Università Ca'Foscari Venezia

Corso di Laurea magistrale
in Informatica - Computer Science

Tesi di Laurea

—

Ca' Foscari
Dorsoduro 3246
30123 Venezia

# A Quantitative Datacentric Approach to Differential Confidentiality Analysis

**Relatore**
Ch. Prof. Agostino Cortesi

**Laureando**
Gianluca Barbon
Matricola 818453

**Anno Accademico**
**2013 / 2014**

## Abstract

The spread of mobile devices that are increasingly connected to the net has lead to a rise of the importance of security in mobile systems. This study can be included in the area of analysis and verification of software involved in the data confidentiality, more specifically in the information flow analysis techniques capable of detecting data leakage in mobile applications. Current research can be divided into two different approaches, quantitative and qualitative. The primary aim of this work consists in combining the two approaches, in such a way to exploits the strengths of both. This yields to an abstract interpretation framework for explicit and implicit information flow in a data-centric approach, where each expression generated by a program keeps track of the explicit and implicit footprints of (possibly confidential) data in the local data-store, also in a quantitative way.

# Contents

# List of Figures

# Chapter 1

# Introduction

Security issues are increasing in mobile devices world, in particular in the Android environment. This is partially due to the lack of control over applications that instead is present in other environments, like in the Apple iOS and Windows Phone ones. In these systems, indeed, applications can only be purchased through a virtual marketplace, where apps are certified by the company that produces the operating systems. Moreover, a lot of applications and libraries, also popular ones, make use of user confidential data, without the user awareness. Mobile security is also becoming an important problem in the business context, where always more frequently firms allow the use of company applications in the personal devices of employees, increasing the risk of leakage of information considered confidential for the same company businesses. Thus, there's the need to verify the behaviour of such applications, and to limit the quantity of data that they release. A complete block of the leakage of data is infeasible, because it would compromise the functionalities of the device. Anyway, it is possible to limit the leakage thanks to policies able to establish a maximum level of data emission.

## 1.1 Purpose of this work

We want to find a method able to locate possible data leakage by mobile devices. In general, we want to establish some policies over the issued informations and understand if the installed applications satisfy these policies. Current research is divided into two main approaches: a statistical one and a language based one (taint analysis) [23, 24, 25, 2, 13, 21]. Both approaches have peculiar weaknesses: the former is weak because it does not fit well for qualitative analysis, while the latter is too strict, because it relies on the non-interference notion [10], that yields false positives which limit the effectiveness of the analysis. In particular, we want to investigate implicit flows, that are not often taken into accounts in current research, and relate them to the quantitative notion of information leakage.

## 1.2 Used methods

This work is an extension of the work of Cortesi et al. [6], from which we inherit the abstract interpretation approach. The advantage of such method is that it supplies a general abstraction of all the possible executions of a given program. Following the abstract interpretation framework [8], we design an enhanced concrete semantics that formalizes how the expressions generated by a program execution maintain footprints of (possibly confidential) data stored in the local data-store of the mobile device. Once the concrete semantics is formalized, we shos how to abstract it properly in order to make the analysis computable.

## 1.3 Results

This work leads to the definition of a framework that constitute a synthesis of the quantitative and the qualitative approach, by taking advantage of the respective strengths. We exploited the evaluation of single operators for the former approach and the collection of quantities of released information for the latter. Last, but certainly not least, the definition of this method has revealed the important role of the implicit flow in the leakage of secret variables. We evaluated the effectiveness of this framework over some benchmark examples.

## 1.4 Thesis structure

We present here the main structure of this thesis. After a brief introduction that describes related research and fundamental notions in Chapter 2, in Chapter 3 the semantics shown in the Cortesi et al. [6] work are introduced and extended in order to capture the implicit flow. Chapter 4, instead, proposes the new quantitative approach that is added to semantics described in the previous sections. Then, Chapter 5 introduces an abstraction of the quantitative analysis. Chapter 6 recalls the confidentiality and obfuscation notions proposed by Cortesi et al., extends them with the quantity concept, and defines two different kind of policies. Finally, in Chapter 7 some examples of real working applications are presented and analysed using our framework. Chapter 8 concludes.

# Chapter 2

# Background

This chapter introduces briefly some important notions that will be used throughout the thesis. Moreover it describes the current research in this (and related) field.

## 2.1 Implicit Flows

Implicit flows were described by Denning [11] in 1976, representing probably one of the first works where this kind of flows appeared. They have origins from the so called control statements, like `if` and `while` statements, where they are generated by their conditional expression. For instance, consider the following example:

```
if b then x = 0 else x = 1;
```

The branch is telling something about the content of the conditional expression. Indeed, even if expression b does not determine directly the value of x, it affect this value indirectly.

## 2.2 Quantitative Approaches

A quantitative approach is based on the examination of the quantity of leaked information. One of the best work in the quantitative information analysis is the one represented by McCarmant et Ernst [18]. Indeed, this paper proposes a new technique for determining the quantity of sensitive information that is revealed to public. The main idea presented by the author consists in computing a maximum flow between the program inputs and the outputs and after setting a sort of limit on the maximum quantity of information revealed. The information flow is not measured using tainting but with a sort of network flow capacity, where the maximum rate of an imaginary fluid into this network represents the maximum extent of revealed confidential information. Anyway such method requires a dynamic approach in order to construct the graph, by performing multiple runs of the examined program.

### 2.2.1 Quantitative value expressed as bits

Again, the work of McCamant et al. [16, 19] express a quantity concept designed to measure bits of information that can be released by the observation of a specific execution of a program. In such analysis they perform an over-estimation, that means they look at the upper-bounds of number of bits leaked.
One of the first attempts of quantifying information flow is the one of Lowe [15]. The author described quantity as number of bits, and defined the information flow as information passing between an high level user and a low level user through a *covert channel*. An interesting feature presented in this work consists in the assignment of *1 bit* also with absence of information flow. This means that the author considers the absence of information as having value 1 *bit*. Finally they also introduce a time notion in the flow analysis.
Another interesting approach is the one of Clark et al.[4, 3]. Indeed, it presents a lot of useful ideas about the use of quantities. First, they analyse *k bit* variables, where $2^k$ are the values that can be represented from such variables. Second, they relate the maximum content of a variable to its data type, and they consider this as the possible quantity of leakage. Finally, they define the difference between the quantity of information of a confidential input and the amount of leaked information.

## 2.3 Declassification

The declassification approach states that private variables can become public with the use of declassification methods, that must follow given policies. If an application does not respect a given policy, it is probably leaking confidential data. An interesting overview about current research on declassification has been done by Sabelfeld et al. [22]. In particular, they analyse this approach from four different perspectives, and they also present four semantic principles: consistency, conservativity, monotonicity of release and non-occlusion.

## 2.4 Confidentiality Analysis in Mobile Environments

The importance of confidentiality analysis is growing in the last years, especially in the mobile environments area. In this field two main approaches can be found, that's to say dynamic and static analysis methods. Among the works that use the former method we can find those regarding the evaluation of permission-hungry mobile applications [20, 2, 14]. In particular, the work of Enck et al. [12] presents a tool that monitors sensitive data by real time tracking, avoiding the needing to get access to the source code of applications. The main idea consists in tracking sensitive data that flows through systems interfaces, used by applications to get access to local data. Anyway this approach presents some lacks: for instance, it does not allow the tracking of control flows, and generates false positives.

# Chapter 3

# Collecting Semantics

In this section we introduce the collecting semantics, that consists in the first fundamental step of our framework design and will later be required to support the quantitative notion in the following chapter. We define here the domains, syntax and expressions. Finally, some particular problems that concerns control statements are debated.

## 3.1 Syntax

As described in Cortesi et al. [6] we consider only three types of data, strings ($s \in \mathbb{S}$), integers ($n \in \mathbb{Z}$) and boolean ($b \in \mathbb{B}$). Consequently, an expression for every type is defined: *sexp* for strings, *nexp* for integers and *bexp* for boolean. Moreover a label $l$ type is defined in order to represent data-store entries, with the corresponding expression *lexp*. String expressions are defined by: $sexp ::= s \mid sexp_1 \circ sexp_2 \mid encrypt(sexp, k) \mid sub(sexp, nexp_1, nexp_2) \mid hash(sexp) \mid read(lexp)$, where $\circ$ represents concatenation, *encrypt* the encryption of a string with a key $k$, *sub* the computation of the substring between two letter positions, *hash* the computation of the hash value and *read* the function that returns the value related to the data-store that corresponds to the given label.

## 3.2 Domain

Cortesi et al. also introduce a class of expressions called atomic data expressions, referenced as *adexp*, that are useful in order to keep track of the data sources used to obtain a specific value. The set of atomic data expressions is defined by: $\mathbb{D} = \{\langle \ell_i,\ L_i \rangle :\ i \in I \subseteq \mathbb{N}, \ell_i \in \mathtt{Lab}, L_i \subseteq \wp(\mathtt{Op} \times \mathtt{Lab})\}$, where $\mathtt{Lab}$ is the set of labels and $\mathtt{Op}$ is the set of operators. Specifically, an *adexp* is a set of elements $\langle \ell_i, \{(op_j, l'_j) : j \in J\} \rangle$. This formula states that the value of the associated expression is obtained by the combination of the datum that corresponds to label $\ell_i$ with data corresponding to labels $\ell'_j$ using corre-

sponding operators $op_j$.
They also define a data environment in order to refer variables to values coming from the data-store as $\Sigma = D \times V$, where $D : \mathbf{Var} \longrightarrow \wp(\mathbb{D})$ maps local variables in **Var** to a corresponding *adexp* and $V : \mathbf{Var} \longrightarrow (\mathbb{Z} \cup \mathbb{S})$ is the usual evaluation function that is used to track value information. They finally propose the $\star$ notation to specify data coming from the user input and from the constants of the program, in order to distinguish them from atomic data that instead come from the data-store. The latter are defined through the concept of *concrete data-store* $C$ represented by a set $\{\langle \ell_i,\ \emptyset \rangle :\ i \in I\} \subseteq \mathbb{D}$ *such that* $\forall i, j \in I : i \neq j \Rightarrow \ell_i \neq \ell_j$, *and* $\ell_i \neq \star$.

## 3.3 Extended Collecting Semantics

We formalize an extended version of the original *adexp* that is capable of collecting also implicit flow, generated by *if* and *while* statements. This flow is treated in the same way as the explicit flow, so we collect the boolean expression (*bexp*) of a conditional or loop statement and we consider it as an *adexp*, with its operators and sources.

**Definition 1 (Extended Atomic Data Expressions)** *We redefine the set of atomic data expression as:*

$$\mathbb{D} = \Big\langle \{\langle \ell_i, L_i \rangle : i \in I\}, \{\langle \ell_j, L_j \rangle : j \in J\} \Big\rangle$$

*where* $L \subseteq \wp(\mathtt{Op} \times \mathtt{Lab})$.

We can consider it as a pair of two *adexp*, where the second one refers to the implicit flows where also a boolean or relational operator may appear. So the *new adexp* will be structured as the following:

$$d = \langle d^e,\ d^i \rangle$$

Where $d^e$ and $d^i$ correspond to the explicit and implicit flows respectively. In this way we are considering also the boolean operators in the boolean expression as operators of an adexp. Let's consider an $if$ statement:

$$\textbf{if } \textit{expression} \textbf{ then } x = case_1 \textbf{ else } x = case_2$$

Following the new syntax, we can declare each expression as a combination of explicit and implicit flow:

$$\begin{array}{lcl} d_{if\ cond} & = & \langle d^e,\ d^i \rangle \\ d_{case_1} & = & \langle d^e_{case_1},\ d^i_{case_1} \rangle \\ d_{case_2} & = & \langle d^e_{case_2},\ d^i_{case_2} \rangle \end{array}$$

Notice that the $d^i$ in $case_1$ and $case_2$ express only the implicit flow generated in the scope of the two cases, but does not include the implicit flow that comes from the `if` statement. So if the $case_1$ is chosen, we will proceed in the following way:

$$d_{result} = \left\langle d^e_{case_1},\ \{d^i_{case_1} \cup d^e_{if} \cup d^i_{if}\}\right\rangle$$

The value associated to the variable $x$ after the `if-then-else` statement makes explicit the fact that $x$ has implicit dependence on the sources of the boolean expression.
Our semantic become a container of the Cortesi's et al. one. Like them, we suppose the standard concrete evaluation of numerical expressions ($S_N$ : $nexp \times V \rightarrow \mathbb{Z}$), string expressions ($S_S : sexp \times V \rightarrow \mathbb{S}$), boolean conditions ($S_B : bexp \times V \rightarrow \{\mathsf{true}, \mathsf{false}\}$) and label expressions ($S_L : lexp \times \Sigma \rightarrow \mathtt{Lab}$) that returns a data label given a label expression. We also reuse the semantics of expressions on atomic data $S_A : sexp \times \Sigma \rightarrow \wp(\mathbb{D})$, described in (Fig. 3.1). The semantics have been improved with a new operator, $checkpwd(sexp_1, sexmp_2)$, used to compare a secret password to the user input.
Moreover we must distinguish between explicit and implicit semantics. Basically, as for the semantics of expressions, we have no addition of new implicit flow, but only the memory of the implicit flow generated by previous expressions, so we have the union between $\emptyset$ (the new implicit flow) and $\{\langle \ell_j, L_j\rangle : j \in J\}$ (the implicit flow that comes from previous statements):

$$\left\langle S[\![c]\!](a, v),\ \emptyset \cup \{\langle \ell_j, L_j\rangle : j \in J\}\right\rangle$$

As for the concrete semantics of statements, we have to rewrite the ones that create implicit flow, so the *if* and the *while* statements (Fig. 3.2). We split the semantics into explicit ($S_e$) and implicit ($S_i$), in order to consider both flows in a statement or an expression. The semantics will not be different for the two flows, this division will only be used to treat the two flows separately. We also add the *skip* statement to handle the exit from a looping statement like the *while*.
In an $if$ statement we expect a *boolean condition*, that is an expression that has a boolean data type as result. In the condition we can find *boolean algebra* expressions, that by definition return a boolean value, but also *relational (comparison) operators*, that are defined in such a way to return a boolean value[1].
The major improvement we introduce is that we treat such expression not as a *bexp* but as an *adexp*. In this way we must consider also logical and relational operators as possible *op* of an *adexp*. In example, let's assume $a$ and $b$ as values that comes from the concrete data-store with labels $\ell_1$ and $\ell_2$, and used in the conditional expression of an $if$ statement with an "*equal to*" relational operator, with no previous implicit flow:

