HAL
archives-ouvertes.fr

Privacy Analysis of Android Apps: Implicit Flows and Quantitative Analysis

Gianluca Barbon, Agostino Cortesi, Pietro Ferrara, Marco Pistoia, Omer Tripp

▶ **To cite this version:**

Gianluca Barbon, Agostino Cortesi, Pietro Ferrara, Marco Pistoia, Omer Tripp. Privacy Analysis of Android Apps: Implicit Flows and Quantitative Analysis. Khalid Saeed; Wladyslaw Homenda. 14th Computer Information Systems and Industrial Management (CISIM), Sep 2015, Warsaw, Poland. Springer, Lecture Notes in Computer Science, LNCS-9339, pp.3-23, 2015, Computer Information Systems and Industrial Management. <10.1007/978-3-319-24369-6_1>. <hal-01444523>

HAL Id: hal-01444523
https://hal.inria.fr/hal-01444523

Submitted on 24 Jan 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution| 4.0 International License

Privacy Analysis of Android Apps: Implicit Flows and Quantitative Analysis

Gianluca Barbon[1], Agostino Cortesi[1],
Pietro Ferrara[2], Marco Pistoia[2], and Omer Tripp[2]

[1] Università Ca' Foscari Venezia, Italy
[2] IBM Thomas J.Watson Research Center, USA

Abstract. A static analysis is presented, based on the theory of abstract interpretation, for verifying privacy policy compliance by mobile applications. This includes instances where, for example, the application releases the user's location or device ID without authorization. It properly extends previous work on datacentric semantics for verification of privacy policy compliance by mobile applications by (i) tracking implicit information flow, and (ii) performing a quantitative analysis of information leakage. This yields to a novel combination of qualitative and quantitative analyses of information flows in mobile applications.

1 Introduction

Security threats are increasing in the mobile space, in particular in the Android environment. Specifically, mobile devices contain different sorts of confidential information that software might access. Such information is usually protected by permissions. However, the solutions provided by current mobile operating systems are not satisfying, and expose the user to various threats [8]. In addition, various applications and (e.g., analytics and advertisement) libraries make use of and sometimes leak user confidential data. Mobile security is also a major concern in an enterprise environment, where firms allow the use of company applications in the employee's personal device, increasing the risk of leakage of confidential business data. Therefore, there is and increasing request and need to formally verify the behavior of mobile applications, and to assess (and possibly limit) the quantity of released data. On the opposite side, a complete absence of data leakage of data would compromise the functionalities of mobile software. For instance, a navigation app like Waze needs to access the user location and communicate it to its servers in order to show appropriate traffic information. However, the user might want to prevent to leakage of her location to the advertisement engine. Ideally, we would like to impose — via suitable privacy policies — constraints on levels of data release, and give the user better awareness of the direct and indirect actual information flow concerning her personal data.

In this context, current research follows two main approaches: a statistical one [14, 17] and a language based one (e.g., information flow taint analysis) [1,

11, 21–24]. Both approaches suffer some weaknesses: the former does not fit well for qualitative analysis, while the latter as it is too strict, due to the fact that the non-interference notion [7] yields too many false positives limiting the effectiveness of the analysis.

In this paper, we extend our previous work [4]. This includes two primary contributions: (i) we investigate also implicit flows, where previously we considered only direct information release paths, and (ii) we relate explicit and implicit information flow to a quantitative notion of information leakage. We formalize our approach in the abstract interpretation framework. The advantage of such a method is that it enabes a general abstraction of all possible executions of a given program. Therefore, following the abstract-interpretation framework [6], we design an enhanced concrete semantics that formalizes how the expressions generated by the program's execution maintain footprints of (possibly confidential) data stored in the local data-store of the mobile device. With this formalization of the concrete semantics, we show how to create a sound abstraction such that the analysis is computable.

This work leads to the definition of a framework that merges quantitative and qualitative approaches by taking advantage of their 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.

The paper is structured as follows. After a brief introduction that describes related research and fundamental notions in Section 2, Section 3 recalls the semantics introduced by Cortesi et al. [4] extending it to capture implicit flows as well. Section 4 introduces the new quantitative approach that is added to the semantics described in the previous Sections. Section 5 introduces an abstraction of the quantitative analysis. Finally, in Section 6 a few significant examples of real working applications are presented and analyzed. Section 7 concludes.

2 Background

This Section introduces some important notions that will be used throughout the rest of the paper, and briefly describes the current related research.

