

B.9 ProbabilisticTrust

Figure B.8 has an overview of the probabilistic trust model.

TrustEdge

TrustEdge vector containing confidence trust, integrity trust and recommended trust. Used to specify trust from one principal to another (one-way). It has a probability which states how likely it is the trust declaration is correct.

The class is abstract.

ConfTrustEdge

Edge in the trust graph. States that a principal has trust in another principal's confidentiality, with probability p.

IntegrTrustEdge

Edge in the trust graph. States that a principal has trust in another principal's integrity, with probability p.

RecEdge

Recommendation edge. Abstract, used by ConfRecEdge og IntegrRecEdge.

ConfRecEdge

Confidentiality recommendation edge. Has probability and distance.

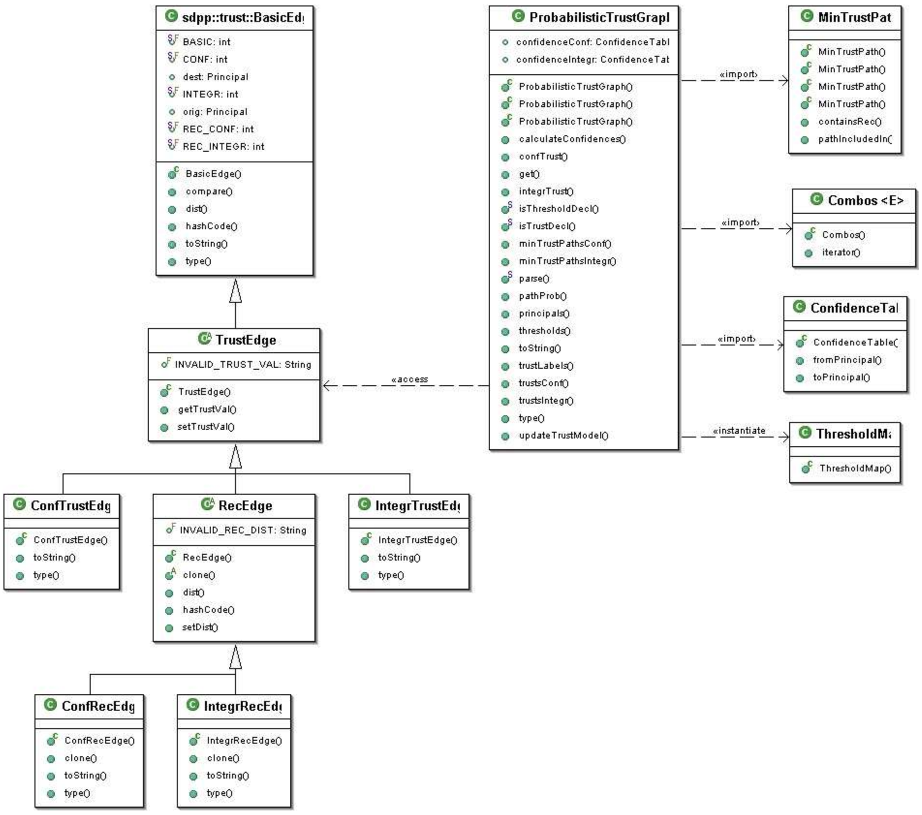


Figure B.8: UML diagram of the probabilistic trust model

IntegrRecEdge

Integrity recommendation edge. Has probability and distance.

ProbabilisticTrustGraph

The probabilistic trust graph is a set of trust edges. Using these trust edges confidence can be derived.

It first finds all the minimal path between the two principals, which trust will be derived for. This is done by the `minTrustPaths` method, which will find minimal paths from the current principal to the destination principal. If it encounters already visited nodes it will terminate, or if no nodes from the principal exists.

By storing the traversed edges, the full set of minimal paths can be constructed in the end.

Using these minimal trust paths, the confidence is calculated using the following formula:

$$\begin{aligned}
\mathit{conf}(\mathit{Trust}_{A,B}) &= \sum_{i=1}^{k} P(\nu_i \subseteq TG) \\
&- \sum_{1 \leq i_1 < i_2 \leq k} P((\nu_{i_1} \cup \nu_{i_2}) \subseteq TG) \\
&+ \sum_{1 \leq i_1 < i_2 < i_3 \leq k} P((\nu_{i1} \cup \nu_{i_2} \cup \nu_{i_3}) \subseteq TG) \\
&- \cdots
\end{aligned}$$

The calculated confidence will be stored in a ConfidenceTable (one for integrity and one for confidentiality) for each principal set, to allow for quickly generating the trust labels which will be used by the splitter. Trust is inferred if the calculated confidence is higher than the threshold.