[1]For simplicity we do not consider bitwise operators.

$$
\begin{array}{rcl}
S_A[\![x]\!](a,v) & = & a(x) \\
S_A[\![read(lexp)]\!](a,v) & = & \{\langle S_L[\![lexp]\!](a,v),\emptyset\rangle\} \\
S_A[\![encrypt(sexp,k)]\!](a,v) & = & \{\langle \ell_1, L_1 \cup \{([encrypt,k],\ell_1)\}\rangle :\ \langle \ell_1, L_1\rangle \in S_A[\![sexp]\!](a,s,n)\} \\
S_A[\![s]\!](a,v) & = & \{\langle \star,\emptyset\rangle\} \\
S_A[\![sexp_1 \circ sexp_2]\!](a,v) & = & \{\langle \ell_1, L_1 \cup \{(\circ,\ell_2)\}\rangle, \langle \ell_2, L_2 \cup \{(\circ,\ell_1)\}\rangle : \\
 & & \langle \ell_1, L_1\rangle \in S_A[\![sexp_1]\!](a,v)\ , \langle \ell_2, L_2\rangle \in S_A[\![sexp_2]\!](a,v)\} \\
S_A[\![sub(sexp,k_1,k_2)]\!](a,v) & = & \{\langle \ell_1, L_1 \cup \{([sub,k_1,k_2],\ell_1)\}\rangle : \langle \ell_1, L_1\rangle \in S_A[\![sexp]\!](a,v)\} \\
S_A[\![hash(sexp)]\!](a,v) & = & \{\langle \ell_1, L_1 \cup (hash,\ell_1)\rangle :\ \langle \ell_1, L_1\rangle \in S[\![sexp]\!](a,v)\} \\
S_A[\![checkpwd(sexp,s)]\!](a,v) & = & \{\langle \ell_1, L_1 \cup \{(checkpwd,\star)\}\rangle :\ \langle \ell_1, L_1\rangle \in S_A[\![sexp_1]\!](a,v)\}
\end{array}
$$

Figure 3.1: Semantics of Expressions on Atomic Data

$$
\begin{array}{rcl}
S[\![x := sexp]\!](a,v) & = & (a[x \mapsto S_A[\![sexp]\!](a,v)], v[x \mapsto S_S[\![sexp]\!](v)]) \\
S[\![skip]\!](a,v) & = & (a,v) \\
S[\![send(sexp)]\!](a,v) & = & (a,v) \\
S[\![c_1; c_2]\!](a,v) & = & S[\![c_2]\!](S[\![c_1]\!](a,v))) \\
S[\![\text{if } c_1 \text{ then } c_2 \text{ else } c_3]\!](a,v) & = & \begin{cases} \langle S_e[\![c_2]\!](a,v),\ S_i[\![c_2]\!](a,v) \cup S_e[\![c_1]\!](a,v) \cup S_i[\![c_1]\!](a,v)\rangle \\ \qquad \textit{if } S_B[\![c_1]\!](v) \\ \langle S_e[\![c_3]\!](a,v),\ S_i[\![c_3]\!](a,v) \cup S_e[\![\neg c_1]\!](a,v) \cup S_i[\![c_1]\!](a,v)\rangle \\ \qquad \textit{otherwise} \end{cases} \\
S[\![\text{while } c_1 \text{ do } c_2]\!](a,v) & = & S[\![\ \text{if } (c_1) \text{ then } (c_2; \text{while } c_1 \text{ do } c_2) \text{ else } skip\ ]\!](a,v)
\end{array}
$$

Figure 3.2: Concrete Semantics of Statements

$$\textbf{if } (a == b) \textbf{ then } case_1 \textbf{ else } case_2$$

If we consider only the conditional expression we obtain:

$$d_{if} = \Big\langle \{\langle \ell_1, \{(==,\ell_2)\}\rangle, \langle \ell_2, \{(==,\ell_1)\}\rangle\},\ \{\emptyset\}\Big\rangle$$

### 3.3.1 Ambiguity problem in `if`

A problem of ambiguity may arise. With the relational operator it is important to maintain the order of elements, as $A < B$ is different from $B < A$. This is due to the fact that some relational operators are not commutative. Let's modify the previous example:

$$\textbf{if } (a < b) \textbf{ then } case_1 \textbf{ else } case_2$$

When we describe the expression in the form of *adexp*, we must collect it by looking at each source, so from the viewpoint of each label. But when we do this from $\ell_2$ viewpoint, if we alter the order of elements we also have to invert the operator direction. For $<, >, <=, >=$ this is very important in order to avoid the alteration of the semantic value of the expression, on the other hand for operators like $==$ and $!=$ this is useless. So, the last example will have the following atomic expressions:

$$d_{if} = \Big\langle \{\langle \ell_1, \{(<, \ell_2)\}\rangle, \langle \ell_2, \{(>, \ell_1)\}\rangle\},\ \{\emptyset\} \Big\rangle$$

### 3.3.2 Collecting `if` second case

The choice of one of the two cases may influence all the following code, both in the explicit and in the implicit flows. Thus, we must collect a precise information about the boolean condition in the implicit flow: indeed, if the second case is chosen, we know exactly that the boolean condition was false. To keep this information we decide to collect the operator that grant the opposite of the result of the boolean condition in the first case, that's to say the negated condition.
We proceed in the following way: in the second case of an if statement, we take into account the opposite of the relational operator (Fig. 3.3 and 3.4). Let's take the previous example and let's assume $a >= b$, so the condition is false. In this way the program will choose the second case. The resulting expression will be the following:

$$\begin{array}{lcl} \langle d^e_{case_2} & , & \{d^i_{case_2} \cup d^e_{if} \cup d^i_{if}\}\rangle \\ \Big\langle \{d^e_{case_2}\} & , & \Big\{d^i_{case_2} \cup \{\langle \ell_1, \{(>=, \ell_2)\}\rangle, \langle \ell_2, \{(<=, \ell_1)\}\rangle\} \cup \emptyset\Big\}\Big\rangle \end{array}$$

We thus collected the negated condition $!(a < b)$.

| | **original** | **negated** |
|---|---|---|
| *equal to* | == | ! = |
| *not equal to* | ! = | == |
| *greater than* | > | <= |
| *less than* | < | >= |
| *greater than or equal to* | >= | < |
| *less than or equal to* | <= | > |

Figure 3.3: Relational (comparison) operators

| | **original** | **negated** |
|---|---|---|
| *NOT* | $!A$ | $A$ |
| *AND* | $A$ && $B$ | $!(A$ && $B)$ |
| *OR* | $A \mid\mid B$ | $!(A \mid\mid B)$ |

Figure 3.4: Logical operators

*Example* Consider a more complex example. We assume that variable $y$ comes from the data-store and is labelled $\ell_1$, while $x$ comes from the user input and we assume that it is given a value $> y$, so the boolean condition of

$$
\begin{aligned}
S[\![sexp]\!](a,v) &= \{\langle \ell_1, L_1 \cup \{(assert, \ell_1)\}\rangle :\ \langle \ell_1, L_1\rangle \in S_A[\![sexp]\!](a,v)\} \\
S[\![\neg sexp]\!](a,v) &= \{\langle \ell_1, L_1 \cup \{(\neg, \ell_1)\}\rangle :\ \langle \ell_1, L_1\rangle \in S_A[\![sexp]\!](a,v)\} \\
S[\![sexp_1 < sexp_2]\!](a,v) &= \{\langle \ell_1, L_1 \cup \{(<, \ell_2)\}\rangle, \langle \ell_2, L_2 \cup \{(>, \ell_1)\}\rangle : \\
&\quad\ \langle \ell_1, L_1\rangle \in S_A[\![sexp_1]\!](a,v)\ , \langle \ell_2, L_2\rangle \in S_A[\![sexp_2]\!](a,v)\} \\
S[\![sexp_1 > sexp_2]\!](a,v) &= \{\langle \ell_1, L_1 \cup \{(>, \ell_2)\}\rangle, \langle \ell_2, L_2 \cup \{(<, \ell_1)\}\rangle : \\
&\quad\ \langle \ell_1, L_1\rangle \in S_A[\![sexp_1]\!](a,v)\ , \langle \ell_2, L_2\rangle \in S_A[\![sexp_2]\!](a,v)\}
\end{aligned}
$$

Figure 3.5: Concrete Semantics of Conditional Expressions

the *if* is `false`.

```
1 y = read(ℓ1);
2 x = userinput();
3 w = 9;
4 if (x <= y)
5     z = w;
6 else
7     z = y+3;
8 x = x + z;
```

The following are the corresponding expressions (we specify in the subscript the code line where the corresponding variable, assignments or other construct is present):

$$
\begin{array}{ccl}
\mathrm{y}_1 & : & \Big\langle \{\langle l_1,\ \emptyset\rangle\},\ \emptyset \Big\rangle \\
\mathrm{x}_2 & : & \Big\langle \{\langle \star,\ \emptyset\rangle\},\ \emptyset \Big\rangle \\
\mathrm{w}_3 & : & \Big\langle \{\langle l_2,\ \emptyset\rangle\},\ \emptyset \Big\rangle \\
(\mathrm{x}{<=}\mathrm{y})_4 & : & \Big\langle \{\langle \star, \{(<=, l_1)\}\rangle,\ \langle l_1, \{(>=, \star)\}\rangle\},\ \emptyset \Big\rangle \\
\mathrm{y}{+}3_7 & : & \Big\langle \{\langle l_1, \{(+, \star)\}\rangle,\ \langle \star, \{(+, l_1)\}\rangle\},\ \emptyset \Big\rangle \\
\mathrm{z}{=}\mathrm{y}{+}3_7 & : & \Big\langle \underbrace{\{\langle l_1, \{(+, \star)\}\rangle,\ \langle \star, \{(+, l_1)\}\rangle\},}_{\textbf{explicit flow}}\ \underbrace{\{\emptyset \cup \{\langle \star, \{(>, l_1)\}\rangle,\ \langle l_1, \{(<, \star)\}\rangle\} \cup \emptyset\}}_{\textbf{implicit flow}} \Big\rangle \\
\mathrm{x}{=}\mathrm{x}{+}\mathrm{z}_8 & : & \begin{array}{l} \Big\langle \{\langle l_1, \{(+, \star), (+, \star)\}\rangle,\ \langle \star, \{(+, l_1), (+, \star)\}\rangle,\ \langle \star, \{(+, \star), (+, l_1)\}\rangle\}, \\ \{\emptyset \cup \{\langle \star, \{(>, l_1)\}\rangle,\ \langle l_1, \{(<, \star)\}\rangle\}\}\Big\rangle \end{array}
\end{array}
$$

The first five expressions are not affected by implicit flows, so the $B$ set in these expressions is $\emptyset$.

# Chapter 4

# Quantitative Semantics

We now extend the concrete semantics described in the previous chapter with the quantity notion. Before the expressions extension, we define the concept of quantity of information applied to labels. Then we establish some restriction concerning conditional statements and, finally, we present the new collecting rules.

## 4.1 Definition of Quantity of Information

We will adopt binary values in order to quantify the information flows. This allows us to use a standard dimension value for quantities, that will be later compared with different labels related to different data types.

### 4.1.1 Quantity of information in a label

We want to retrieve the dimension in bits of a label. Thus, we create a simple function in order to retrieve the data size in bits of the allocated memory, by using a label as argument.

**Definition 2 (Label Dimension)** *Let $\ell_i$ be a label that denoting a location in the data-store. We introduce a function $\omega$ that returns the size of the memory location corresponding to the given label, such that:*

$$nbit_{\ell_i} := \omega(\ell_i)$$

*where nbit is the retrieved dimension in bits.*

The following are the possible values returned by the $\omega$ function when applied to a label:

- *numbers:* for simplicity we consider only integer numbers. The number of bits for a label containing such kind of data is: $nbits = \lfloor log_2(n) \rfloor + 1$

- *string:* instead of adopting the ASCII encoding, that uses 8 *bits* for each character, we consider a simpler encoding, designed represents only English alphabet, with uppercase and lowercase letters. Assuming this, we have 26+26 elements, thus this encoding needs only 6 *bits* for each character.
- *boolean:* since a boolean variable can only adopt two possible values, we produce only 1 *bit* as dimension of this data type

## 4.2 Quantitative Expression and Value Collection

We propose an extension of the semantics in order to take into account the quantity of information in the implicit flow, by adding a new expression associated to the extended *adexp*.

**Definition 3 (Quantitative Expression)** *We define as* qadexp *a sequence of couples of labels and associated quantitative value $\langle \ell,\ q \rangle$, that collect the quantity of information generated in the implicit flow for that specific label $\ell$. This sequence is combined with the extended adexp in a unique expression as following:*

$$d := (\ \langle d_e, d_i \rangle,\ d_q\ )$$

*In detail, we consider as expression representation the set:*

$$\mathbb{D} = \Big( \Big\langle \{\langle \ell_i, L_i \rangle : i \in I\}, \{\langle \ell_j, L_j \rangle : j \in J\} \Big\rangle,\ \{\langle \ell_k, q_k \rangle : k \in J\}\ \Big)$$

*where $\ell_k$ are the labels used in statements that generate implicit flow, while $q_k$ is the quantity of information associated. This value is incremented each time the label is involved in the implicit flow. If the label is never present in the implicit flow, no couple with the associated value appears.*

Thus every single couple $\langle \ell_k, q_k \rangle$ represents the quantity of information that the label $\ell_k$ has potentially released in the implicit flow. Notice that the expression $d := (\ \langle d_e, d_i \rangle,\ d_q\ )$ highlights how our analysis is the result of the combination of two approaches. While the first two components of the *adexp* expression comes from a *qualitative* approach, the latter is the result of a *quantitative* approach.

We define as $Q$ the domain of quantities of information. We can now introduce a function that describe how the quantitative value is collected.

**Definition 4 (Quantitative function $\phi$)** *Let $\phi$ be a function that update a quantitative value each time the associated label is involved in an implicit flow, such that:*

$$val^{\ell}_{post} := val^{\ell}_{pre} + \phi_{stm}(\ell)$$

*where $\phi_{stm} : \ell \mapsto val$ and pre and post refer to the statement stm execution. The quantity is mapped as an interval were higher and lower bound are the same value. This will allow an easier lift to the abstract value.*

*Example* For instance, let's consider the following sequence: $\{\langle \ell_1, 2\rangle, \langle \ell_2, 3\rangle, \langle \ell_3, 1\rangle\}$. Then, let's consider $\ell_1$ and $\ell_2$ involved in a code portion that generates an implicit flow that increments the quantitative value of one unit. Thus, the new sequence will be the following: $\{\langle \ell_1, 3\rangle, \langle \ell_2, 4\rangle, \langle \ell_3, 1\rangle\}$.