Implicit Flows. Implicit flows were described by Denning [7] in 1976. Implicit flows 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;
```

Even if the final value of x does not allows to directly recover the value of b, the latter affects the value of x, and an indirect information flow occurs from b to x. Of course, implicit flows may yield to malicious effects[19].

Quantitative Approaches. A quantitative approach tracks some quantity, or measure, of leaked information. In [17] a new technique is proposed for determining the quantity of sensitive information that is revealed to public. The main idea presented by the authors consists in computing a maximum flow between the program inputs and the outputs, and by setting a limit on the maximum quantity of information revealed. The information flow is measured using 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. This method requires a dynamic approach in order to construct the graph, by performing multiple runs of the target program.

Quantitative value expressed as bits. McCamant et al. [17] introduced a quantity concept in order to measure bits of information that can be released by the observation of a specific execution of a program. One of the first attempts to quantify information flow is the one of Lowe [14]. 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* of quantity 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 by Clark et al.[2, 3]. 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.

Security in Mobile Environments. Nowadays smartphones are used to store, modify and collect private and confidential data, e.g. location data or phone identifier. At the same time, a lot of malicious applications able to stole data or to track users exist. Mobile operating systems are not able to grant to users an appropriate control over confidential data and on how applications manage such data [8]. These limits make these platforms a potential target for attackers. An evolution of threats in mobile environments has been stressed also by the MacAfee Labs Threats Report [15]; in particular it underlines the existence of untrusted marketplaces and the increasing diffusion of open-source and commercial mobile malware source code, that facilitate the creation of such threats by unskilled attackers. Among mobile operating systems, Android is currently the most prevalent one [13], thus becoming the target of various threats. This mobile environment present different vulnerabilities. First, there is a lack of common definitions for securities and a high volume of available applications that guarantees the diffusion of malicious programs [9]. Second, many applications make use of private information, like the `IMEI` (International Mobile Station Equipment Identity), and of advertising and analytic libraries, that sends user data to remote servers for profiling. Third, the opportunity

to install also applications coming from untrusted marketplaces makes the verification of these applications harder [18]. Last, but not least, even if Android requests to grant permission in order to install an application, this kind of control is not sufficient to avoid undesirable behaviour, because restrictions are not fine-grained [10, 12].

Confidentiality Analysis in Mobile Environments. The importance of confidentiality analysis is growing in recent years, especially in the mobile space. In this field two main approaches can be found: 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 [1, 18]. In particular, the work of Enck et al. [8] presents TaintDroid, a tool that monitors sensitive data by using real time tracking, avoiding the needing to get access to the applications source code. The main idea consists in tracking sensitive data that flows through systems interfaces, used by applications to get access to local data. This approach has some limitations. For instance, it does not allow the tracking of control flows, and generates false positives. Another approach is the one of Hornyack et al. with AppFence [12], which imposes privacy controls by retrofitting the Android environment, without the need to modify applications. Yet another approach, by Tripp and Rubin [25], is to reason about information release in terms of data values, rather than data flow, where the judgment is based on value similarity measures fed into a Bayesian learning algorithm. However, dynamic approaches present some weaknesses. First, they fail to discover some malicious behaviour, because applications have learned to recognize analysis during execution [10]. Second, the majority of dynamic approaches uses coarse-grained approximations that lead to false alarms and also missed leaks [1], while on the contrary static ones are able to discover potential leaks before the execution of the analysed application.

3 Collecting Semantics

In this section we introduce the collecting semantics, that consists in the first fundamental step of our framework design. We define the syntax, the domains, and the semantics with a specific focus on control statements.

3.1 Syntax

The formalization is focused onthree types of data: strings ($s \in \mathbb{S}$), integers ($n \in \mathbb{Z}$) and Booleans ($b \in \mathbb{B}$). *sexp*, *nexp*, and *bexp* denotes string, integer, and Boolean expressions, respectively. ℓ is used to represent data-store entries, and *lexp* denotes label expressions. For instance, 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 substring of *sexp* between n_1 and n_2, *hash* the computation of the hash value, and *read* the function that returns the value in the data-store that corresponds to the given label.

3.2 Domain

By *adexp* we denote an atomic data expression that tracks the data sources of a specific value. Formally, an atomic data expression *adexp* is a set of elements $\langle \ell_i, \{(op_j, l'_j) : j \in J\}\rangle$, representing that the datum corresponding to label ℓ_i has been combined with data corresponding to labels ℓ'_j through operators op_j to get the actual value of the expression.

The set of atomic data expressions is defined by: $\mathbb{D} = \{\langle \ell_i,\ L_i\rangle :\ i \in I \subseteq \mathbb{N}, \ell_i \in \texttt{Lab}, L_i \subseteq \wp(\texttt{Op} \times \texttt{Lab})\}$, where `Lab` is the set of labels, and `Op` is the set of operators.

An environment relates variables to their values as well as to their atomic data expression. Formally, $\Sigma = D \times V$, where (i) D : **Var** $\longrightarrow \wp(\mathbb{D})$ maps local variables in **Var** to a corresponding *adexp*, and (ii) V : **Var** $\longrightarrow (\mathbb{Z} \cup \mathbb{S})$ is the usual environment that tracks value information. The special symbol $\star$ represents data coming from the user input and from the constants of the program. Instead, data coming from the *concrete data-store* C are represented by= $\{\langle \ell_i,\ \emptyset\rangle :\ i \in I\} \subseteq \mathbb{D}$ *such that* $\forall i, j \in I : i \neq j \Rightarrow \ell_i \neq \ell_j$, *and* $\ell_i \neq \star$.

3.3 Collecting Semantics

We extend the notion of atomic data expressions to collect also implicit flows generated by *if* and *while* statements. Such flows are treated in the same way as explicit flows by collecting the Boolean expression (*bexp*) of a conditional or loop statement, and considering it as an *adexp* with its operators and sources.

Definition 1 (Extended Atomic Data Expressions). *We redefine the set of atomic data expressions as:* $\mathbb{D} = \left\langle \{\langle \ell_i, L_i\rangle : i \in I\}, \{\langle \ell_j, L_j\rangle : j \in J\}\right\rangle$ *where* $L \subseteq \wp(\texttt{Op} \times \texttt{Lab})$.

An extended atomic data expression can be seen as a pair of two atomic data expressions, where the second one refers to the implicit flows (notice that also a Boolean or relational operator may appear). Formally, $d = \langle d^e,\ d^i\rangle$, where d^e and d^i correspond to the explicit and implicit flows, respectively. In this way we collect also the Boolean operators.

Consider the conditional statement: **if** *exp* **then** $x = case_1$ **else** $x = case_2$. We can interpret each expression as a combination of explicit and implicit flow:

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

Notice that d^i in $case_1$ and $case_2$ expresses only the implicit flow generated inside the two branches of the `if` statement, and it does not include the implicit flow that comes from the evaluation of the Boolean expression `exp`. Then, if the $case_1$ is chosen, we obtain: $d_{result} = \left\langle d^e_{case_1},\ \{d^i_{case_1} \cup d^e_{if} \cup d^i_{if}\}\right\rangle$ where d^e_{if} and d^i_{if} represent the flows in the `if` condition, both collected in the implicit flow of all the subsequent expressions. The value associated to x after the `if-then-else` statement makes explicit that x has implicit dependence on the sources of the

Boolean expression. For instance, if $exp = y \geq 0$, it will track that the value of x is dependent on the value of y.

We denote by $S_N : nexp \times V \to \mathbb{Z}$, $S_S : sexp \times V \to \mathbb{S}$, and $S_B : bexp \times V \to \{\mathsf{true}, \mathsf{false}\}$ the standard concrete evaluations of numerical, string, and Boolean expressions. In addition, $S_L : lexp \times \Sigma \to \mathtt{Lab}$ that returns a data label given a label expression. The semantics of expressions on atomic data $S_A : sexp \times \Sigma \to \wp(\mathbb{D})$ is described in Figure 1. The semantics has been improved w.r.t. [4] with a new operator, $checkpwd(sexp_1, sexp_2)$, that returns true if a secret password is equal to a given value. In addition, we track both explicit and implicit flows. In particular, we do not create any implicit flow, and we simply propagate the implicit flows generated by previous expressions: $\left\langle S[\![c]\!](a,v),\ \{\langle \ell_j, L_j\rangle : j \in J\}\right\rangle$.

Similarly, we rewrite the semantics of statements that create implicit flow, that is, the semantics of *if* and *while* statements (Fig. 2). The definition of this semantics is split into explicit (S_e) and implicit (S_i) flows. We add the *skip* statement to handle the exit from a loop statement like the *while*.

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

Fig. 1: Semantics of Expressions on Atomic Data

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

Fig. 2: Concrete Semantics of Statements

Example Suppose that y contains a value arising from a data store labeled ℓ_1, while x contains user input. We assume that $> y$.