ConfidenceTable

Table containing confidences. It maps a trust edge to a probability E.g.

$$Alice - Bob \rightarrow 0.90$$

Meaning that Alice has 0.90 confidence in Bob. The table can be either a confidentiality or an integrity table.

ThresholdMap

Table of confidence thresholds. The required confidence a principal need to have to trust somebody. E.g

$$[Alice \rightarrow 0.8, Bob \rightarrow 0.9, ...]$$

ProbabilisticTrustDecl

Probabilistic trust declaration. Contains the principal who owns them, its threshold and a collection of trust edges which originate from the owner. Also contains a method for parsing trust declarations from strings.

MinTrustPath

Minimal trust path between to principals.

Combos

Returns all possible combinations of n elements of an ArrayList. Original developed by David Ripton: http://sourceforge.net/projects/colossus Has been modified to support parameters.

TrustException

Exception to be thrown when error is occured during trust computations.

B.10 Optimizers

Next, the optimizers will be presented.

OptimalSplit

Abstract class which all optimizers must extend. Optimizer must decide based on AST, symbol table, trust model, and the possible assignments found by the spliiter.

SimpleSplit

A simple optimizer. Takes the first of the possible principal (organized lexiographically), which the statement can be assigned to.

OptimalProbability

Optimize with regards to probability. Statement will be assigned to the principal, which the owners of the data can best agree on. This means the one were the lowest probability of any of the principals is the highest.

OptimalProbMetric

Each node is assigned, according to the metric. The owners of defined variables wants the integrity metric to be low (low is better!), cause they want to make sure that the variable is defined correctly.

The owners of used variables, wants the confidentiality metric to be low, as they want to make sure that their data is not leaked.

The principals are treated equally, no metric has higher priority than others. This means we look for the principal, where the highest metric (both integrity and confidentiality) is the lowest. E.g.

```
int{A:} a; int{B:} b; int{C:} c;
a := b + c;
```

Possible assignments: D,E

$$\begin{aligned} metricIntegr(A,D) = 1.2, metricIntegr(A,E) = 0.4 \\ metricConf(B,D) = 1.1, metricConf(B,E) = 0.6 \\ metricConf(C,D) = 1.3, metricConf(C,E) = 2.0 \end{aligned}$$

Even though the average is lower if assigned to E, D will be chosen, as $1.3 < 2.0$. So the max metric will be kept as low as possible.

MetricTable

Table for storing metrics, when calculating metrics in the metric optimization method.

B.11 System Manager

The system manager is the interface where user will interact with the splitter. It keeps track of the program, trust model, optimizer

SysManGUI

Graphical User Interface for the system manager.

Appendix C

Test Scheme

This appendix contains a description of the test scheme that has been used to verify the correctness of the implementation. The appendix consists of two parts. First, all the high level functional test is described, hereafter the unit test cases are listed.

C.1 Functional Test

The functional test scheme consists of a list of high level tests, which will be carried out from both the graphical user interface, as well as from the abstract user interface.

The test will try out many possible errors, and ensure the program responds correctly. All examples from this thesis is tested as well, on both trust models, and the three optimizers.

The following test cases has been identified:

1. Loading program with bad syntax.

2. Loading program with illegal information flow.
3. Loading program with undefined variables.
4. Loading legal program.
5. Loading bad trust graph.
6. Loading legal trust graph.
7. Splitting with no program.
8. Splitting with no trust model.
9. Splitting with no optimizer.
10. Splitting when trust model is incompatible with optimizer.
11. Splitting Oblivious Transfer example with all trust models and optimizer.
12. Splitting Insurance Quote example with all trust models and optimizer.
13. Splitting Online Store example with all trust models and optimizer.
14. Splitting Hospital example (see Listing C.3) with all trust models and optimizer.
15. Principal joining, resulting in a new split.
16. Principal joining, resulting in no change.
17. Principal leaving, resulting in a new split.
18. Principal leaving, resulting in no valid split exists
19. Principal leaving, resulting in the same split.

All tests produces the correct result, so this test scheme has shown no errors in the implemented program.