As already described in the previous chapter, we look for quantities of implicit flow generated by `if` and `while` statements.

### 4.2.1 Quantity of info in `if` statements

As for the `if` statement, we consider that the conditional expression has only two possible results: *true* and *false*. Thus, the obtained information consists only of one *bit* (as in McCamant et al. [17]), that corresponds to the two values that the boolean expression can take (notice that in both cases of the `if` we collect one bit). This bit represent the quantity of information that the statement reveals to an observer, allowing him to learn something of the labels in the conditional expression.

#### Restrictions on `if` statements

The quantification of information in the `if` statement is not so simple. Let's evaluate the following condition: ($a == b$). This is a *strong* condition, because it says that the code contained in the subsequent scope will be executed only if $a$ is *exactly equal* to $b$. If the condition is true, the maximum information can be obtained. Let's think about an external observer: in this case, he will know that the two values are the same, or worse, if he already knows $b$, he will also know the content of $a$. This example tells that we must consider that relational operators may introduce different "quantities" of data in the implicit flow. Indeed, a $<$ operator leaks less data than an $==$ operator, because the first one only tell us that a value is lower than the other, so it is not as strong as the first one. The problem is: how to decide a quantitative value? What is the quantity value we have to add for different type of operators? It is clear that it is difficult to assign a quantity value to the implicit flow in case of equality operator.
We may consider that in case the condition $a == b$ is true we should have obtained a quantity of implicit flow that almost reach the dimension in bits of the $a$ label (or $b$ label too). But if we do not know the value of $a$ and $b$, we only know that it is equal to $b$, and it appears difficult to recognize these different cases. For this reason, we have decided to adopt a reduced version of the `if` statement: in order to simplify the collection of quantities in case

of `if` statements we consider only > and < (strict) operators. This allow us to exclude the problem of equality ($a == b$) and allows the collection of only one bit of information for each `if` statement (notice that issues with equality have been described also by Clark et al. [3] [4]). Anyway, we rephrase the `if` condition when the label has a boolean value. Indeed we consider the value of a boolean label as an integer with values 1 and 0 for *True* and *False* respectively. See the two examples in figure 4.1. In this way the quantity of

$$
\begin{array}{l}
(a) \xrightarrow{\textit{is equivalent to}} (a == \mathit{True}) \xrightarrow{\textit{is equivalent to}} (a == 1) \xrightarrow{\textit{becomes}} (a > 0) \\
(!a) \xrightarrow{\textit{is equivalent to}} (a == \mathit{False}) \xrightarrow{\textit{is equivalent to}} (a == 0) \xrightarrow{\textit{becomes}} (a < 1)
\end{array}
$$

Figure 4.1: Conversion of boolean values in conditional expressions

information generated is also equal to the maximum quantity of information carried by a boolean value, thus avoiding the problem caused by equality operator. It would be possible to convert also the latter, by using an equality function that returns only boolean values, but the information produced in the information flow would be not measurable, because the peculiar issue persists. See figure 4.2 for the semantics of conditional expressions. We added the *assert* operator in order to collect conditional expression that contains only the label, thus when $label_a == true$.

$$
\begin{array}{rcl}
S[\![sexp]\!](a,v) & = & \{\langle \ell_1, L_1 \cup \{(>,\star)\}\rangle : \langle \ell_1, L_1\rangle \in S_A[\![sexp_1]\!](a,v)\} \\
S[\![\neg sexp]\!](a,v) & = & \{\langle \ell_1, L_1 \cup \{(<,\star)\}\rangle : \langle \ell_1, L_1\rangle \in S_A[\![sexp_1]\!](a,v)\} \\
S[\![sexp_1 > sexp_2]\!](a,v) & = & \{\langle \ell_1, L_1 \cup \{(>,\ell_2)\}\rangle, \langle \ell_2, L_2 \cup \{(<,\ell_1)\}\rangle : \\
 & & \langle \ell_1, L_1\rangle \in S_A[\![sexp_1]\!](a,v)\ , \langle \ell_2, L_2\rangle \in S_A[\![sexp_2]\!](a,v)\} \\
S[\![sexp_1 < sexp_2]\!](a,v) & = & \{\langle \ell_1, L_1 \cup \{(<,\ell_2)\}\rangle, \langle \ell_2, L_2 \cup \{(>,\ell_1)\}\rangle : \\
 & & \langle \ell_1, L_1\rangle \in S_A[\![sexp_1]\!](a,v)\ , \langle \ell_2, L_2\rangle \in S_A[\![sexp_2]\!](a,v)\}
\end{array}
$$

Figure 4.2: Concrete Semantics of Conditional Expressions

### 4.2.2 Quantity of info in `while` statements

In `while` statements we consider as quantity the number of loop iteration, to be converted into bits. Thus we compute the number of iterations by incrementing a counter and at the end we convert this number into a binary quantity value.

#### Restrictions on `while` statement

Similarly to the `if` statement conditional expression, we avoid the use of the equality operator. As for the `if` statement, we handle boolean values as numerical. Consider the following example:

```
found = False                                  found = 0
while (!found)                              →  while (found<1)
        pwd = user_input()                             pwd = user_input()
        found = checkpwd(pwd, secretpwd)               found = checkpwd(pwd, secretpwd)
```

We convert the variable *found* into an integer (code in the right column). Notice that in order to do this we have to consider functions that return boolean values as returning integer values between 0 and 1.
Moreover, we introduce a further restriction: we assume, in order to simplify the analysis, that in our semantic the label computed inside the while is the one in the left position in the condition, while the one in the right position is the label that can be inferred thanks to the implicit flow.

$$\texttt{while}\ (\underbrace{left\_position\_label}_{\textbf{computed inside while}} < \underbrace{right\_position\_label}_{\textbf{defined outside while}})\ \{\dots\}$$

## 4.3 Concrete Semantics Extended with Quantity

We define $\phi$ as a function that maps variable to quantitative expressions $\phi$ : $\texttt{Var} \rightarrow qadexp$. We now insert the quantity notion into our extended collecting semantics. Notice that the value of $\phi$ is modified only in the statements that generate implicit flow. This means that in the other expressions the $\phi$ component will be "carried" as is. The new semantics are described in figure 4.3, 4.4 and 4.5

$$
\begin{array}{rcl}
S_A[\![x]\!](a,\phi,v) & = & a(x) \\
S_A[\![read(lexp)]\!](a,\phi,v) & = & \{\langle S_L[\![lexp]\!](a,\phi,v),\emptyset\rangle\} \\
S_A[\![encrypt(sexp,k)]\!](a,\phi,v) & = & \{\langle \ell_1, L_1 \cup \{([encrypt,k],\ell_1)\}\rangle : \\
 & & \langle \ell_1, L_1\rangle \in S_A[\![sexp]\!](a,\phi,s,n)\} \\
S_A[\![s]\!](a,\phi,v) & = & \{\langle \star,\emptyset\rangle\} \\
S_A[\![sexp_1 \circ sexp_2]\!](a,\phi,v) & = & \{\langle \ell_1, L_1 \cup \{(\circ,\ell_2)\}\rangle, \langle \ell_2, L_2 \cup \{(\circ,\ell_1)\}\rangle : \\
 & & \langle \ell_1, L_1\rangle \in S_A[\![sexp_1]\!](a,\phi,v)\ , \\
 & & \langle \ell_2, L_2\rangle \in S_A[\![sexp_2]\!](a,\phi,v)\} \\
S_A[\![sub(sexp,k_1,k_2)]\!](a,\phi,v) & = & \{\langle \ell_1, L_1 \cup \{([sub,k_1,k_2],\ell_1)\}\rangle : \\
 & & \langle \ell_1, L_1\rangle \in S_A[\![sexp]\!](a,\phi,v)\} \\
S_A[\![hash(sexp)]\!](a,\phi,v) & = & \{\langle \ell_1, L_1 \cup (hash,\ell_1)\rangle : \\
 & & \langle \ell_1, L_1\rangle \in S[\![sexp]\!](a,\phi,v)\} \\
S_A[\![checkpwd(sexp,s)]\!](a,\phi,v) & = & \{\langle \ell_1, L_1 \cup \{(checkpwd,\star)\}\rangle : \\
 & & \langle \ell_1, L_1\rangle \in S_A[\![sexp_1]\!](a,\phi,v)\}
\end{array}
$$

Figure 4.3: Quantitative Semantics of Expressions on Atomic Data

$$
\begin{array}{rcl}
S[\![x := sexp]\!](a,\phi,v) & = & (a[x \mapsto S_A[\![sexp]\!](a,\phi,v)], v[x \mapsto S_S[\![sexp]\!](v)]) \\
S[\![skip]\!](a\phi,,v) & = & (a,\phi,v) \\
S[\![send(sexp)]\!](a,\phi,v) & = & (a,\phi,v) \\
S[\![c_1;c_2]\!](a,\phi,v) & = & S[\![c_2]\!](S[\![c_1]\!](a,\phi,v)))
\end{array}
$$

Figure 4.4: Quantitative Concrete Semantics of Statements

$S[\![\text{if } c_1 \text{ then } c_2 \text{ else } c_3]\!](a,v) =$

- *if* $S_B[\![c_1]\!](v)$ *is* `True` *then:*

  $$
  \begin{array}{rl}
  \textit{let } (a',\phi',v') = & \langle S_e[\![c_2]\!](a,\phi,v), \\
  & S_i[\![c_2]\!](a,\phi,v) \cup S_e[\![c_1]\!](a,\phi,v) \cup S_i[\![c_1]\!](a,\phi,v)\rangle
  \end{array}
  $$
  *in* $(a',\phi'[(\ell_i,k)/(\ell_i,h(k)) : \ell_i \in \texttt{src}(c_1)])$

- *otherwise:*

  $$
  \begin{array}{rl}
  \textit{let } (a',\phi',v') = & \langle S_e[\![c_3]\!](a,\phi,v), \\
  & S_i[\![c_3]\!](a,\phi,v) \cup S_e[\![\neg c_1]\!](a,\phi,v) \cup S_i[\![c_1]\!](a,\phi,v)\rangle
  \end{array}
  $$
  *in* $(a',\phi'[(\ell_i,k)/(\ell_i,h(k)) : \ell_i \in \texttt{src}(c_1)])$

$S[\![\text{while } c_1 \text{ do } c_2]\!](a,\phi,v) =$

- $S[\![\text{ if } (c_1) \text{ then } (c_2; \text{while } c_1 \text{ do } c_2) \text{ else } \textit{skip}\ ]\!](a,\phi,v)$

Figure 4.5: Quantitative Concrete Semantics of Control Statements

# Chapter 5

# Abstract Semantics

We now introduce an extended version of the abstract semantics proposed by Cortesi et al. [6]. The extension concerns the abstraction of the implicit flow and of the quantity of information notion, discussed in the previous sections. We adopt values and labels abstraction, that represent an over-approximation of the content variables used in the concrete. We use $V^a$ as value abstract domain. While the implicit flow behaves as the explicit one, adopting the same abstraction, we need to specify a new abstraction for the quantity.

**Definition 5 (Quantity Value Abstraction)** *The quantity associated to label expressions is an interval. Each obtained label $\ell^a$ is associated to an interval of quantities where the lower and upper bound are the minimum and the maximum quantities of information that can be released through the implicit flow for that specific label, respectively.*

Notice that we can have unbounded quantities, in case the analysis revealed a complete leakage of the associated label. In this event, the interval is only unbounded in the upper bound. As for the lower bound, the minimum quantity of information flow is zero.

## 5.1 Atomic Data Abstraction Extension

We now define the atomic data abstraction extended for handling implicit flow and associated quantity.

**Definition 6 (Abstract Extended Atomic Data and Abstract Quantities)** *Let's consider a set of atomic data and associated quantity values. The set of tuples*

$$\Big( \Big\langle \{\langle \ell^a_w, L^{a\sqcap}_w, L^{a\sqcup}_w \rangle : w \in W\}, \{\langle \ell^a_z, L^{a\sqcap}_z, L^{a\sqcup}_z \rangle : z \in Z\} \Big\rangle,\ \{\langle \ell^a_g, q^{a\sqcap}_g, q^{a\sqcup}_g \rangle : g \in Z\} \Big)$$

*that belong to $\mathbb{AD}$, is an extended abstract element where:*

- $\ell^a_w$ *is an element that abstracts labels in* Lab *and belongs to the explicit flow*
- $\ell^a_z$ *and* $\ell^a_g$ *are elements that abstract labels in* Lab *and belong to the implicit flow*
- $L^{a\sqcap}_w = \{(op^a_{iw}, \ell^a_{iw}) : i \in I\}$ *and* $L^{a\sqcap}_z = \{(op^a_{jz}, \ell^a_{jz}) : j \in J\}$ *represent the under-approximation of the set of couples operator-label* $\ell^a_w$ *and* $\ell^a_z$ *(respectively) with labels abstracted by* $\ell^a_{iw}$ *and* $\ell^a_{jz}$ *(respectively), and belong to the explicit and implicit flow respectively*
- $L^{a\sqcup}_w = \{(op^a_{iw}, \ell^a_{iw}) : i \in I'\}$ *and* $L^{a\sqcup}_z = \{(op^a_{jz}, \ell^a_{jz}) : j \in J'\}$ *represent the over-approximation of the set of couples operator-label* $\ell^a_w$ *and* $\ell^a_z$ *(respectively) with labels abstracted by* $\ell^a_{iw}$ *and* $\ell^a_{jz}$ *(respectively), and belong to the explicit and implicit flow respectively*
- $L^{a\sqcap}_w \subseteq L^{a\sqcup}_w$ *and* $L^{a\sqcap}_z \subseteq L^{a\sqcup}_z$
- $q^a_g$ *is an element that abstract quantity values associated to a* $\ell^a_g$ *element*
- $q^{a\sqcap}_{kg} : k \in J$ *is an under-approximation of the interval of possible quantities of information associated to* $\ell^a_g$ *with values represented by* $q^a_{kg}$
- $q^{a\sqcap}_{kg} : k \in J'$ *is an over-approximation of the interval of possible quantities of information associated to* $\ell^a_g$ *with values represented by* $q^a_{kg}$
- $q^{a\sqcap}_g \subseteq q^{a\sqcup}_g$.