```
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 expressions computed by the collecting semantics, where the subscript represents the code line of the expression:

$$
\begin{aligned}
\mathrm{y}_1 &: \Big\langle \{\langle \ell_1, \emptyset\rangle\}, \emptyset \Big\rangle \\
(\mathrm{x<=y})_4 &: \Big\langle \{\langle \ell_1, \{(>=, \star)\}\rangle\}, \emptyset \Big\rangle \\
\mathrm{y+3}_7 &: \Big\langle \{\langle \ell_1, \{(+, \star)\}\rangle\}, \emptyset \Big\rangle \\
\mathrm{z=y+3}_7 &: \Big\langle \underbrace{\{\langle \ell_1, \{(+, \star)\}\rangle\}}_{\textbf{explicit flow}}, \underbrace{\{\langle \ell_1, \{(<, \star)\}\rangle\}}_{\textbf{implicit flow}} \Big\rangle \\
\mathrm{x=x+z}_8 &: \Big\langle \{\langle \ell_1, \{(+, \star)\}\rangle\}, \{\langle \ell_1, \{(<, \star)\}\rangle\} \Big\rangle
\end{aligned}
$$

The first three expressions are not affected by implicit flows, so the implicit component in these expressions is $\emptyset$. The second expression refers to the `if` Boolean expression, and the fourth one refers to the assignment of the *else* branch, and it takes into account the implicit flow generated by the `if` statement.

4 Quantitative Semantics

In this Section, we extend the collecting semantics by introducing a quantitative notion of information flow.

4.1 Quantitative Concrete Domain

We begin by representing values having a binary form to express quantities of information flows. In this way, we adopt a standardized evaluation of quantities coming from different data types.

Definition 2 (Label Dimension). *Let ℓ_i be the label of a location in the data-store. ω returns the size of the memory location corresponding to the given label, and it is defined by $nbit_{\ell_i} := \omega(\ell_i)$, where nbit is the retrieved dimension in bits.*

The value returned ω depends on the particular type of the datum:

– *numbers:* for the sake of 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:* we adopt a simplified representation of characters. In particular, we consider an encoding representing only English alphabet (with uppercase and lowercase letters) and numerical digits. We then have 26+26+10 elements. Thus this encoding requires 6 bits for each character, and $nbits = 6 \times l_{str}$ where l_{str} represents the number of characters of string *str*, and
– *Boolean:* such values can be only 0 or 1, so they require only 1 bit.

We now extend the collecting semantics to take into account quantities of information by adding a new expression associated to the extended *adexp* to the concrete state.

Definition 3 (Quantitative Expression). *We define a quantitative expression* qadexp *as a sequence of pairs of labels associated with quantitative values* $\langle \ell,\ q \rangle$*. This collects the quantity of information generated by implicit flows for the given label* ℓ*. This sequence is combined with a* d_q *in* qadexp *in a unique expression as follows:*

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

Therefore, we represent data expressions as follows:

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

where ℓ_k *are the labels used in statements that generate implicit flow, while* q_k *is the associated quantity of information.*

Every single pair $\langle \ell_k, q_k \rangle$ tracks the quantity of information that label ℓ_k potentially released through 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, and in particular (i) the first two components expression comes from a *qualitative* approach to explicit and implicit flows, and (ii) the last one is the result of a *quantitative* approach.

We define as Q the domain of quantities of information. We are now in position to introduce a function that describes how quantities are collected.

Definition 4 (Quantitative function ϕ). *Let* ϕ *be a function that updates 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.*

Quantities are represented as an interval $[val, val]$ were higher and lower bound coincide. This will allow an easier lift to the abstract value. Anyway, for the sake of readability, the singleton interval $[val, val]$ will be denoted by a single value *val*.

We have to track quantities of implicit flow generated by `if` and `while` statements. Conditional expressions can result only in *true* and *false*. Thus, the information obtained from the evaluation of a Boolean condition consists only of one *bit* [16].