C.2 Unit Tests

During development, unit test classes were constructed to test the implemented functionality. Whenever a key class was developed, a unit test class was created to test the functionality. A test class consists of:

- **Setup** – The data needed will be constructed, e.g. parsing programs, constructing the trust graph, etc.
- **Test cases** – Each method is tested individually with a set of test cases.
- **Teardown** – The constructed data is deleted before the next method is tested. Hereby it is ensured that the data is always correct before each test case is run.

In Listing C.1 a simple example illustrates how unit testing is done in practice. The class *Example* contains two methods, *sum* and *product*, which need to be tested. In the *setup* method two *Example*-objects are created and used in order to test both methods.

Worth noting is the use of the static method *assertEquals* inherited from the *TestCase*-class. This method checks if the two integer values match. If not, an error is reported. The teardown method is unnecessary, as the objects are reinitialized each time. It is simply included to illustrate its purpose.

The unit test approach achieved several things:

- The code is tested as soon as it is created. Testing is easier when the development process is still fresh in memory. Additionally, the code will to be more complete, as you are forced to conceive test cases, and thereby you realize where problems might occur at an earlier stage.
- Unit testing is modular, which means each module is tested individually before being put into the larger context. This results in fewer bugs, as the functionality of each module has been verified.
- The unit test can be run each time a change is made. So when future development creates bugs in the old code, this can quickly be identified and corrected.

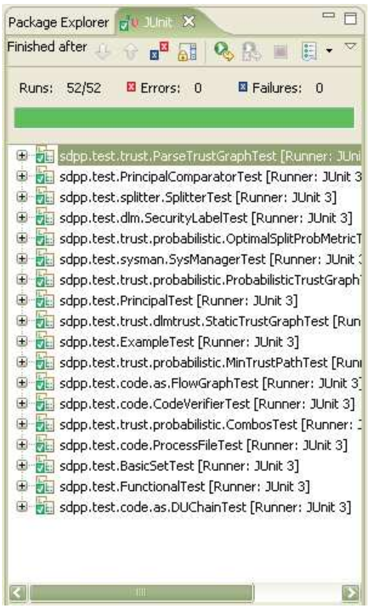
```
public class Example {
    private int[] numbers;

    public Example(int[] numbers) {
        this.numbers = numbers;
    }
    public int sum() {
        int sum = 0;
        for (int i = 0; i < numbers.length; i++)
            sum += numbers[i];
        return sum;
    }
    public int product() {
        int product = 1;
        for (int i = 0; i < numbers.length; i++)
            product *= numbers[i];
        return product;
    }
}

public class ExampleTest extends TestCase {
    Example ex1, ex2;

    protected void setUp() throws Exception {
        super.setUp();
        int[] numbers1 = {1,2,3};
        int[] numbers2 = {1,2,3,4};
        ex1 = new Example(numbers1);
        ex2 = new Example(numbers2);
    }
    protected void tearDown() throws Exception {
        super.tearDown();
        ex1 = null;
        ex2 = null;
    }
    public void testSum() {
        assertEquals("sum case 1",6, ex1.sum());
        assertEquals("sum case 2",10, ex2.sum());
    }
    public void testProduct() {
        assertEquals("product case 1",6, ex1.product());
        assertEquals("product case 2",24, ex2.product());
    }
}
```

Listing C.1: Example of unit testing



```
Runs:  52/52    Errors:  0    Failures:  0

sdpp.test.trust.ParseTrustGraphTest [Runner: JUni
sdpp.test.PrincipalComparatorTest [Runner: JUnit 3
sdpp.test.splitter.SplitterTest [Runner: JUnit 3]
sdpp.test.dlm.SecurityLabelTest [Runner: JUnit 3]
sdpp.test.trust.probabilistic.OptimalSplitProbMetricT
sdpp.test.sysman.SysManagerTest [Runner: JUnit
sdpp.test.trust.probabilistic.ProbabilisticTrustGraph
sdpp.test.PrincipalTest [Runner: JUnit 3]
sdpp.test.trust.dlmtrust.StaticTrustGraphTest [Run
sdpp.test.ExampleTest [Runner: JUnit 3]
sdpp.test.trust.probabilistic.MinTrustPathTest [Run
sdpp.test.code.as.FlowGraphTest [Runner: JUnit 3]
sdpp.test.code.CodeVerifierTest [Runner: JUnit 3]
sdpp.test.trust.probabilistic.CombosTest [Runner:
sdpp.test.code.ProcessFileTest [Runner: JUnit 3]
sdpp.test.BasicSetTest [Runner: JUnit 3]
sdpp.test.FunctionalTest [Runner: JUnit 3]
sdpp.test.code.as.DUChainTest [Runner: JUnit 3]
```

Figure C.1: Running JUnit test cases in Eclipse

- Unit tests serve as documentation. Future developers can look at the unit tests, to see how the system works.