As a corollary, we define the source set of an atomic datum $\langle\{\ell^a_w : w \in W\}, \{\ell^a_z : z \in Z\}\rangle$ expressed as $\texttt{src}(d)$.
There exists a partial order both on the abstract atomic data and on the quantitative values. While we inherit the former form Cortesi et al., both in implicit and explicit flows, we must define the second.

**Definition 7 (Partial Order on Quantities)** *Let's consider two abstract atomic data elements defined on the same abstract domain for values, labels and quantities:*

$$d_1 = \Big( \Big\langle \{\langle \ell^a_{1i}, L^{a\sqcap}_{1i}, L^{a\sqcup}_{1i}\rangle : i \in I_1\}, \{\langle \ell^a_{1j}, L^{a\sqcap}_{1j}, L^{a\sqcup}_{1j}\rangle : j \in J_1\}\Big\rangle,\ \{\langle \ell^a_{1k}, q^{a\sqcap}_{1k}, q^{a\sqcup}_{1k}\rangle : k \in J_1\} \Big)$$
$$d_2 = \Big( \Big\langle \{\langle \ell^a_{2i}, L^{a\sqcap}_{2i}, L^{a\sqcup}_{2i}\rangle : i \in I_2\}, \{\langle \ell^a_{2j}, L^{a\sqcap}_{2j}, L^{a\sqcup}_{2j}\rangle : j \in J_2\}\Big\rangle,\ \{\langle \ell^a_{2k}, q^{a\sqcap}_{2k}, q^{a\sqcup}_{2k}\rangle : k \in J_2\} \Big)$$

*a partial order between them exists and can be described as follows:*

$$\begin{aligned} d_1 \sqsubseteq d_2 \quad \Leftrightarrow \quad & \forall i \in I_1\, \exists w \in I_2 : \ell^a_{1i} = \ell^a_{2w},\ L^{a\sqcap}_{1i} \supseteq L^{a\sqcap}_{2w},\ L^{a\sqcup}_{1i} \subseteq L^{a\sqcup}_{2w} \quad \wedge \\ & \forall j \in J_1\, \exists z \in J_2 : \ell^a_{1j} = \ell^a_{2z},\ L^{a\sqcap}_{1j} \supseteq L^{a\sqcap}_{2z},\ L^{a\sqcup}_{1j} \subseteq L^{a\sqcup}_{2z} \quad \wedge \\ & \forall k \in J_1\, \exists g \in J_2 : \ell^a_{1k} = \ell^a_{2g},\ q^{a\sqcap}_{1k} \supseteq q^{a\sqcap}_{2g},\ q^{a\sqcup}_{1k} \subseteq q^{a\sqcup}_{2g} \end{aligned}$$

Although we inherit the abstraction and concretization function for the explicit flows, we must redefine such functions in such a way to handle quantities.

**Definition 8 (Quantitative Abstraction function)** *We denote by $\alpha_{\mathtt{Q}}$ be the abstraction function that, applied to $\{(\ell_k, q_k) : k \in J\}$, returns the set $\{(\alpha_{\mathtt{Lab}}(\ell_k), q_k^{a\sqcap}, q_k^{a\sqcup}) : k \in J\}$, where $q_k^{a\sqcap}, q_k^{a\sqcup}$ represent the abstraction of the interval of quantitative values that approximates $q_i$ in the abstract domain $\mathtt{Q}^{\mathtt{a}}$.*

**Definition 9 (Quantitative Abstraction function for Atomic Data)** *Given a concrete atomic datum $d = \Big( \Big\langle \{\langle \ell_i, L_i\rangle : i \in I\}, \{\langle \ell_j, L_j\rangle : j \in J\} \Big\rangle, \{\langle \ell_k, q_k\rangle : k \in J\} \Big)$, we define an abstraction function $\alpha : \wp(\mathbb{D}) \longrightarrow \mathbb{AD}$ as:*

$$\begin{aligned}
\alpha_s(d) \quad = \quad & \Big( \Big\langle \{\langle \alpha_{\mathtt{Lab}}(\ell_i), \alpha_{\mathtt{Lab}}(L_i), \alpha_{\mathtt{Lab}}(L_i)\rangle : i \in I\}, \\
& \{\langle \alpha_{\mathtt{Lab}}(\ell_i), \alpha_{\mathtt{Lab}}(L_j), \alpha_{\mathtt{Lab}}(L_j)\rangle : j \in J\} \Big\rangle, \\
& \{\langle \alpha_{\mathtt{Lab}}(\ell_h), \alpha_{\mathtt{Q}}(q_k)\rangle : h \in J\} \Big)
\end{aligned}$$

*The abstraction function can easily be extended to sets thanks to the least upper bound operator [6]. As for the concretization function, this is defined as an adjoint of the abstraction function [6].*

### 5.1.1 Semantics of Statements

We now define the extended abstract semantics. Expressions are all described in the abstract domain, represented by the Cartesian product of $AD^a$ and $V^a$, that are the Atomic Data abstract domain and the value domain respectively. In figure 5.1 the abstract semantics of statements are described, improved with the implicit flow. Then, in figures 5.2 and 5.3 the same semantics are extended with the quantitative approach. We do not show the abstract semantics of expression, because they are similar to the concrete ones. Moreover, the semantics of expression do not generate implicit flow, consequently no quantity is added, so they are less important at this step of the analysis.

## 5.2 `While` statement 'in the deep'

In the concrete case it is quite simple to compute the number of iterations of a loop, with the use of a counter. Anyway, in the abstract we need an approximation of the number of iterations. Our approach will be composed of two steps:

(a) `while` interval analysis

(b) *extended adexp* collection with quantitative value

$$\begin{array}{rcl}
S^a\llbracket x := sexp\rrbracket(a^a, v^a) & = & (a^a, S^a_v\llbracket x := sexp\rrbracket(v^a)) \\
S^a\llbracket skip\rrbracket(a^a, v^a) & = & (a^a, v^a) \\
S^a\llbracket send(sexp)\rrbracket(a^a, v^a) & = & (a^a, v^a) \\
S^a\llbracket c_1; c_2\rrbracket(a^a, v^a) & = & S^a\llbracket c_2\rrbracket(S^a\llbracket c_1\rrbracket(a^a, v^a)) \\
S^a\llbracket \text{if } c_1 \text{ then } c_2 \text{ else } c_3\rrbracket(a^a, v^a) & = & \\
\multicolumn{3}{r}{\Big\langle S^a_e\llbracket c_2\rrbracket(a^a, S^a_e\llbracket c_1\rrbracket(v^a)) \sqcup S^a_e\llbracket c_3\rrbracket(a^a, S^a_e\llbracket \neg c_1\rrbracket(v^a)),} \\
\multicolumn{3}{r}{S^a_i\llbracket c_2\rrbracket(a^a, S^a_e\llbracket c_1\rrbracket(v^a)) \sqcup S^a_e\llbracket c_1\rrbracket(a^a, v^a)\sqcup} \\
\multicolumn{3}{r}{S^a_i\llbracket c_1\rrbracket(a^a, v^a) \sqcup S^a_i\llbracket c_3\rrbracket(a^a, S^a_e\llbracket \neg c_1\rrbracket(v^a))\sqcup} \\
\multicolumn{3}{r}{S^a_e\llbracket \neg c_1\rrbracket(a^a, v^a) \sqcup S^a_i\llbracket c_1\rrbracket(a^a, v^a)\Big\rangle} \\
S^a\llbracket \text{while } c_1 \text{ do } c_2\rrbracket(a, v) & = & \\
\multicolumn{3}{r}{fix(S^a\llbracket \text{ if } (c_1) \text{ then } (c_2; \text{while } c_1 \text{ do } c_2)\rrbracket(a^a, v^a))}
\end{array}$$

Figure 5.1: Abstract Semantics of Statements

$$\begin{array}{rcl}
S^a\llbracket x := sexp\rrbracket(a^a, \phi^a, v^a) & = & (a^a[x \mapsto S^a_A\llbracket sexp\rrbracket(a^a, \phi^a, v^a)], v^a[x \mapsto S^a_S\llbracket sexp\rrbracket(v^a)]) \\
S^a\llbracket skip\rrbracket(a^a, \phi^a, v^a) & = & (a^a, \phi^a, v^a) \\
S^a\llbracket send(sexp)\rrbracket(a^a, \phi^a, v^a) & = & (a^a, \phi^a, v^a) \\
S^a\llbracket c_1; c_2\rrbracket(a^a, \phi^a, v^a) & = & S^a\llbracket c_2\rrbracket(S^a\llbracket c_1\rrbracket(a^a, \phi^a, v^a))
\end{array}$$

Figure 5.2: Quantitative Abstract Semantics of Statements

### 5.2.1 Step (a): `while` interval analysis

This first step allow us to understand the number of iteration of the loop. We add an operator initialized to 0, and we insert an instruction inside the loop that will increase the counter of one unit at each iteration. We define an abstract semantics of our language on the infinite lattice of intervals. The domain of intervals is a partial order, because the order is present only in some pairs of elements, and it is also a complete lattice, because it's always possible to define lub and glb. The lattice of intervals is defined in the following way:

$$L = \{\bot\} \cup \{[l, u] \mid l \in \mathbb{Z} \cup \{-\infty\},\ u \in \mathbb{Z} \cup \{+\infty\},\ l \leq u\}$$

We also define the interval arithmetics. Given an interval $[a, b] = \{x \in \mathbb{Z} \mid a \leq x \leq b\}$, the basic arithmetic operations are:

- $[a, b] \oplus [c, d] = [a + c, b + d]$
- $[a, b] \ominus [c, d] = [a - d, b - c]$
- $[a, b] \otimes [c, d] = [min(a * c, a * d, b * c, b * d), max(a * c, a * d, b * c, b * d)]$

$$
\begin{aligned}
&S^a[\![\text{if } c_1 \text{ then } c_2 \text{ else } c_3]\!](a^a, \phi^a, v^a) = \\
&\quad \textit{let } (a'^a, \phi'^a, v'^a) = \Big\langle S^a_e[\![c_2]\!](a^a, \phi^a, S^a_e[\![c_1]\!](v^a)) \sqcup S^a_e[\![c_3]\!](a^a, \phi^a, S^a_e[\![\neg c_1]\!](v^a)), \\
&\qquad S^a_i[\![c_2]\!](a^a, \phi^a, S^a_e[\![c_1]\!](v^a)) \sqcup S^a_e[\![c_1]\!](a^a, \phi^a, v^a) \sqcup \\
&\qquad S^a_i[\![c_1]\!](a^a, \phi^a, v^a) \sqcup S^a_i[\![c_3]\!](a^a, \phi^a, S^a_e[\![\neg c_1]\!](v^a)) \sqcup \\
&\qquad S^a_e[\![\neg c_1]\!](a^a, \phi^a, v^a) \sqcup S^a_i[\![c_1]\!](a^a, \phi^a, v^a)\Big\rangle \\
&\quad \textit{in } (a'^a, \phi'^a[(\ell^a_i, q^{a\sqcap}_i, q^{a\sqcup}_i)/(\ell^a_i, h(q^{a\sqcap}_i), h(q^{a\sqcup}_i)) : \ell^a_i \in \texttt{src(ad)}(c_1)]) \\
&S^a[\![\text{while } c_1 \text{ do } c_2]\!](a^a, \phi^a, v^a) = \\
&\quad \textit{fix}(S^a[\![\text{ if } (c_1) \text{ then } (c_2; \text{while } c_1 \text{ do } c_2) \text{ else } \textit{skip} ]\!](a^a, \phi^a, v^a))
\end{aligned}
$$

Figure 5.3: Quantitative Abstract Semantics of Control Statements

- $[a,b] \oslash [c,d] = [min(a/c, a/d, b/c, b/d), max(a/c, a/d, b/c, b/d)]$

Division must be extended, because in this form we cannot have 0 in the second interval. In order to solve this we split the divisor interval into two parts: $[c,d]$ : $[c,-1]$, $[1,d]$. We thus have to compute the division with these two new intervals.

*Example* We propose the following example in order to see interval analysis in action. We have added a $i$ variable as counter:

```
1 x=−2                       1 x=−2    (i = 0)
2 while (x<27)          →    2 while (x<27)
3         x= x+2             3         x= x+2    (i = i + 1)
4 print (x)                  4 print (x)
```

We now perform the analysis:

$$
\begin{cases}
x_1 = [-2,-2],\ i_1 = [0,0] \\
x_2 = (x_1 \cup x_3) \cap [-\infty,\ 26],\ i_2 = (i_1 \cup i_3) \cap [-\infty,\ +\infty] \\
x_3 = x_2 \oplus [2,2],\ i_3 = i_2 \oplus [1,1] \\
x_4 = (x_1 \cup x_3) \cap [27,\ +\infty],\ i_4 = (i_1 \cup i_3) \cap [-\infty,\ +\infty]
\end{cases}
$$

$$\text{it. } 0 \begin{cases} x_1 \mapsto \emptyset,\ i_1 \mapsto \emptyset \\ x_2 \mapsto \emptyset,\ i_2 \mapsto \emptyset \\ x_3 \mapsto \emptyset,\ i_3 \mapsto \emptyset \\ x_4 \mapsto \emptyset,\ i_4 \mapsto \emptyset \end{cases} \quad \text{it. } 1 \begin{cases} x_1 \mapsto [-2,-2],\ i_1 \mapsto [0,0] \\ x_2 \mapsto [-2,-2],\ i_2 \mapsto [0,0] \\ x_3 \mapsto [0,0],\ i_3 \mapsto [1,1] \\ x_4 \mapsto \emptyset,\ i_4 \mapsto \emptyset \end{cases} \quad \text{it. } 2 \begin{cases} x_1 \mapsto [-2,-2],\ i_1 \mapsto [0,0] \\ x_2 \mapsto [-2,0],\ i_2 \mapsto [0,1] \\ x_3 \mapsto [0,2],\ i_3 \mapsto [1,2] \\ x_4 \mapsto \emptyset,\ i_4 \mapsto \emptyset \end{cases}$$

$$\ldots \quad \text{it. } 15 \begin{cases} x_1 \mapsto [-2,-2],\ i_1 \mapsto [0,0] \\ x_2 \mapsto [-2,26],\ i_2 \mapsto [0,14] \\ x_3 \mapsto [0,28],\ i_3 \mapsto [1,15] \\ x_4 \mapsto \emptyset,\ i_4 \mapsto \emptyset \end{cases} \quad \text{it. } 16 \begin{cases} x_1 \mapsto [-2,-2],\ i_1 \mapsto [0,0] \\ x_2 \mapsto [-2,26],\ i_2 \mapsto [0,15] \\ \ldots \\ x_3 \mapsto [27,28],\ i_3 \mapsto [1,15] \end{cases}$$

The upper bound of the interval of variable $i$ returns the number of iterations of the loop. We convert the interval into a quantity dimension, by expressing values as number of bits of the equivalent of the number expressed in binary form.