For the sake of simplicity, we consider only the > and < (strict) operators. This avoids problems with equalities ($a == b$) allowing the collection of only one bit of information for each `if` statement [2, 3]. Figure 3 defines the semantics of conditional expressions. Notice that we do not yet introduce quantities, while we only express how to collect equality in conditional statements using integers instead of Boolean values..

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

Fig. 3: Concrete Semantics of Conditional Expressions

4.2 Quantitative Concrete Semantics

Let φ be function that maps variables to quantities ($\varphi : \texttt{Var} \rightarrow qadexp$). We now introduce the quantity notion into our quantitative collecting semantics. The initial quantity value is $\phi_{stm}(\ell_i) = 0 \ \forall i$, and it is modified by ϕ_{stm} only for the statements that generate implicit flow. This means that in the other expressions the φ component will be carried as is. The new semantics is defined in Figures 4, 5 and 6.

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

Fig. 4: Quantitative Semantics of Expressions on Atomic Data

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

Fig. 5: 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:*

  $\quad$ *let* $(a',\varphi',v') = \langle S_e[\![c_2]\!](a,\varphi,v),$
  $\quad\quad S_i[\![c_2]\!](a,\varphi,v) \cup S_e[\![c_1]\!](a,\varphi,v) \cup S_i[\![c_1]\!](a,\varphi,v)\rangle$
  $\quad$ *in* $(a',\varphi'[(\ell_i,\ q_i)/(\ell_i,\ q_i + \phi_{stm}(c_1)) : \ell_i \in \mathtt{src}(c_1)], v')$

- *otherwise:*

  $\quad$ *let* $(a',\varphi',v') = \langle S_e[\![c_3]\!](a,\varphi,v),$
  $\quad\quad S_i[\![c_3]\!](a,\varphi,v) \cup S_e[\![\neg c_1]\!](a,\varphi,v) \cup S_i[\![c_1]\!](a,\varphi,v)\rangle$
  $\quad$ *in* $(a',\varphi'[(\ell_i,\ q_i)/(\ell_i,\ q_i + \phi_{stm}(c_1)) : \ell_i \in \mathtt{src}(c_1)], v')$

where ϕ_{stm} is the quantitative function in Def. 4

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

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

Fig. 6: Quantitative Concrete Semantics of Control Statements

5 Abstract Semantics

We now extend the abstract semantics proposed by Cortesi et al. [4] to implicit flows and a quantitative analysis. Our abstract semantics is parameterized by a value domain V^a, and a label abstraction. First of all, we need to define a computable abstraction of quantities.

Definition 5 (Quantity Value Abstraction). *The quantity associate with label expressions is an interval. Each label* ℓ^a *is associated to an interval of quantities where the lower and the upper bounds are the minimum and the maximum quantities of information that can be released through the implicit flow for that specific label. Therefore, the abstract* qadexp *is defined as* $\langle \ell^a_k, q^a_{k_{min}}, q^a_{k_{max}} \rangle$.

If we have unbounded quantities, the analysis reveals a complete leakage of the associated label. In this case, the upper bound of the intervals is unbounded. Instead, for the lower bound the minimum quantity is zero.

5.1 Atomic Data Abstraction Extension

We now define the atomic data abstraction extended for handling implicit flows and quantities.

Definition 6 (Abstract Extended Atomic Data and Quantities). *Let us consider a set of atomic data and quantity values. We define abstract elements as*

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

where:

- ℓ^a_w *is an element that abstracts labels in* `Lab` *to track explicit flow,*
- ℓ^a_z *and* ℓ^a_g *are elements that abstract labels in* `Lab` *to track 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* ℓ^a_w *and* ℓ^a_z *with labels abstracted by* ℓ^a_{iw} *and* ℓ^a_{jz}*, and track explicit and implicit flows, 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* ℓ^a_w *and* ℓ^a_z *with labels abstracted by* ℓ^a_{iw} *and* ℓ^a_{jz}*, and track explicit and implicit flows, 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 quantites associated to a* ℓ^a_g *element,*
- $q^{a\sqcap}_{kg} : k \in J$ *is an under-approximation of the interval of possible quantities of information associated to* ℓ^a_g *with values represented by* q^a_{kg}*,*
- $q^{a\sqcup}_{kg} : k \in J'$ *is an over-approximation of the interval of possible quantities of information associated to* ℓ^a_g *with values represented by* q^a_{kg}*, and*
- $q^{a\sqcap}_g \leqslant q^{a\sqcup}_g$*.*

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 $\mathtt{src}(d)$.

Although we inherit the abstraction and concretization functions for the explicit flows [4], we have to extend them to handle quantities.

Definition 7 (Quantitative Abstraction function). *We denote by* α_{Q} *the abstraction function that given a set* $\{(\ell_k, q_k) : k \in J\}$ *returns* $\{(\alpha_{\mathtt{Lab}}(\ell_k), q^{a\sqcap}_k, q^{a\sqcup}_k) : k \in J\}$*, where* $q^{a\sqcap}_k, q^{a\sqcup}_k$ *represent the bounds of the interval that approximates all possible quantitative values in the abstract domain* $\mathcal{Q}^a$*.*

Definition 8 (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:*

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

The abstraction function is then extended to sets by computing the upper bound of the point-wise application of α_s *to all the elements of the given set.*

5.2 Abstract Semantics of Statements

Expressions are abstracted via an abstract data label and an abstract value (AD^a and V^a, respectively). In Figure 7 the abstract semantics of statements taking into account implicit flows is defined Then, in Figure 8 this semantics is extended with the quantitative dimension. We omit here the abstract semantics of expressions, as it does not generate any implicit flow, and we refer the interested reader to Cortesi et. al [4] for more details.