Unit testing has been carried out by using the JUnit framework [JUn06a, JUn06b].

Figure C.1 shows the result of executing the JUnit test cases in Eclipse.

Below all the test cases are listed and described.

C.2.1 Basic classes

BasicSet

Class tested on integer lists. Test cases:

- union
- intersection
- subsetOf
- complement

The operations should leave the original sets unchanged, so after each test, the original sets are tested, to see they have not changed.

Principal

Two methods are tested on the principal class:

- compare
- equals

It uses the PrincipalComparator class.

PrincipalComparator

The comparator is tested to check if it orders principal correctly.

- compare

C.2.2 Decentralized Label Model

SecurityLabel

Tests the operations on security labels.

- glb
- lub
- lessRestrictive

C.2.3 Abstract Syntax

FlowGraph

The flowgraph is constructed from the abstract syntax tree. The method *abstractSyntaxTree2FlowGraph* is tested on three programs in listings C.2, C.3, and C.4.

In all three cases the correct flowgraph is generated, thus the class is considered correct.

DUChainAnalyzer

The DUChainAnalyzer class is tested on the program in Listing C.2

If the resulting DUChain matches the expected ones, the class is considered correct.

```
int{Alice:;} n;
int{Alice:;} t;
int{Alice:;} v;
;
n := 4;
t := 3;

if n then {
    t := 1;
    while n ≠ 0 do {
        t := t * n;
        n := n − 1;
    };
}
else {
    t := 0;
};

v := t;
```

Listing C.2: FlowTest program

```
//HOSPITAL − Calculate insurance

int{user:ins_company;} salary;
int{ins_company:user,hospital; user:ins_company;} insurance;
int{hospital:doctor,user; user:hospital;} med_hist;
;
if salary > 1000000 then {
    insurance := salary * 3/4;
}
else {
    insurance := salary * 1/2;
};

if med_hist ≠ 0 then {
    insurance := insurance * 10;
}
else {
    insurance := insurance * 2;
};
```

Listing C.3: Hospital program

```
int{Alice:; ?:Alice} m1;
int{Alice:; ?:Alice} m2;
int{Alice:; ?:Alice} isAccessed;
int{Bob:;} n;
int{Alice:; Bob:; ?:Alice} val;
int{Bob:;} return_val;
;

if isAccessed then {
    val := 0;
}
else {
    isAccessed := 1;
    if endorse(n,{?:Alice}) = 1 then {
        val := m1;
    }
    else {
        val := m2;
    };
};
return_val := declassify(val,{Bob:;});
```

Listing C.4: ObliviousTransfer program

```
int{Alice:;} a;
int{Bob:;} b;
;
b := a;
```

Listing C.5: DLMError program

CodeVerifier

The CodeVerifier is tested on the three programs in listings C.3, C.4 and C.5. The first and the last will result in an illegal flow exception (DLMException), while the oblivious transfer program contains no illegal flow.

C.2.4 Splitter

The splitter is tested by checking the individual methods in the class. The assignment methods are tested by checking different statement and field types, and seing they are assigned correctly.

Methods tested:

- assignField
- assignStatement
- def
- uses
- split

C.2.5 Trust

ParseTrustGraph

The class is tested by parsing trust graphs.

DLMTrustModel

The trust model is tested by adding a principal to the trust graph.

C.2.5.1 CombosTest

Test generating combos.

C.2.5.2 MinTrustPathTest

Test the pathIncludedIn method.

C.2.5.3 OptimalSplitProbMetric

Test the assignNodes method for the probabilistic model.

C.2.5.4 ProbabilisticTrustGraph

The following methods are tested:

- minTrustPathsConf
- minTrustPathsIntegr

- confTrust
- integrTrust
- trustsConf
- trustsIntegr
- trustLabels

Bibliography

[ARH97] Alfarez Abdul-Rahman and Stephen Hailes. Using recommendations for managing trust in distributed systems. In *IEEE Malaysia International Conference on Communication*, November 1997.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[BBK94] T. Beth, M. Borcherding, and B. Klein. Valuation of trust in open networks. In *Proc. 3rd European Symposium on Research in Computer Security – ESORICS '94*, pages 3–18, 1994.

[BIS06] `http://www.gnu.org/software/bison/`, October 2006.