*Example* The interval of iterations of the previous example, $i \mapsto [1, 15]$, becomes $[1, 4]$, where 4 are the number of bits needed leaked in 15 iterations.

We can improve the interval analysis with the use of a *widening operator* with threshold (see [9], [5], [7]). A widening operator on a partial order $D$ is a binary operator $\nabla : D \to D$ such that:

- it's an upper bound operator $x, y \in P : x \leq x\nabla y,$ and $y \leq x\nabla y.$
- it enforce convergence: for every ascending chain $\{x_i\}$, $i \geq 0$ the a.c. defined by $y_0 = x_0$, $y_{i+1} = y_i \nabla x_{i+1}$ stabilizes after a finite numbers of terms.

### 5.2.2 Step (b): *extended adexp* collection with quantitative value

The analysis is performed as defined in previous section. Then the quantitative value is added at the end of the cycle.

*Example* Consider the following example:

```
secret = read(\ldots)
found = False
while (!Found)
        pwd = user_input()
        found = checkpwd(pwd, secret)
```

The following are the corresponding expressions in the concrete case after the first iteration of the loop:

| | | |
|---|---|---|
| secret | : | $\Big\langle \{\langle l_1,\ \emptyset\rangle\},\ \emptyset \Big\rangle$ |
| found | : | $\Big\langle \{\langle \star,\ \emptyset\rangle\},\ \emptyset \Big\rangle$ |
| !found | : | $\Big\langle \{\langle \star,\ \{(\neg,\star)\}\rangle\},\ \emptyset \Big\rangle$ |
| pwd | : | $\Big\langle \{\langle \star,\ \emptyset\rangle\},\ \{\langle \star,\ \{(\neg,\star)\}\rangle\} \Big\rangle$ |
| found | : | $\Big\langle \{\langle l_1,\ \{(\mathit{checkpwd},\star)\}\rangle\},\ \{\langle \star,\ \{(\neg,\star)\}\rangle\} \Big\rangle$ |

But, if the condition is still true, at the beginning of the second iteration the implicit flow will contain the new definition of the variable *found*, thus each expression inside the scope of the `while` will be:

$$\Big\langle \{\ldots \mathit{explicit\ flow} \ldots\},\ \{\langle l_1,\ \{(\mathit{checkpwd},\star),(\neg,l_1)\}\rangle\} \Big\rangle$$

Notice that the function *checkpw($p_1$,$p_2$)* returns a boolean value, so *one bit*. This means that we are introducing a bit for each iteration in the implicit flow.

If inside the scope of a while we have an operator that obfuscates a confidential data, we must know the quantity of information that is released by the operator. For instance, the operator `checkpwd(`$p_1$`,`$p_2$`)` checks if the password given by the user is the correct one, and returns a boolean value of 1 bit. Thus in this case the analysis captures a single bit at each iteration, and the quantitative value will depend only on the number of iterations. But, there may also exist operators that releases more than one single bit. In such case, we must perform a sort of product of the number of iterations and the released bits. In case the analysis returns an infinite interval of iterations $[1,+\infty]$, we limit the quantity of released information as quantity of bits of the labels used in the `while` scope.

# Chapter 6

# Confidentiality and Obfuscation Policies

This chapter discuss the concepts of Confidentiality and Obfuscation inherited from the work of Cortesi et al. [6], extending them with the quantitative approach. At the end, two verification approaches based on policies will be proposed by using the extended Confidentiality and Obfuscation notions. The confidentiality concept allow us to introduce a differentiation between labels coming from the data-store, characterizing data according to its secrecy.

**Definition 10 (Confidentiality)** *Confidentiality is represented using a lattice of confidentiality levels $S$, thus each label $\ell \in \mathtt{Lab}$ is related to an element $s_\ell \in S$. We define as $\eta$ the function that maps $\ell$ into the corresponding element $s_\ell$.*

Although the confidentiality represent the secrecy degree of the value, we need something to characterize functions designed to hide secret data. For this purpose, Cortesi et al. introduce the notion of *obfuscation degree*, related to an operator. This concept also tell how difficult is for a malicious observer to recover the original value of the label. Indeed an operator that have an high level obfuscation will make it harder to recover the original value.

**Definition 11 (Obfuscation)** *A complete lattice $(\mathtt{O}, \sqsubseteq_{\mathtt{O}})$ is introduced to describe the partial-order relation between operators that introduce different levels of obfuscations. We define obfuscation as a function $\zeta$ that maps each operator to the lattice: $\zeta : Op \rightarrow \mathtt{O}$. If the obfuscation power of $op_1$ is smaller than that of $op_2$, we have $\zeta(op_1) \sqsubseteq_{\mathtt{O}} \zeta(op_2)$.*

## 6.1 Confidentiality of Concrete Atomic Data

After the introduction of the confidentiality and obfuscation concepts, Cortesi et al. combine these notions with atomic data. For this purpose, they look at an under- and over-approximation of confidentiality levels and of obfuscation power of operators respectively, defining first in the concrete and after in the abstract. We extend their notions by adding the quantitative concept and considering it for both explicit and implicit flows. Even if they also consider the case in which only monotonic operators are used for the confidentiality of atomic data, we only refer to the general case. This means that we consider expressions where there is a non-monotonic combination of operators, by assigning obfuscation values to set of operators.

**Definition 12 (Extended Confidentiality of Atomic Data)** *Let $\eta$ be the function that maps labels to the lattice of confidentiality level $\mathsf{S}$. Let $\zeta$ be the function that maps sets of operators to intervals of min and max obfuscation powers, where an interval is defined in $\mathsf{O} \times \mathsf{O}$ and $\mathsf{O}$ is the lattice of obfuscation. A $\pi$ function is added to designate the minimum and the maximum element of the interval. Confidentiality and obfuscation functions are applied both to the explicit and to the implicit flow. Let $\phi$ be a function that assign the quantity of information, and that is applied only to implicit flow labels. The lattice $\mathsf{P}$ represents the quantities of information assignable to a label. Thus the extended confidentiality value of an atomic datum*

$$\Big\langle \{\langle \ell_i, L_i\rangle : i \in I\}, \{\langle \ell_j, L_j\rangle : j \in J\}, \{\langle \ell_k, q_k\rangle \ : k \in J\} \Big\rangle$$

*with respect to $(\eta, \zeta, \phi)$ is the tuple $(sc_{min}, sc_{max}, lc_{min}, lc_{max}, qc_{min}, qc_{max})$, where:*

$$\begin{array}{rcl} sc_{min} & = & \sqcap_{\mathsf{S}}\{\eta(\ell_i) : i \in I, J\} \\ sc_{max} & = & \sqcup_{\mathsf{S}}\{\eta(\ell_i) : i \in I, J\} \end{array} \qquad \begin{array}{rcl} qc_{min} & = & \sqcap_{\mathsf{S}}\{\phi(\ell_j) : j \in J\} \\ qc_{max} & = & \sqcup_{\mathsf{S}}\{\phi(\ell_j) : j \in J\} \end{array}$$

$$\begin{array}{rcl} lc_{min} & = & \sqcap_{\mathsf{O}}\{\pi_1(\zeta(\{op_{ij} : (op_{ij}, \ell_j) \in L_i\})) : i \in I, J\} \\ lc_{max} & = & \sqcup_{\mathsf{O}}\{\pi_2(\zeta(\{op_{ij} : (op_{ij}, \ell_j) \in L_i\})) : i \in I, J\} \end{array}$$

## 6.2 Confidentiality of Abstract Atomic Data

We need something more before defining the extension of the confidentiality also for abstract atomic data. As for the obfuscation, no modifications are required, because operators are the same, so we only need to lift the value. On the other side, we cannot consider a single confidentiality value, but an interval of possible confidentiality values, so the $\eta^a$ function must return an interval of values in $\mathsf{S} \times \mathsf{S}$, where $\mathsf{S}$ is the lattice of confidentialities. Finally, we also need to assign the quantity value to abstract labels, so the $\phi^a$ function must return the interval $\mathsf{P} \times \mathsf{P}$ of possible quantities that a label can introduce in the flow.

**Definition 13 (Extended Confidentiality of Abstract Atomic Data)** *Consider the concrete functions $\eta, \zeta, \phi$ and respective lattices $\mathsf{S}, \mathsf{O}, \mathsf{P}$ already used in the previous definition. Let $\eta^a$ be the function that maps labels to the interval of confidentiality level $\mathsf{S} \times \mathsf{S}$, such that $\eta^a(\ell^a) = [\sqcap\{\eta(\ell) : \ell \in \gamma(\ell^a)\}, \sqcup\{\eta(\ell) : \ell \in \gamma(\ell^a)\}]$. Let $\zeta^a$*

*be the function that maps sets of abstract operators to intervals of min and max obfuscation powers, where an interval is defined in* $\mathtt{O} \times \mathtt{O}$ *and* $\mathtt{O}$ *is the lattice of obfuscation. As for the concrete case, confidentiality and obfuscation functions are applied both to the explicit and to the implicit flow. Let* $\phi^a$ *be a function, applied only to implicit flow labels, that assign the quantity of information to the interval of quantities* $\mathtt{P} \times \mathtt{P}$*, such that* $\phi^a(\ell^a) = [\sqcap\{\phi(\ell) : \ell \in \gamma(\ell^a)\}, \sqcup\{\phi(\ell) : \ell \in \gamma(\ell^a)\}]$*. Finally, a* $\pi$ *function is added to designate the minimum and the maximum element of the intervals. Thus the extended confidentiality value of an abstract atomic datum*

$$\left\langle \{\langle \ell^a_i, L^{a\sqcap}_i, L^{a\sqcup}_i \rangle : i \in I\}, \{\langle \ell^a_j, L^{a\sqcap}_j, L^{a\sqcup}_j \rangle : j \in J\}, \{\langle \ell^a_k, q^{a\sqcap}_k, q^{a\sqcup}_k \rangle : k \in J\} \right\rangle$$

*with respect to* $(\eta^a, \zeta^a, \phi^a)$ *is the tuple* $(sc^a_{min}, sc^a_{max}, lc^a_{min}, lc^a_{max}, qc^a_{min}, qc^a_{max})$*, where:*

$$\begin{array}{lcl@{\qquad}lcl} sc^a_{min} & = & \sqcap_{\mathtt{S}}\{\pi_1(\eta^a(\ell^a_i)) : i \in I, J\} & qc^a_{min} & = & \sqcap_{\mathtt{P}}\{\pi_1(\phi^a(\ell^a_i)) : i \in J\} \\ sc^a_{max} & = & \sqcup_{\mathtt{S}}\{\pi_2(\eta^a(\ell^a_i)) : i \in I, J\} & qc^a_{max} & = & \sqcup_{\mathtt{P}}\{\pi_2(\phi^a(\ell^a_i)) : i \in J\} \end{array}$$

$$\begin{array}{lcl} lc^a_{min} & = & \sqcap_{\mathtt{O}}\{\pi_1(\zeta^a(S)) : S \subseteq \{op^a_{ij} : (op^a_{ij}, \ell^a_j) \in L^{a\sqcup}_i\}, i \in I, J\} \\ lc^a_{max} & = & \sqcup_{\mathtt{O}}\{\pi_2(\zeta^a(S)) : S \subseteq \{op^a_{ij} : (op^a_{ij}, \ell^a_j) \in L^{a\sqcup}_i\}, i \in I, J\} \end{array}$$

In the end of this chapter we propose a way to establish if a given program $P$ can be considered safe or not, by using results coming from our static analysis. In order to do this, we propose two different approaches, both based on verification of the satisfaction of a given policy. The former approach will be focused on the extension of the confidentiality policy described by Cortesi et al. [6], thus yielding a more qualitative perspective. On the other side, the latter will introduce a quantitative policy, grounded on the examination of the quantity of information in the implicit flow.

## 6.3 A Qualitative Approach: extension of the Confidentiality Policy

We extend the original policy with the introduction of the quantity of information notion, by the use of a quantitative threshold. This threshold will represent the maximum quantity of information of labels that is allowed to be present in the implicit flow. We represnt this threshold with the value $\kappa_{qc\_max}$.

**Definition 14 (Extended Confidentiality Policy)** *Let* $\mathtt{Lab}$ *be the set of data source labels,* $\mathtt{S}$*,* $\mathtt{O}$ *and* $\mathtt{P}$ *the lattices of confidentiality, obfuscation and quantity respectively. We define an extended confidentiality policy as a tuple:*

$$\pi^q = (\eta, \zeta, \phi, \kappa_{sc\_max}, \kappa_{lc\_min}, \kappa_{qc\_max})$$

*such that:*

- *$\eta, \phi$ are the functions that assign to each label a specific value in* $\mathsf{S}$ *(confidentiality lattice) and* $\mathsf{P}$ *(quantity lattice) respectively*
- *$\zeta$ is the function that assign to each operator the analogous label in the obfuscation lattice* $\mathsf{O}$
- *$\kappa_{sc\_max}$ is the threshold that represents the maximum confidentiality level for each source*
- *$\kappa_{lc\_min}$ is the threshold that represents the minimum obfuscation level recommended for each operator*
- *$\kappa_{qc\_max}$ is the threshold that represents the maximum quantity of information level that sources can release in the implicit flow*

Thanks to this policy, we can now propose the following definition to check if a given program is compliant to a given policy.

**Definition 15 (Policy Compliance)** *Let $P$ be a program and $X$ the set of expressions generated by our analysis at the end of the program execution. Let $\pi_1^q = (\eta, \zeta, \phi, \kappa_{sc\_max}, \kappa_{lc\_min}, \kappa_{qc\_max})$ be a proposed policy.*
*Given $d \in X$, let $(sc_{min}, sc_{max}, lc_{min}, lc_{max}, qc_{min}, qc_{max})$ be the extended confidentiality value related to $d$ with respect to $(\eta, \zeta, \phi)$. The program $P$ is said to be* policy compliant *with respect to $\pi_1^q$ if:*

$$sc_{max} \sqsubseteq_{\mathsf{S}} \kappa_{sc\_max} \ \wedge \ lc_{min} \sqsupseteq_{\mathsf{O}} \kappa_{lc\_min} \wedge qc_{max} \sqsubseteq_{\mathsf{S}} \kappa_{qc\_max}.$$

In an analogous way, we can define the policy compliance for a given program $P$ in the abstract, by considering $X^a$ as the set of abstract expressions generated by our analysis on the $P$ program and the abstract extended confidentiality value $(sc^a_{min}, sc^a_{max}, lc^a_{min}, lc^a_{max}, qc^a_{min}, qc^a_{max})$ related to $d^a : d^a \in X^a$ with respect to $(\eta^a, \zeta^a, \phi^a)$. In this case, we must consider abstract functions $(\eta^a, \zeta^a, \phi^a)$ also in the policy definition.