$$
\begin{aligned}
S^a[\![x := sexp]\!](a^a, v^a) &= (a^a, S^a_v[\![x := sexp]\!](v^a)) \\
S^a[\![skip]\!](a^a, v^a) &= (a^a, v^a) \\
S^a[\![send(sexp)]\!](a^a, v^a) &= (a^a, v^a) \\
S^a[\![c_1; c_2]\!](a^a, v^a) &= S^a[\![c_2]\!](S^a[\![c_1]\!](a^a, v^a)) \\
S^a[\![\text{if } c_1 \text{ then } c_2 \text{ else } c_3]\!](a^a, v^a) &= \\
&\Big\langle S^a_e[\![c_2]\!](a^a, S^a_e[\![c_1]\!](v^a)) \sqcup S^a_e[\![c_3]\!](a^a, S^a_e[\![\neg c_1]\!](v^a)), \\
&\quad S^a_i[\![c_2]\!](a^a, S^a_e[\![c_1]\!](v^a)) \sqcup S^a_e[\![c_1]\!](a^a, v^a) \sqcup \\
&\quad S^a_i[\![c_1]\!](a^a, v^a) \sqcup S^a_i[\![c_3]\!](a^a, S^a_e[\![\neg c_1]\!](v^a)) \sqcup \\
&\quad S^a_e[\![\neg c_1]\!](a^a, v^a) \sqcup S^a_i[\![c_1]\!](a^a, v^a)\Big\rangle \\
S^a[\![\text{while } c_1 \text{ do } c_2]\!](a, v) &= \\
&fix(S^a[\![\text{ if } (c_1) \text{ then } (c_2; \text{while } c_1 \text{ do } c_2) \text{ else } skip]\!](a^a, v^a))
\end{aligned}
$$

Fig. 7: Abstract Semantics of Statements

$$
\begin{aligned}
S^a[\![x := sexp]\!](a^a, \varphi^a, v^a) &= (a^a[x \mapsto S^a_A[\![sexp]\!](a^a, \varphi^a, v^a)], v^a[x \mapsto S^a_S[\![sexp]\!](v^a)]) \\
S^a[\![skip]\!](a^a, \varphi^a, v^a) &= (a^a, \varphi^a, v^a) \\
S^a[\![send(sexp)]\!](a^a, \varphi^a, v^a) &= (a^a, \varphi^a, v^a) \\
S^a[\![c_1; c_2]\!](a^a, \varphi^a, v^a) &= S^a[\![c_2]\!](S^a[\![c_1]\!](a^a, \varphi^a, v^a))
\end{aligned}
$$

$$
\begin{aligned}
&S^a[\![\text{if } c_1 \text{ then } c_2 \text{ else } c_3]\!](a^a, \varphi^a, v^a) = \\
&\quad let\ (a'^a, \varphi'^a, v'^a) = \Big\langle S^a_e[\![c_2]\!](a^a, \varphi^a, S^a_e[\![c_1]\!](v^a)) \sqcup S^a_e[\![c_3]\!](a^a, \varphi^a, S^a_e[\![\neg c_1]\!](v^a)), \\
&\qquad\qquad S^a_i[\![c_2]\!](a^a, \varphi^a, S^a_e[\![c_1]\!](v^a)) \sqcup S^a_e[\![c_1]\!](a^a, \varphi^a, v^a) \sqcup \\
&\qquad\qquad S^a_i[\![c_1]\!](a^a, \varphi^a, v^a) \sqcup S^a_i[\![c_3]\!](a^a, \varphi^a, S^a_e[\![\neg c_1]\!](v^a)) \sqcup \\
&\qquad\qquad S^a_e[\![\neg c_1]\!](a^a, \varphi^a, v^a) \sqcup S^a_i[\![c_1]\!](a^a, \varphi^a, v^a)\Big\rangle \\
&\quad in\ (a'^a, \varphi'^a[(\ell^a_i, q^{a\sqcap}_i, q^{a\sqcup}_i)/(\ell^a_i, q^{a\sqcap}_i + \phi^{a\sqcap}_{stm}(c_1), q^{a\sqcup}_i + \phi^{a\sqcup}_{stm}(c_1)) : \ell^a_i \in \texttt{src}(c_1)], v'^a)
\end{aligned}
$$

where ϕ_{stm} is the quantitative function in Def. 4

$$
\begin{aligned}
&S^a[\![\text{while } c_1 \text{ do } c_2]\!](a^a, \varphi^a, v^a) = \\
&\quad fix(S^a[\![\text{ if } (c_1) \text{ then } (c_2; \text{while } c_1 \text{ do } c_2) \text{ else } skip]\!](a^a, \varphi^a, v^a))
\end{aligned}
$$

Fig. 8: Quantitative Abstract Semantics of Statements

We need to abstract the number of iterations of a while loop to precisely approximate the quantity of information leaked by a loop. Our approach is composed by two steps: a *while* interval analysis approximating the number of iterations, followed by an *extended adexp* collection with quantitative values.

Step (a): `while` interval analysis We add a counter initialized to 0 at the beginning, and we increment it by one at each loop iteration. In this way, we can apply a standard interval analysis to infer an upper bound on the number of iterations of a loop.
Example Consider the following program, where i represents the counter we added:

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

At the end of the analysis, the interval domain can infer $i \mapsto [0..15]$ (e.g., by applying a narrowing operator after widening [5]). The upper bound of the interval of variable i returns the number of iterations of the loop, that is, 15. Then the upper bound on the number of iterations 15 becomes $[1, 4]$, where 4 are the number of bits leaked in 15 iterations.