[BL73] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.

[BM76] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications.* MacMillan, London and Basingstoke, 1976.

[BYA06] `http://dickey.his.com/byacc/byacc.html`, October 2006.

[CM05] Stephen Chong and Andrew C. Myers. Language-based information erasure. In *CSFW '05: Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 241–254, Washington, DC, USA, 2005. IEEE Computer Society.

[CUP06] `http://www.cs.princeton.edu/~appel/modern/java/CUP/`, October 2006.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. In *Communications of the ACM, Vol. 19, No. 5*, 1976.

[DKS98] Ivan B. Damgård, Joe Kilian, and Louis Salvail. On the (im)possibility of basing oblivious transfer and bit commitment on weakened security assumptions, 1998.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. *Proceedings of the Symposium on Security and Privacy*, pages 11–20, 1982.

[Hai84] Theodore Hailperin. Probability logic. *Notre Dame Journal of Formal Logic 25*, 1984.

[HP05] René Rydhof Hansen and Christian W. Probst. Secure dynamic program partitioning. In *Proceedings of Nordic Workshop on Secure IT-Systems (NordSec'05)*, Tartu, Estonia, October 2005.

[JS97] Jyh-Shing Roger Jang and Chuen-Tsai Sun. *Neuro-fuzzy and soft computing: a computational approach to learning and machine intelligence.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[JUn06a] `http://www.junit.org`, October 2006.

[JUn06b] `http://www.clarkware.com/articles/JUnitPrimer.html`, October 2006.

[Jøs96] A. Jøsang. The right type of trust for distributed systems. In *Proc. of the 1996 New Security Paradigms Workshop. ACM*, 1996.

[LSM$^+$98] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *Proceedings of the 21st National Information Systems Security Conference,*, pages 303–314, October 1998.

[Mau96] Ueli Maurer. Modelling a public-key infrastructure. In *ESORICS: European Symposium on Research in Computer Security.* LNCS, Springer-Verlag, 1996.

[ML97] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Symposium on Operating Systems Principles*, pages 129–142, 1997.

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

[MSZ04] A. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification, 2004.

[Mye99] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, pages 228–241, 1999.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[Ope] `http://www.openssl.org/news/secadv_20030219.txt`.

[Pob04] Eric Poblenz. Language-based approaches to secure information flow. Departement of Computer Science, University of California, Santa Cruz, 2004.

[Rab81] M. Rabin. How to exchange secrets by oblivious transfer. Aiken Computation Laboratory, Harvard U., 1981.

[RS06] A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Proceedings of IEEE CSFW'06*, 2006.

[SM03] A. Sabelfeld and A. Myers. Language-based information-flow security, 2003.

[SPJH06] Dan Søndergaard, Christian W. Probst, Christian Damsgaard Jensen, and Rene Rydhof Hansen. Program partitioning using dynamic trust. In *Proceeding of the Formal Aspect of Security and Trust workshop 2006*, 2006.

[SV98] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 355–364, New York, NY, 1998.

[vm06] *Moving Trust Out of Application Programs: A Software Architecture Based on Multi-Level Security Virtual Machines*, 2006.

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

[WSJ00] W. Winsborough, K. Seamons, and V. Jones. Automated trust negotiation. Technical Report TR-2000-05, Department of Computer Science, North Carolina State University, April 24 2000. Mon, 24 Apr 2000 17:07:47 GMT.

[WYS$^+$02] M. Winslett, T. Yu, K. E. Seamons, A. Hess, J. Jacobson, R. Jarvis, B. Smith, and L. Yu. The trustbuilder architecture for trust negotiation. *IEEE Internet Computing, volume 6, number 6*, pages pages 30–37, 2002.

[YKB93] R. Yahalom, B. Klein, and T. Beth. Trust relationships in secure systems–A distributed authentication perspective. In *RSP: IEEE Computer Society Symposium on Research in Security and Privacy*, 1993.

[ZCMZ03] L. Zheng, S. Chong, A. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems, 2003.

[ZH99] Lidong Zhou and Zygmunt J. Haas. Securing ad hoc networks. *IEEE Network*, 13(6):24–30, 1999.

[Zim95] Philip R. Zimmermann. *The official PGP user's guide.* MIT Press, Cambridge, MA, USA, 1995.

[ZZNM01] S. Zdancewic, L. Zheng, N. Nystrom, and A. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Symposium on Operating Systems Principles*, pages 1–14, 2001.