Notice that if a quantity interval result bounded, the policy is satisfied. But let's consider some particular cases:

- if the interval is not bounded (infinite bounds), the policy will not be satisfied, because the analysis is revealing possible leakage of information; for instance, this can be the consequence of an infinite loop where part of a confidential label is leaked at each iteration;
- if the interval remains bounded, but the quantity value outperforms the dimension of the label, the label is completely leaked through the implicit flow; in this case, if the quantitative policy is not strict, it could be possibly satisfied, even if the label is leaked.

The following theorem guarantees the satisfaction of policies for concrete executions and it is inherited as-is, because both the implicit flow and the quantitative notion do not alter its formulation.

**Theorem 1** *Let P be a program, A an abstract datastore, $\pi^q$ an extended confidentiality policy. Moreover, let's assume that the program P terminates correctly. If the analysis on P and A satisfies the policy, then every possible execution of the same program on a concrete datastore $\gamma(A)$ will satisfy the same policy $\pi^q$.*

*Proof* The theorem proof is similar to the one of the corresponding theorem in Cortesi et al. [6], but extended by considering also the implicit flow. □

In an orthogonal way, we can inherit also the notion of *sources' confidentiality policy*. If we implement this policy as-is, there will be no interesting improvement, because it will consider only implicit flow as additional notion, using the same formulas used for the explicit one. Instead, what we can do is to implement an analogous policy for sources by considering quantities.

## 6.4 A Quantitative Approach: Implicit Flow Ratio

We now describe a more quantitative approach compared to the other one. We will use a percentage metric that will allow us to establish a policy for the program by supplying a threshold value. The aim of the policy is to grant that the quantity of information in the implicit flow is lower than the given threshold. If a program obtains a metric value greater than the threshold, the policy is not respected. On the opposite, if the flow is higher, that means that an attacker could infer the value of secret labels. Notice that in this kind of policy we do not consider confidentiality and obfuscation. These notion are not useful in this method, so the best approach consists in defining two different policies: one as described in the original version of Cortesi et al. work [6] and the other as described in the following part. Then, the compliance to the two policies will be evaluated separately.
Before defining the policy, we need to introduce some metrics.

**Definition 16 (Total Quantity of Information)** *We introduce a new measure: the total quantity of information potentially leaked in the implicit flow. This measure is obtained by summing all the quantitative values q in a qadexp. We call this quantity ibits. In the abstract, this quantity will be the interval where bound are the minimum and the maximum quantity of information respectively.*

**Definition 17 (Total Data)** *Let $\omega(\ell_1)$ be the function that retrieves the dimension of a label in terms of bits, as already described in the quantitative chapter. We* *introduce a metric to count the dimension of data of all the labels used in an implicit flow Lj. We call this quantity tdata, and we compute it as follows:*

$$tdata = \sum_j 2^{\omega(\ell_j)}\ ,\ \forall j \in J$$

Notice that if we consider a string label, the corresponding data are calculated by multiplying data in a single char times the number of characters in the string.

*Example* For instance, consider a string of 8 characters and the simplified char encoding of $6bits$. The dimension of data will be: $2^6 \times 8$.

We now introduce a measure that represents the percentage of bits of data present in the implicit flow (compared to the labels used in the same flow).

**Definition 18 (Leaked bits)** *Let ibits be the total quantity of information collected as qadexp. Let tbits be the total dimension of all the labels used in the implicit flow. Thus, the ratio between the data generated during implicit flow and the dimension of data corresponding to labels used in the implicit flow, called iratio, is:*

$$iratio := \frac{2^{ibits}}{tdata}$$

*If we multiply iratio by 100, we obtain the percentage of data that comes from labels and that can be understood by a malicious observer thanks to the implicit flow:*

$$lbits := iratio \times 100$$

*we call this measure lbits.*

Thus the policy is described as follows:

**Definition 19 (Quantitative Policy)** *Let* Lab *be the set of data source labels and* P *the lattice of quantities. We define a quantitative confidentiality policy as a tuple* $\sigma^q = (\phi, \kappa_{perc})$ *such that:*

- $\phi$ *is the function that assign to each label a specific value in the quantitative lattice* P
- $\kappa_{perc}$ *is the threshold that represents the maximum percentage of quantity of information that sources can release in the implicit flow.*

**Definition 20 (Quantitative Policy Compliance)** *Let P be a program and X the set of expressions generated by our analysis at the end of the program execution. Let lbits be the percentage of quantity of data in the implicit flow. Let* $\sigma_1^q = (\phi, \kappa_{perc})$ *be a proposed policy. The program P is said to be* policy compliant *with respect to* $\sigma_1^q$ *if:*

$$lbits \sqsubseteq_S \kappa_{perc}.$$

Similarly to the former policy, we now supply a theorem for the latter.

**Theorem 2** *Let P be a program, A an abstract data-store, $\sigma^q$ an quantitative policy. Moreover, let's assume that the program P terminates correctly. If the analysis on P and A satisfies the policy, then every possible execution of the same program on a concrete data-store $\gamma(A)$ will satisfy the same policy $\sigma^q$.*

We produce a further theorem to show that the collection of quantities can reveal leakage even without the need of a policy.

**Theorem 3** *Given a P program and X defined as its set of expression. Let $\ell_1$ be a label associated to the variable with dimension $\omega(\ell_1)$ bits. If the quantity of information $q_1$ in the qadexp associated to $\ell_1$ is higher than the dimension $k_{\ell_1}$, then the label $\ell_1$ is completely leaked.*

*Proof* We prove the theorem by contradiction. Let's suppose that the label $\ell_1$ is not completely leaked. This means that, by the soundness of the analysis, the quantity value $q_1$ collected in the corresponding *quadexp* is lower than the dimension of the label $\omega(\ell_1)$. But we assumed that $q_1$ is higher than the dimension of the label, and this is absurd. Thus the theorem is proven. $\square$

Notice that this theorem is always true with the exception of a quantitative policy with a threshold of 100%, thus allowing complete leakage.

# Chapter 7

# Motivating Examples

We will now test out analysis in real working Android applications. This chapter is divided in two section: the first one will consider the `INMOBI` example, already used in the work of Cortesi et al. [6]. Instead, the second part will be focused in the DroidBench application set [1], created by the Secure Software Engineering group of the Technische Universität Darmstadt. This set is open source and represent a testing ground specifically designed for static and dynamic security tools. We have chosen some examples that specifically present an implicit flow inside the code.
For the sake of readability, we simplified some library functions. Anyway, we did not manage to avoid every function not covered by our semantics. So new semantic rules are added in examples where new functions are present in the code. In each example, a concrete analysis is proposed, based on a given assumption. Then we perform abstract analysis, followed by an abstract quantitative analysis and the verification of a policy compliance. For simplicity, we have chosen to evaluate quantitative values only after the abstract analysis, thus abstract semantics (and concrete too) in the examples will not contain the *qadexp*.
Notice that in every analysis, for each example, expressions at a given program point are preceded by the code line associated to the variable or command that generated such expression, such that:

$$\mathit{variable}_{\mathit{code\ line}} \; : \; \mathit{expression}.$$

```
1 public class IMBanner {
2  public void loadBanner() {
3   UserInfo user = new UserInfo();
4   user.updateInfo();
5   BannerView banner = new BannerView(user);
6   banner.loadNewAd();
7   show(banner);
8  }
9 }
10
11 public class BannerView {
12  private UserInfo user;
13  BannerView(UserInfo user) {
14   this.user = user;
15  }
16  void loadNewAd() {
17    String url = "http :// www.inmobi.com/...?id="
18       + user.id + "&lang="+user.language+
19       "&country=" + user.country + "&loc=" + user.loc;
20   //open an http connection with url
21   //update the new ad to display
22  }
23 }
24 public class UserInfo {
25  String language;
26  String country;
27  String id;
28  Location loc;
29
30  void updateInfo() {
31   Locale localLocale = Locale.getDefault();
32   language = localLocale.getLanguage();
33   country = localLocale.getCountry();
34   String androidId = Settings.Secure.getAndroidId();
35   id = MessageDigest.hashSHA1(androidId);
36   loc = LocationManager.getLastKnownLocation();
37  }
38 }
```

Figure 7.1: `Inmobi` Library

## 7.1 INMOBI example

In this first example we consider the `Inmobi` library, one of the most important advertising engine for the Android platform. In figure 7.1 we present a portion of the library involved in the loading of a banner, where some functions have been simplified for the sake of readability. The importance of this code snippet for our purpose is that there is the collection of sensible data from the datastore, like the *androidId* and the geographic location of the device, and the leakage of this data to a server.

### 7.1.1 Concrete Analysis

We first analyse the concrete datastore defined for this example. We consider the following labels: $\langle \texttt{Language}, \emptyset \rangle$ that contains the system language, $\langle \texttt{Country}, \emptyset \rangle$ that define the Country value,$\langle \texttt{AndroidId}, \emptyset \rangle$ for the AndroidId and $\langle \texttt{Location}_i, \emptyset \rangle : i \in \mathbb{N}$ for the location of the device, that can change with different calls. Thus we have:

$$\begin{aligned} \mathit{user.language}_{32} &: \{\langle \texttt{Language}, \emptyset \rangle\} \\ \mathit{user.country}_{33} &: \{\langle \texttt{Country}, \emptyset \rangle\} \\ \mathit{user.id}_{35} &: \{\langle \texttt{AndroidId}, \{(\texttt{hash.AndroidId})\} \rangle\} \\ \mathit{user.loc}_{36} &: \{\langle \texttt{Location}_1, \emptyset \rangle\} \end{aligned}$$

We can now perform the concrete analysis:

$$\begin{aligned} url_{17} \; : \; & \Big\langle \{\langle \mathtt{AndroidId}, \{(\mathtt{hash}, \mathtt{AndroidId}), (\circ, \mathtt{Language}), (\circ, \mathtt{Country}), (\circ, \mathtt{Location}_1)\}\rangle\}, \{\emptyset\} \Big\rangle, \\ & \Big\langle \{\langle \mathtt{Location}_1, \{(\circ, \mathtt{AndroidId})\}\rangle\}, \{\emptyset\} \Big\rangle \end{aligned}$$

Notice that there are no `while` or `if` statement, so we have no implicit flow.

### 7.1.2 Abstract Analysis

As for the abstract analysis, we do not need to abstract `Language`, `Country`, and `AndroidId` because their value is always the same during the program execution. As for the location, we define $\mathtt{Location}^{\mathtt{pp}}$ as abstraction for all the possible location values $\mathtt{Location}_i : i \in \mathbb{N}$ generated at the program line `pp`. Thus, the abstract analysis does not differ so much from the concrete one, with the exception of the label Location.

### 7.1.3 Quantitative Analysis

In this code snippet there are no statements that generates implicit flow, thus the quantity of information is null.

### 7.1.4 Policy Compliance

If we consider whatever quantitative policy, this will always trivially be respected, because there are no quantitative value. As already said, this is due to the lack of control statements inside the code. This shows that our extended analysis does not add so much to the work of Cortesi if no implicit flows are present in the analysed program.

## 7.2 ImplicitFlow1

The first example is an application that get the DeviceId and then leaks it, one time with a low obfuscation and the second one with an high obfuscation. We modify the original example by introducing functions simpleFunct and hardFunct, instead of complex code portions. Both convert one element of the `IMEI` to a char. The main difference between the two is the obfuscation power: while in the former is low, in the latter is high.

```java
public class ImplicitFlow1 extends Activity {

  protected void onCreate(...) {
    // ...

    String imei = getDeviceId(); //device id
    String obfuscatedIMEI
      = obfuscateIMEI(imei);
    writeToLog(obfuscatedIMEI);

    obfuscatedIMEI
      = reallyHardObfIMEI(imei);
    writeToLog(obfuscatedIMEI);
  }

  private String obfuscateIMEI(String imei){
    String result = "";
    char[] imeiAsChar = imei.toCharArray();
    int len = imeiAsChar.length();
    int i = 0;

    while (i < len){
      result += simpleFunct(imeiAsChar[i]);
      //returns 'a' for '0', 'b' for '1', ...
```

```
        i++;
    }
    return result;
}
private String reallyHardObfIMEI(String imei){
    Integer[] numbers
        = new Integer []{0,1,...,56,57};

    char[] imeiAsChar = imei.toCharArray();
    String result = "";
    int len = imeiAsChar.length();
    int i = 0;

    while (i < len){
        result
            += hardFunct(imeiAsChar[i], numbers);
        i++;
    }

    return result;
}

private void writeToLog(String message){
    Log.i("INFO", message); //sink
}
}
```

First we must define new semantic rules for new functions:

$$\begin{array}{rcl}
S_A[\![getDeviceId()]\!](a,v) & = & \{\langle S_L[\![lexp]\!](a,v),\emptyset\rangle\} \\
S_A[\![toCharArray(sexp)]\!](a,v) & = & \{\langle \ell_1, L_1 \cup \{(toCharArray,\ell_1)\}\rangle : \langle \ell_1, L_1\rangle \in S_A[\![sexp_1]\!](a,v)\} \\
S_A[\![length(sexp)]\!](a,v) & = & \{\langle \ell_1, L_1 \cup \{(lenght,\ell_1)\}\rangle : \langle \ell_1, L_1\rangle \in S_A[\![sexp_1]\!](a,v)\} \\
S_A[\![Log(sexp)]\!](a,v) & = & (a,v) \\
S_A[\![simpleFunct(sexp)]\!](a,v) & = & \{\langle \ell_1, L_1 \cup \{(simpleFunct,\ell_1)\}\rangle : \langle \ell_1, L_1\rangle \in S_A[\![sexp_1]\!](a,v)\} \\
S_A[\![hardFunct(sexp,s)]\!](a,v) & = & \{\langle \ell_1, L_1 \cup \{(hardFunct,\star)\}\rangle : \langle \ell_1, L_1\rangle \in S_A[\![sexp_1]\!](a,v)\}
\end{array}$$

where *getDeviceId* returns the `IMEI` from the datastore, *toCharArray* covert the label to a char and *lenght* returns the dimension (in integer) of an array and *Log* writes the argument to a log file. As for arrays, when we are referring to a single element of the array, we assume to perform a $S_A[\![sub(sexp,k_1,k_2)]\!](a,v)$ where $k_1$ and $k_2$ are the same element and are used as a sort of index in the array.