Step (b): *extended adexp* collection with quantitative value At the end of the while interval analysis, we apply a quantitative value analysis.
Example Consider the following example:

```
1 secret = read(...)
2 found = false
3 while (!found) {
4   pwd = user_input()
5   found = checkpwd(pwd, secret)}
```

At the end of the first iteration of the analysis of the loop, we obtain:

$$\begin{aligned}
\text{secret}_1 &: \left\langle \left\{ \langle \ell_1,\ \emptyset,\ \emptyset \rangle \right\},\ \emptyset \right\rangle \\
\text{found}_2 &: \left\langle \emptyset,\ \emptyset \right\rangle \\
!\text{found}_3 &: \left\langle \emptyset,\ \emptyset \right\rangle \\
\text{pwd}_4 &: \left\langle \emptyset,\ \emptyset \right\rangle \\
\text{found}_5 &: \left\langle \left\{ \langle \ell_1,\ \{(\mathit{checkpwd}, \star)\},\ \{(\mathit{checkpwd}, \star)\} \rangle \right\},\ \emptyset \right\rangle
\end{aligned}$$

Notice that no quantitative information has yet been released. In fact, the Boolean condition of the while loop is checked against a constant value (*false*)

during the first iteration. However, *found* might be still false, and the loop would be iterated another time. Then, starting from 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:

$$\left\langle \left\{ \ldots \textit{explicit flow} \ldots \right\},\ \left\{ \langle \ell_1,\ \{(\textit{checkpwd}, \star)\},\ \{(\textit{checkpwd}, \star), (<, \star), (>, \star)\} \rangle \right\} \right\rangle$$

Notice that the function *checkpw(p_1,p_2)* returns a Boolean value, so *one bit*. This means that we are accumulating a bit of information at each iteration.

As soon as the implicit flow comes into the picture, we have to consider the quantity interval computed in *step (a)*:

$$\left\{ \left\langle \left\{ \ldots \textit{explicit flow} \ldots \right\},\ \left\{ \langle \ell_1,\ \{(\textit{checkpwd}, \star)\},\ \{(\textit{checkpwd}, \star), (<, \star), (>, \star)\} \rangle \right\} \right\rangle,\ \langle \ell_1,\ 1,\ 4 \rangle \right\}$$

If inside a while loop there is an obfuscating operator (e.g., encryption or hashing) applied to confidential data, we need to know the quantity of information that is released by the operator. For instance, in the previous example the operator `checkpwd(`p_1`,`p_2`)` checks if the password given by the user is correct, and it returns a Boolean value. Thus the analysis accumulates a single bit at each iteration, and the quantitative value will depend on the number of iterations. However, other operators might release more information. In this case, we compute the product of the number of iterations and the released bits.

6 Applications

In this Section, we discuss the results of our analysis of some examples listed in the DroidBench application set [20], created by the Secure Software Engineering group of the Technische Universität Darmstadt. This set is open source, and it is a standard benchmark to test static and dynamic analyses. We chose the examples that specifically deal with implicit flows.
An interesting comparison can be made with the work by Tripp and Rubin [25], as they also used DroidBench as testing environment. Their approach performs very well on the whole test set (also with respect to other tools, like TaintDroid), but it suffers from false negatives in case of implicit flows. Our analysis instead allows to cope with these particular cases, for instance the ones due to custom transformations of private data in the ImplicitFlow1 test program. This is because we adopted a different approach, that instead of looking for privacy sinks, observes the whole flow of confidential data and operations applied to them.
For the sake of readability, we simplified some library functions, and we added some semantic rules to support some primitive functions contained in these examples, and that were not part of the minimal language we adopted in our formalization. For each example, we describe the results of the collecting semantics on a particular concrete state, and we then perform a two-stages static analysis. First we illustrate the results without the quantitative component *qadexp*, and we then discuss the results of the quantitative semantics.

6.1 ImplicitFlow1

The first example is an application that reads the device identifier (`IMEI`), obfuscates it, and then leaks it. The obfuscation can be performed with two functions with two different obfuscation powers (namely, low and high).

```
1  public class ImplicitFlow1 extends Activity {
2
3    protected void onCreate(...) {
4      // ...
5
6      String imei = getDeviceId(); //device id
7      String obfuscatedIMEI
8        = obfuscateIMEI(imei);
9      writeToLog(obfuscatedIMEI);
10
11     obfuscatedIMEI
12       = hardObfIMEI(imei);
13     writeToLog(obfuscatedIMEI);
14   }
15
16   private String obfuscateIMEI(String imei){
17     String result = "";
18     char[] imeiAsChar = imei.toCharArray();
19     int len = imeiAsChar.length();
20     int i = 0;
21     int shift = 49;
22
23     while (i < len){
24       result += (char)
25         (((int)imeiAsChar[i]) + shift );
26       //returns 'a' for '0', 'b' for '1', ...
27       i++;
28     }
29     return result;
30   }
31   private String hardObfIMEI(String imei){
32     char[] imeiAsChar = imei.toCharArray();
33     String result = "";
34     int len = imeiAsChar.length();
35     int i = 0;
36
37     while (i < len){
38       result += (char)
39         (((int)imeiAsChar[i]) +1 + (i*i)%62);
40       i++;
41     }
42
43     return result;
44   }
45
46   private void writeToLog(String message){
47     Log.i("INFO", message); //sink
48   }
49 }
```

First of all, we extend the semantics to the new functions contained in this example. For the most part, this extension is very intuitive and straightforward.

$$
\begin{aligned}
S_A[\![getDeviceId()]\!](a,v) &= \{\langle S_L[\![lexp]\!](a,v), \emptyset\rangle\} \\
S_A[\![toCharArray(sexp)]\!](a,v) &= \{\langle \ell_1, L_1 \cup \{(toCharArray, \ell_1)\}\rangle : \langle \ell_1, L_1\rangle \in S_A[\![sexp_1]\!](a,v)\} \\
S_A[\![length(sexp)]\!](a,v) &= \{\langle \ell_1, L_1 \cup \{(length, \ell_1)\}\rangle : \langle \ell_1, L_1\rangle \in S_A[\![sexp_1]\!](a,v)\} \\
S_A[\![Log(sexp)]\!](a,v) &= (a,v) \\
S_A[\![(typecast)sexp]\!](a,v) &= \{\langle \ell_1, L_1 \cup \{(cast(type), \ell_1)\}\rangle : \langle \ell_1, L_1\rangle \in S_A[\![sexp_1]\!](a,v)\}
\end{aligned}
$$

getDeviceId returns the `IMEI` from the datastore, *toCharArray* convert the label to a char, *length* returns the dimension (in integer) of an array and *Log* writes the argument to a log file. $cast(type)$ represents the type casting. We also add the modulo % operator to the semantics. We collect it as *mod* in our domain, and its behavior is similar to other arithmetic operators. As for arrays, when we refer 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.

Concrete Analysis The device identifier `IMEI` is contained in the data-store and can be retrieved through *getDeviceId*, 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 value will be equal to the dimension of the `IMEI`, that is, 14.

$$
\begin{array}{rl}
\text{imei}_{6} & : \Big\langle \Big\{ \langle \ell_1,\ \emptyset \rangle \Big\},\ \emptyset \Big\rangle \\
\text{imeiAsChar}_{18} & : \Big\langle \Big\{ \langle \ell_1,\ \{(\mathit{toCharArray}, \ell_1)\} \rangle \Big\},\ \emptyset \Big\rangle \\
\text{len}_{19} & : \Big\langle \Big\{ \langle \ell_1,\ \{(\mathit{length}, \ell_1)\} \rangle \Big\},\ \emptyset \Big\rangle \\
\mathit{while\ cond}_{23} & : \Big\langle \Big\{ \langle \ell_1, \{(\mathit{length}, \ell_1), (>, \star)\} \rangle \Big\},\ \emptyset \Big\rangle \quad (\mathit{count}_1 = 0) \\
\text{result}_{24} & : \Big\langle \Big\{ \langle \ell_1, \{([\mathit{sub}, \star, \star], \ell_1), (\mathit{cast}(\mathit{int}), \ell_1), (+, \star), (\mathit{cast}(\mathit{char}), \ell_1), (+, \star)\} \rangle \Big\}, \\
& \qquad \Big\{ \langle \ell_1, \{(\mathit{length}, \ell_1), (>, \star)\} \rangle \Big\} \Big\rangle \\
\text{i}_{27} & : \Big\langle \emptyset,\ \Big\{ \langle \ell_1, \{(\mathit{length}, \ell_1), (>, \star)\} \rangle \Big\} \Big\rangle \\
\ldots & \ \mathit{(after\ loop\ exit\ the\ condition\ in\ the\ flow\ is\ inverted)}, \quad (\mathit{count}_1 = 14) \\
\mathit{log}_{47} & : \Big\langle \Big\{ \langle \ell_1, \{([\mathit{sub}, \star, \star], \ell_1), (\mathit{cast}(\mathit{int}), \ell_1), (+, \star), (\mathit{cast}(\mathit{char}), \ell_1), (+, \star), \ldots\} \rangle \Big\}, \\
& \qquad \Big\{ \langle \ell_1, \{(\mathit{length}, \ell_1), (>, \star)(<, \star)\} \rangle \Big\} \Big\rangle \\
\text{imeiAsChar}_{32} & : \Big\langle \Big\{ \langle \ell_1,\ \{(\mathit{toCharArray}, \ell_1)\} \rangle \Big\},\ \emptyset \Big\rangle \\
\text{len}_{34} & : \Big\langle \Big\{ \langle \ell_1\ \{(\mathit{length}, \ell_1)\} \rangle \Big\},\ \emptyset \Big\rangle \\
\mathit{while\ cond}_{37} & : \Big\langle \Big\{ \langle \ell_1, \{(\mathit{length}, \ell_1), (>, \star)\} \rangle \Big\},\ \emptyset \Big\rangle \quad (\mathit{count}_2 = 0) \\
\text{result}_{38} & : \Big\langle \Big\{ \langle \ell_1, \{([\mathit{sub}, \star, \star], \ell_1), (\mathit{cast}(\mathit{int}), \ell_1), (+, \star), (\times, \star), (\mathit{mod}, \star), (\mathit{cast}(\mathit{char}), \ell_1), \\
& \qquad (+, \star)\} \rangle \Big\},\ \Big\{ \langle \ell_1, \{(\mathit{length}, \ell_1), (>, \star)\} \rangle \Big\} \Big\rangle \\
\text{i}_{40} & : \Big\langle \emptyset,\ \Big\{ \langle \ell_1, \{(\mathit{length}, \ell_1), (>, \star)\} \rangle \Big\} \Big\rangle \\
\ldots & \ \mathit{(after\ loop\ exit\ the\ condition\ in\ the\ flow\ is\ inverted)}, \quad (\mathit{count}_2 = 14) \\
\mathit{log}_{47} & : \Big\langle \Big\{ \langle \ell_1, \{([\mathit{sub}, \star, \star], \ell_1), (\mathit{cast}(\mathit{int}), \ell_1), (+, \star), (\times, \star), (\mathit{mod}, \star), (\mathit{cast}(\mathit{char}), \ell_1), \\
& \qquad (+, \star), \ldots\} \rangle \Big\},\ \Big\{ \langle \ell_1, \{(\mathit{length}, \ell_1), (>, \star)(<, \star)\} \rangle \Big\} \Big\rangle
\end{array}
$$

As we can see from the concrete analysis, there is no noticeable difference between the two functions. Indeed, both example apply these function inside a loop, and in both the conditional expressions depend on the dimension of the secret label.