### 7.2.1 Concrete Analysis

In this example user input is not required. We assume that the `IMEI` is contained in the datastore and can be retrieved thanks to the *getDeviceId* function, that behaves like a $S_A[\![read(lexp)]\!](a,v)$. We also add a counter that allows to count the number of iterations in the loop. Trivially, at the end of each cycle this number will be equal to the dimension of the `IMEI` (`IMEI` dimension is of 14 characters)

$$
\begin{array}{rcl}
\text{imei}_{6} & : & \Big\langle\{\langle\ell_1,\ \emptyset\rangle\},\ \emptyset\Big\rangle \\
\text{result}_{17} & : & \Big\langle\{\langle\star,\ \emptyset\rangle\},\ \emptyset\Big\rangle \\
\text{imeiAsChar}_{18} & : & \Big\langle\{\langle\ell_1,\ \{(\mathit{toCharArray},\ell_1)\}\rangle\},\ \emptyset\Big\rangle \\
\text{len}_{19} & : & \Big\langle\{\langle\ell_1\ \{(\mathit{length},\ell_1)\}\rangle\},\ \emptyset\Big\rangle \\
\text{i}_{20} & : & \Big\langle\{\langle\star,\emptyset\rangle\},\ \emptyset\Big\rangle \\
\mathit{while\ cond}_{22} & : & \Big\langle\{\langle\ell_1,\{(\mathit{length},\ell_1),(>,\star)\}\rangle\},\ \emptyset\Big\rangle \quad (\mathit{count}_1 = 0) \\
\text{imeiAsChar[i]}_{23} & : & \Big\langle\{\langle\ell_1,\{([\mathit{sub},\star,\star],\ell_1)\}\rangle\},\ \{\langle\ell_1,\{(\mathit{length},\ell_1),(>,\star)\}\rangle\}\Big\rangle \\
\text{result}_{23} & : & \Big\langle\{\langle\ell_1,\{[\mathit{sub},\star,\star],\ell_1),(\mathit{simpleFunct},\ell_1)\}\rangle\},\ \{\langle\ell_1,\{(\mathit{length},\ell_1),(>,\star)\}\rangle\}\Big\rangle \\
\text{i}_{25} & : & \Big\langle\{\langle\star,\{(+,\ \star)\}\rangle\},\ \{\langle\ell_1,\{(\mathit{length},\ell_1),(>,\star)\}\rangle\}\Big\rangle \\
\dots & & \textit{(after loop exit the condition in the flow is inverted),} \quad (\mathit{count}_1 = 14) \\
\mathit{log}_{48} & : & \Big\langle\{\langle\ell_1,\{[\mathit{sub},\star,\star],\ell_1),(\mathit{simpleFunct},\ell_1)\}\rangle\}, \\
 & & \quad \{\langle\ell_1,\{(\mathit{length},\ell_1),(>,\star)(<,\star)\}\rangle\}\Big\rangle \\
\text{numbers}_{30} & : & \Big\langle\{\langle(\star,\emptyset)\rangle\},\ \emptyset\Big\rangle \\
\text{imeiAsChar}_{33} & : & \Big\langle\{\langle\ell_1,\ \{(\mathit{toCharArray},\ell_1)\}\rangle\},\ \emptyset\Big\rangle \\
\text{result}_{34} & : & \Big\langle\{\langle\star,\ \emptyset\rangle\},\ \emptyset\Big\rangle \\
\text{len}_{35} & : & \Big\langle\{\langle\ell_1\ \{(\mathit{length},\ell_1)\}\rangle\},\ \emptyset\Big\rangle \\
\text{i}_{36} & : & \Big\langle\{\langle\star,\emptyset\rangle\},\ \emptyset\Big\rangle \\
\mathit{while\ cond}_{38} & : & \Big\langle\{\langle\ell_1,\{(\mathit{length},\ell_1),(>,\star)\}\rangle\},\ \emptyset\Big\rangle \quad (\mathit{count}_2 = 0) \\
\text{imeiAsChar[i]}_{39} & : & \Big\langle\{\langle\ell_1,\{([\mathit{sub},\star,\star],\ell_1)\}\rangle\},\ \{\langle\ell_1,\{(\mathit{length},\ell_1),(>,\star)\}\rangle\}\Big\rangle \\
\text{result}_{39} & : & \Big\langle\{\langle\ell_1,\{[\mathit{sub},\star,\star],\ell_1),(\mathit{hardFunct},\star)\}\rangle\},\ \{\langle\ell_1,\{(\mathit{length},\ell_1),(>,\star)\}\rangle\}\Big\rangle \\
\text{i}_{41} & : & \Big\langle\{\langle\star,\{(+,\ \star)\}\rangle\},\ \{\langle\ell_1,\{(\mathit{length},\ell_1),(>,\star)\}\rangle\}\Big\rangle \\
\dots & & \textit{(after loop exit the condition in the flow is inverted),} \quad (\mathit{count}_2 = 14) \\
\mathit{log}_{48} & : & \Big\langle\{\langle\ell_1,\{[\mathit{sub},\star,\star],\ell_1),(\mathit{hardFunct},\star)\}\rangle\}, \\
 & & \quad \{\langle\ell_1,\{(\mathit{length},\ell_1),(>,\star)(<,\star)\}\rangle\}\Big\rangle
\end{array}
$$

As we can see from the concrete analysis, it does not change so much between the first and the second function. Indeed, both uses a loop, and in both the conditional expressions depend on the dimension of the secret label. Notice that if the loop condition does not depend on what happens in the loops scope, the implicit flow generated only depends on the dimension.

### 7.2.2 Abstract Analysis

As there is no user inputs in this program, the abstract analysis is quite similar to the concrete one. Changes consists in the collection of both the normal and the inverse relational operator for the conditional expression of loops, and the use of the interval analysis for the acquirement of the number of iteration. Notice that, as the dimension of the `IMEI` (remember that we assumed an `IMEI` of 14 characters) does not change, the corresponding label does not need to be abstracted, as its value is persistent during the execution. For the same reason, also its dimension does not change and loops will always performs the same number of iterations.

$$\begin{array}{rcl}
\text{imei}_{6} & : & \Big\langle\{\langle \ell_1,\ \emptyset,\ \emptyset\rangle\},\ \emptyset\Big\rangle \\
\text{result}_{17} & : & \Big\langle\{\langle \star,\ \emptyset,\ \emptyset\rangle\},\ \emptyset\Big\rangle \\
\text{imeiAsChar}_{18} & : & \Big\langle\{\langle \ell_1,\ \{(\mathit{toCharArray},\ell_1)\},\ \{(\mathit{toCharArray},\ell_1)\}\rangle\},\ \emptyset\Big\rangle \\
\text{len}_{19} & : & \Big\langle\{\langle \ell_1,\ \{(\mathit{length},\ell_1)\},\ \{(\mathit{length},\ell_1)\}\rangle\},\ \emptyset\Big\rangle \\
\text{i}_{20} & : & \Big\langle\{\langle \star,\ \emptyset,\ \emptyset\rangle\},\ \emptyset\Big\rangle \\
\textit{while cond}_{22} & : & \Big\langle\{\langle \ell_1,\{(\mathit{length},\ell_1),(>,\star)\},\{(\mathit{length},\ell_1),(>,\star)\}\rangle\},\ \emptyset\Big\rangle \\
\text{imeiAsChar[i]}_{23} & : & \Big\langle\{\langle \ell_1,\{([\mathit{sub},\star,\star],\ell_1)\}, \\
 & & \{([\mathit{sub},\star,\star],\ell_1)\}\rangle\},\ \{\langle \ell_1,\{(\mathit{length},\ell_1)\},\{(\mathit{length},\ell_1),(>,\star),(<,\star)\}\rangle\}\Big\rangle \\
\text{result}_{23} & : & \Big\langle\{\langle \ell_1,\{[\mathit{sub},\star,\star],\ell_1),(\mathit{simpleFunct},\ell_1)\},\{[\mathit{sub},\star,\star],\ell_1),(\mathit{simpleFunct},\ell_1)\}\rangle\}, \\
 & & \{\langle \ell_1,\{(\mathit{length},\ell_1)\},\{(\mathit{length},\ell_1),(>,\star),(<,\star)\}\rangle\}\Big\rangle \\
\text{i}_{25} & : & \Big\langle\{\langle \star,\{(+,\ \star)\}\rangle\}, \\
 & & \{\langle \ell_1,\{(\mathit{length},\ell_1)\},\{(\mathit{length},\ell_1),(>,\star),(<,\star)\}\rangle\}\Big\rangle \\
\mathit{log}_{48} & : & \Big\langle\{\langle \ell_1,\{[\mathit{sub},\star,\star],\ell_1),(\mathit{simpleFunct},\ell_1)\},\{[\mathit{sub},\star,\star],\ell_1),(\mathit{simpleFunct},\ell_1)\}\rangle\}, \\
 & & \{\langle \ell_1,\{(\mathit{length},\ell_1)\},\{(\mathit{length},\ell_1),(>,\star),(<,\star)\}\rangle\}\Big\rangle \\
\text{numbers}_{30} & : & \Big\langle\{\langle(\star,\emptyset,\emptyset)\rangle\},\ \emptyset\Big\rangle \\
\text{imeiAsChar}_{33} & : & \Big\langle\{\langle \ell_1,\ \{(\mathit{toCharArray},\ell_1)\},\ \{(\mathit{toCharArray},\ell_1)\}\rangle\},\ \emptyset\Big\rangle \\
\text{result}_{34} & : & \Big\langle\{\langle \star,\ \emptyset,\ \emptyset\rangle\},\ \emptyset\Big\rangle \\
\text{len}_{35} & : & \Big\langle\{\langle \ell_1,\ \{(\mathit{length},\ell_1)\},\ \{(\mathit{length},\ell_1)\}\rangle\},\ \emptyset\Big\rangle \\
\text{i}_{36} & : & \Big\langle\{\langle \star,\ \emptyset,\ \emptyset\rangle\},\ \emptyset\Big\rangle \\
\textit{while cond} & : & \Big\langle\{\langle \ell_1,\{(\mathit{length},\ell_1),(>,\star)\},\{(\mathit{length},\ell_1),(>,\star)\}\rangle\},\ \emptyset\Big\rangle \\
\text{imeiAsChar[i]}_{39} & : & \Big\langle\{\langle \ell_1,\{([\mathit{sub},\star,\star],\ell_1)\},\{([\mathit{sub},\star,\star],\ell_1)\}\rangle\}, \\
 & & \{\langle \ell_1,\{(\mathit{length},\ell_1)\},\{(\mathit{length},\ell_1),(>,\star),(<,\star)\}\rangle\}\Big\rangle \\
\text{result}_{39} & : & \Big\langle\{\langle \ell_1,\{[\mathit{sub},\star,\star],\ell_1),(\mathit{hardFunct},\star)\},\{[\mathit{sub},\star,\star],\ell_1),(\mathit{hardFunct},\star)\}\rangle\}, \\
 & & \{\langle \ell_1,\{(\mathit{length},\ell_1)\},\{(\mathit{length},\ell_1),(>,\star),(<,\star)\}\rangle\}\Big\rangle \\
\text{i}_{41} & : & \Big\langle\{\langle \star,\{(+,\ \star)\}\rangle\}, \\
 & & \{\langle \ell_1,\{(\mathit{length},\ell_1)\},\{(\mathit{length},\ell_1),(>,\star),(<,\star)\}\rangle\}\Big\rangle \\
\mathit{log}_{48} & : & \Big\langle\{\langle \ell_1,\{[\mathit{sub},\star,\star],\ell_1),(\mathit{hardFunct},\star)\},\{[\mathit{sub},\star,\star],\ell_1),(\mathit{hardFunct},\star)\}\rangle\}, \\
 & & \{(\mathit{length},\ell_1),(>,\star),(<,\star)\}\rangle\}\Big\rangle
\end{array}$$

Notice that at the end of the loop the implicit flow contains both original and inverse conditional operators.

### 7.2.3 Quantitative Analysis

Generally, the interval analysis is computed in order to infer the minimum and maximum number of iteration of the loop. Thanks to these two values, it is possible to build the interval of quantities, that will tell the minimum and the maximum of the revealed information in the implicit flow. Anyway, in this example the number of iteration does not change, because it is performed on the fixed dimension of the `IMEI` (see previous notes at the beginning of the abstract analysis). Thus, we already know the number of iterations, that is 14, without the need to perform interval analysis. Moreover, we knows that the conditional expression of the two loops does not depend on a function inside the scope of the loop. This means that only *one bit* is leaked for each iteration. We now infer the number of bits using the method described in the quantitative chapter:

$$quantity = \lfloor log_2(n\_iterations) \rfloor + 1 \ = 4 \ bits$$

So, in both loops, the *qadexp* will be $\langle \ell_1, 4, 4 \rangle$.

### 7.2.4 Policy Compliance

We do not consider confidentiality and obfuscation and we propose a quantitative threshold $k_{perc}$ that define 20% as the maximum percentage of released data. We apply a quantitative policy such that if the quantitative flow percentage of information *lbits* is higher than the threshold, the program will not be policy compliant. In this example, we release only 4 *bits* of the label $\ell_1$, that corresponds to the `IMEI`. A simplified version of the `IMEI`, with the character encoding proposed in the quantitative chapter, has a *tdata* value of $2^6 \times 14 = 896$. So the corresponding *lbits* value will be: $(2^4 \times 100) \div 896 \approx 1.8\%$ . Thus the percentage of information released is lower than the threshold, so the program respects the policy.

## 7.3 ImplicitFlow2

This Android application require to the user the insertion of a password. Then, this password is compared to the correct one, that comes from the datastore. After the evaluation, a message is saved to a log file. This operation carries an implicit flow, because it tells something about the secret value.

```
public class ImplicitFlow2 extends Activity {

  protected void onCreate(...){
    // ...
  }

  public void checkPassword(View view){
    String userInputPassword = //user input
    String superSecure = //secret password

    if (checkpwd(superSecure,userInputPassword))
      passwordCorrect = true;
    else
      passwordCorrect = false;

    if (passwordCorrect)
      Log.i("INFO", "Password is correct");
    else
      Log.i("INFO", "Password is not correct");
  }
}
```

### 7.3.1 Concrete Analysis

Before starting an analysis, we must define some semantics peculiar to this example. In particular, we must define the semantics for the function `Log`. For this purpose, we reuse the semantics of *send*: $S_A[\![Log(sexp)]\!](a, v) = (a, v)$. This kind of expression does not generate implicit flow, thus no quantitative value. As for the abstract version, the semantics is similar.
We now perform a concrete analysis, supposing that the password inserted by the user is not the correct one. We define the concrete datastore where we have only the label that corresponds to the variable *superSecure*, thus $\{\langle \ell_1, \rangle\}$.

$$\begin{array}{rcl}
\text{userInputPassword}_8 & : & \Big\langle \{\langle \star,\ \emptyset\rangle\},\ \emptyset \Big\rangle \\
\text{superSecure}_9 & : & \Big\langle \{\langle \ell_1,\ \emptyset\rangle\},\ \emptyset \Big\rangle \\
\textit{if condtion}_{11} & : & \Big\langle \{\langle \ell_1,\ \{(checkpwd, \star)\}\rangle\},\ \emptyset \Big\rangle \\
\text{passwordCorrect}_{14} & : & \Big\langle \{\langle \star,\ \emptyset\rangle\},\ \{\langle \ell_1,\ \{(checkpwd, \star), (<, \star)\} \Big\rangle \\
2^{nd}\ \textit{if condition}_{16} & : & \Big\langle \{\langle \star,\ \emptyset\rangle\},\ \{\langle \ell_1,\ \{(checkpwd, \star), (<, \star)\} \Big\rangle \\
\text{log}_{19} & : & \Big\langle \emptyset,\ \{\langle \ell_1,\ \{(checkpwd, \star), (<, \star)\} \Big\rangle
\end{array}$$

### 7.3.2 Abstract Analysis

We now perform an abstract analysis on the given program. We assume a string abstraction $\langle NULL,\ \top\rangle$ and no abstraction for labels (we have only one label, and it is always the same).

$$\begin{array}{rcl}
\text{userInputPassword}_8 & : & \Big\langle \{\langle \star,\ \emptyset, \emptyset\rangle\},\ \emptyset \Big\rangle \\
\text{superSecure}_9 & : & \Big\langle \{\langle \ell_1,\ \emptyset, \emptyset\rangle\},\ \emptyset \Big\rangle \\
\textit{if condtion}_{11} & : & \Big\langle \{\langle \ell_1,\ \{(checkpwd, \star)\},\ \{(checkpwd, \star)\}\}\rangle\},\ \emptyset\} \Big\rangle \\
\text{passwordCorrect}_{12,14} & : & \Big\langle \{\langle \star,\ \emptyset, \emptyset\rangle\},\ \{\langle \ell_1,\ \{(checkpwd, \star)\}, \{(checkpwd, \star), (>, \star), (<, \star)\}\rangle \Big\rangle \\
2^{nd}\ \textit{if condition}_{16} & : & \Big\langle \{\langle \star,\ \{(>, \star)\}\rangle\},\ \{\langle \ell_1,\ \{(checkpwd, \star)\}, \{(checkpwd, \star), (>, \star), (<, \star)\}\rangle \Big\rangle \\
\text{log}_{17,19} & : & \Big\langle \emptyset,\ \{\langle \ell_1,\ \{(checkpwd, \star)\}, \{(checkpwd, \star), (>, \star), (<, \star), (>, \star), (<, \star)\}\rangle \Big\rangle
\end{array}$$

### 7.3.3 Quantitative Analysis

Remember that after the abstraction, quantities are collected with the minimum and the maximum of the interval of possible quantity of information. In the current program, this value appears with the first `if` condition, by introducing a bit of information:

$$\textit{if condtion}_{11} \quad : \quad \Big(\Big\langle \{\langle l_1,\ \{(checkpwd, \star)\},\ \{(checkpwd, \star)\}\}\rangle\},\ \{\emptyset, \emptyset\} \Big\rangle, \langle \ell_1, 1, 1\rangle\Big)$$

Notice that this value will remain the same in the rest of the abstract expressions, even if there is another `if` statement. This happens because in the second `while` the conditional expression is evaluated on an explicit flow that does not contain any labels (even if they are present in the implicit flow).

### 7.3.4 Policy Compliance

Let's consider `L` < `M` < `H` as the lattice for confidentiality, obfuscation and also quantity of information. We consider that $\ell_1$ has `H` confidentiality level, the *chechpwd* function has `H` obfuscation level (it releases only a *bit* of information) and the released information in the implicit flow has `L` quantity level, because the total amount of released information is of only 1*bit*.
We apply an extended confidentiality policy such that data, with level of confidentiality $H$, can be released only if the associated level of released quantity is equal or less than $L$. In this case, the program is compliant to the policy because the released data has level $L$.

## 7.4 ImplicitFlow3

As the previous example, also this one require the insertion of a password that will compared to the correct one. Anyway, in this case the information is leaked through the creation of new objects.

```
 1 public class ImplicitFlow3 extends Activity {
 2
 3   protected void onCreate(...) {
 4     // ...
 5   }
 6
 7   public void leakData(View view){
 8     String userInputPassword = //user input
 9     String superSecure = //secret password
10
11     Interface classTmp;
12     if (checkpwd(superSecure,userInputPassword))
13       classTmp = new ClassA();
14     else
15       classTmp = new ClassB();
16
17     classTmp.leakInfo();
18   }
19   interface Interface{
20     public void leakInfo ();
21   }
22
23   public class ClassA implements Interface{
24       public void leakInfo (){
25         Log.i("INFO", "pwd correct");
26       }
27   }
28
29   public class ClassB implements Interface{
30       public void leakInfo (){
31         Log.i("INFO", "pwd incorrect");
32       }
33   }
34 }
```

### 7.4.1 Concrete Analysis

We reuse the `Log` semantic described in the previous example. We suppose that the password inserted by the user is the correct one. We define the concrete datastore where we have only the label that corresponds to the variable *superSecure*, thus $\{\langle \ell_1, \rangle\}$.

$$\begin{array}{rcl}
\text{userInputPassword}_8 & : & \Big\langle \{\langle \star,\ \emptyset \rangle\},\ \emptyset \Big\rangle \\
\text{superSecure}_9 & : & \Big\langle \{\langle \ell_1,\ \emptyset \rangle\},\ \emptyset \Big\rangle \\
\textit{if condtion}_{12} & : & \Big\langle \{\langle \ell_1,\ \{(\textit{checkpwd}, \star), (>, \star)\} \rangle\},\ \emptyset \Big\rangle \\
\text{classTmp}_{13} & : & \Big\langle \{\langle \star,\ \emptyset \rangle\},\ \{\langle \ell_1,\ \{(\textit{checkpwd}, \star), (>, \star)\} \rangle\} \Big\rangle \\
\textit{leakInfo}_{25} & : & \Big\langle \emptyset,\ \{\langle \ell_1,\ \{(\textit{checkpwd}, \star), (\textit{assert}, l_1)\} \rangle\} \Big\rangle
\end{array}$$

### 7.4.2 Abstract Analysis

We assume a string abstraction $\langle NULL,\ \top\rangle$ and no abstraction for labels, as in the previous example.

$$\begin{array}{rcl}
\text{userInputPassword}_8 & : & \Big\langle \{\langle \star,\ \emptyset,\ \emptyset\rangle\},\ \emptyset \Big\rangle \\
\text{superSecure}_9 & : & \Big\langle \{\langle \ell_1,\ \emptyset,\ \emptyset\rangle\},\ \emptyset \Big\rangle \\
\textit{if condtion}_{12} & : & \Big\langle \{\langle \ell_1,\ \{(\mathit{checkpwd},\star),(>,\star)\},\{(\mathit{checkpwd},\star),(>,\star)\}\rangle\},\ \emptyset \Big\rangle \\
\text{classTmp}_{13,14} & : & \Big\langle \{\langle \star,\ \emptyset,\ \emptyset\rangle\},\ \{\langle \ell_1,\ \{(\mathit{checkpwd},\star)\},\ \{(\mathit{checkpwd},\star),(>,\star),(<,\star)\}\rangle\} \Big\rangle \\
\textit{leakInfo}_{25,31} & : & \Big\langle \emptyset,\ \{\langle \ell_1,\ \{(\mathit{checkpwd},\star)\}\rangle\},\{\langle \ell_1,\ \{(\mathit{checkpwd},\star),(>,\star),(<,\star)\}\rangle\} \Big\rangle
\end{array}$$

### 7.4.3 Quantitative Analysis

In this example there is only a statement that generates implicit flow. This statement is the `if` where the password supplied by the user is compared to the one present in the datastore. This statement produces *1 bit* of quantity of information:

$$\textit{if condtion}_{12} \quad : \quad \Big(\Big\langle \{\langle \ell_1,\ \{(\mathit{checkpwd},\star),(>,\star)\}\rangle\},\ \emptyset \Big\rangle, \langle \ell_1, 1, 1\rangle\Big)$$

This value will remain the same in all the following *qadexp*s.

### 7.4.4 Policy Compliance

As in the first example, we do not consider confidentiality and obfuscation and we propose a quantitative threshold $k_{perc}$ that define 10% as the maximum percentage of released data. We apply a quantitative policy such that if the quantitative flow percentage of information *lbits* is higher than the threshold, the program will not be policy compliant. In this example, we release only 1 *bits* of the label $\ell_1$, that corresponds to the *superSecure*. If we suppose a password of 8 char and an encoding like the one described in the quantitative chapter (6 *bits* for each char), the total data of *superSecure* is of $2^6 \times 8 = 512$. The corresponding *lbits* value will be: $(2^1 \ \times 100) \div 512 \approx 0.4\%$. Thus the percentage of information released is lower than the threshold, so the program respects the policy.

# Chapter 8

# Conclusions

In this thesis we presented a framework able to analyse in a qualitative way both explicit and implicit flows. Moreover this framework is also capable of collecting quantities of information in the implicit flows, allowing a better modelling of the information release. Thanks to this design, the framework can support complex hybrid policies, that's to say policies that can grant both qualitative and quantitative threshold. It is clear that such features well represent the synthesis of the two different approaches, the qualitative and the quantitative one, bringing a new perspective in this research field.
The practical evaluation of this analysis by an automatic tool is the mandatory step in order to complete this work.
A lot of research themes are still open. First of all, this work can be further extended with some enhancements. The most urgent ones are the improvement of the collecting semantics by the introduction of the evaluation of strong equality in conditional expression of control statements, and the introduction of exception handling. The latter, in particular, would be very important, because of the involvement of exceptions in the generation of implicit flows. Moreover, it would be interesting to address also the issue of differential privacy, when the inter-application data flows are considered, on top of valuable works in this area [20]. Our analysis should fit well also in this context, because of the emphasis on expressions (that might be released by different applications).

# Bibliography

[1] Droidbench. `http://sseblog.ec-spride.de/tools/droidbench/`. Accessed: 2015-02-09.

[2] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not. 49*, 6 (June 2014), 259–269.

[3] Clark, D., Hunt, S., and Malacaria, P. Quantitative analysis of the leakage of confidential data. *Electronic Notes in Theoretical Computer Science 59*, 3 (November 2002), 1 – 14. Quantitative Aspects of Programming Languages (Satellite Event for PLI 2001).

[4] Clark, D., Hunt, S., and Malacaria, P. Quantified interference for a while language, 2004.

[5] Cortesi, A. Widening operators for abstract interpretation. In *In SEFM 08: Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods* (2008), IEEE Computer Society, pp. 31–40.

[6] Cortesi, A., Ferrara, P., Pistoia, M., and Tripp, O. Datacentric semantics for verification of privacy policy compliance by mobile applications. In *Verification, Model Checking, and Abstract Interpretation*, D. DSouza, A. Lal, and K. Larsen, Eds., vol. 8931 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2015, pp. 61–79.

[7] Cortesi, A., and Zanioli, M. Widening and narrowing operators for abstract interpretation. *Computer Languages, Systems & Structures* (2011).

[8] Cousot, P., and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Los Angeles, California, 1977), ACM Press, New York, NY, pp. 238–252.

[9] Cousot, P., and Cousot, R. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming* (London, UK, 1992), Springer-Verlag, pp. 269–295.

[10] Denning, D. E. A lattice model of secure information flow. *Communications of the ACM 19* (1976), 236–243.

[11] Denning, D. E. A lattice model of secure information flow. *Commun. ACM 19*, 5 (May 1976), 236–243.

[12] Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst. 32*, 2 (June 2014), 5:1–5:29.

[13] Hammer, C., and Snelting, G. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security 8* (2009), 399–422.

[14] Hornyack, P., Han, S., Jung, J., Schechter, S., and Wetherall, D. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2011), CCS '11, ACM, pp. 639–652.

[15] Lowe, G. Quantifying information flow. In *In Proc. IEEE Computer Security Foundations Workshop* (2002), pp. 18–31.

[16] Mccamant, S., and Ernst, M. D. Quantitative information-flow tracking for c and related languages. Tech. rep., 2006.

[17] Mccamant, S., and Ernst, M. D. A simulation-based proof technique fordynamic information flow, 2007.

[18] McCamant, S., and Ernst, M. D. Quantitative information flow as network flow capacity. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2008), PLDI '08, ACM, pp. 193–205.

[19] McCamant, S., and Ernst, M. D. Quantitative information flow as network flow capacity. *SIGPLAN Not. 43*, 6 (June 2008), 193–205.

[20] Rasthofer, S., Arzt, S., Lovat, E., and Bodden, E. Droidforce: Enforcing complex, data-centric, system-wide policies in android. In *Proceedings of the 9th International Conference on Availability, Reliability and Security (ARES)* (Sept. 2014), IEEE. to appear.

[21] Sabelfeld, A., and Myers, A. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on 21*, 1 (Jan 2003), 5–19.

[22] Sabelfeld, A., and Sands, D. Declassification: Dimensions and principles. *J. Comput. Secur. 17*, 5 (Oct. 2009), 517–548.

[23] Sridharan, M., Artzi, S., Pistoia, M., Guarnieri, S., Tripp, O., and Berg, R. F4f: taint analysis of framework-based web applications. In *OOPSLA* (2011), ACM.

[24] Tripp, O., Ferrara, P., and Pistoia, M. Hybrid security analysis of web javascript code via dynamic partial evaluation. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2014), ISSTA 2014, ACM, pp. 49–59.

[25] Tripp, O., Pistoia, M., Fink, S. J., Sridharan, M., and Weisman, O. Taj: effective taint analysis of web applications. In *PLDI* (2009), ACM.