Abstract Analysis The main differences between the concrete and abstract semantics consist in (i) considering when the loop condition holds and does not (unlike the concrete semantics that always knows a precise value), and (ii) the application of the interval analysis to over-approximate the number of iterations. Notice that since the dimension of the `IMEI` does not change, the corresponding label can be abstracted with full precision. The same applies to its dimension.

$$
\begin{aligned}
\text{imei}_{6} &: \Big\langle \big\{ \langle \ell_1,\ \emptyset,\ \emptyset \rangle \big\},\ \emptyset \Big\rangle \\
\text{imeiAsChar}_{18} &: \Big\langle \big\{ \langle \ell_1,\ \{(\mathit{toCharArray}, \ell_1)\},\ \{(\mathit{toCharArray}, \ell_1)\} \rangle \big\},\ \emptyset \Big\rangle \\
\text{len}_{19} &: \Big\langle \big\{ \langle \ell_1,\ \{(\mathit{length}, \ell_1)\},\ \{(\mathit{length}, \ell_1)\} \rangle \big\},\ \emptyset \Big\rangle \\
\mathit{while\ cond}_{23} &: \Big\langle \big\{ \langle \ell_1, \{(\mathit{length}, \ell_1), (>, \star)\}, \{(\mathit{length}, \ell_1), (>, \star)\} \rangle \big\},\ \emptyset \Big\rangle \\
\text{result}_{24} &: \Big\langle \big\{ \langle \ell_1, \{([\mathit{sub}, \star, \star], \ell_1), (\mathit{cast}(\mathit{int}), \ell_1), (+, \star), (\mathit{cast}(\mathit{char}), \ell_1), (+, \star)\}, \\
&\qquad \{([\mathit{sub}, \star, \star], \ell_1), (\mathit{cast}(\mathit{int}), \ell_1), (+, \star), (\mathit{cast}(\mathit{char}), \ell_1), (+, \star)\} \rangle \big\}, \\
&\qquad \big\{ \langle \ell_1, \{(\mathit{length}, \ell_1)\}, \{(\mathit{length}, \ell_1), (>, \star), (<, \star)\} \rangle \big\} \Big\rangle \\
\text{i}_{27} &: \Big\langle \emptyset,\ \big\{ \langle \ell_1, \{(\mathit{length}, \ell_1)\}, \{(\mathit{length}, \ell_1), (>, \star), (<, \star)\} \rangle \big\} \Big\rangle \\
\mathit{log}_{47} &: \Big\langle \big\{ \langle \ell_1, \{([\mathit{sub}, \star, \star], \ell_1), (\mathit{cast}(\mathit{int}), \ell_1), (+, \star), (\mathit{cast}(\mathit{char}), \ell_1), (+, \star)\}, \\
&\qquad \{([\mathit{sub}, \star, \star], \ell_1), (\mathit{cast}(\mathit{int}), \ell_1), (+, \star), (\mathit{cast}(\mathit{char}), \ell_1), (+, \star), \ldots\} \rangle \big\}, \\
&\qquad \big\{ \langle \ell_1, \{(\mathit{length}, \ell_1)\}, \{(\mathit{length}, \ell_1), (>, \star), (<, \star)\} \rangle \big\} \Big\rangle \\
\text{imeiAsChar}_{32} &: \Big\langle \big\{ \langle \ell_1,\ \{(\mathit{toCharArray}, \ell_1)\},\ \{(\mathit{toCharArray}, \ell_1)\} \rangle \big\},\ \emptyset \Big\rangle \\
\text{len}_{34} &: \Big\langle \big\{ \langle \ell_1,\ \{(\mathit{length}, \ell_1)\},\ \{(\mathit{length}, \ell_1)\} \rangle \big\},\ \emptyset \Big\rangle \\
\mathit{while\ cond}_{37} &: \Big\langle \big\{ \langle \ell_1, \{(\mathit{length}, \ell_1), (>, \star)\}, \{(\mathit{length}, \ell_1), (>, \star)\} \rangle \big\},\ \emptyset \Big\rangle \\
\text{result}_{38} &: \Big\langle \big\{ \langle \ell_1, \{\langle \ell_1, \{([\mathit{sub}, \star, \star], \ell_1), (\mathit{cast}(\mathit{int}), \ell_1), (+, \star), (\times, \star), (\mathit{mod}, \star), \\
&\qquad (\mathit{cast}(\mathit{char}), \ell_1), (+, \star)\}, \{([\mathit{sub}, \star, \star], \ell_1), (\mathit{cast}(\mathit{int}), \ell_1), (+, \star), (\times, \star), \\
&\qquad (\mathit{mod}, \star), (\mathit{cast}(\mathit{char}), \ell_1), (+, \star)\} \rangle \big\}, \\
&\qquad \big\{ \langle \ell_1, \{(\mathit{length}, \ell_1)\}, \{(\mathit{length}, \ell_1), (>, \star), (<, \star)\} \rangle \big\} \Big\rangle \\
\text{i}_{40} &: \Big\langle \emptyset,\ \big\{ \langle \ell_1, \{(\mathit{length}, \ell_1)\}, \{(\mathit{length}, \ell_1), (>, \star), (<, \star)\} \rangle \big\} \Big\rangle \\
\mathit{log}_{47} &: \Big\langle \big\{ \langle \ell_1, \{\langle \ell_1, \{([\mathit{sub}, \star, \star], \ell_1), (\mathit{cast}(\mathit{int}), \ell_1), (+, \star), (\times, \star), (\mathit{mod}, \star), \\
&\qquad (\mathit{cast}(\mathit{char}), \ell_1), (+, \star)\}, \{([\mathit{sub}, \star, \star], \ell_1), (\mathit{cast}(\mathit{int}), \ell_1), (+, \star), (\times, \star), \\
&\qquad (\mathit{mod}, \star), (\mathit{cast}(\mathit{char}), \ell_1), (+, \star), \ldots\} \rangle \big\}, \\
&\qquad \big\{ \langle \ell_1, \{(\mathit{length}, \ell_1)\}, \{(\mathit{length}, \ell_1), (>, \star), (<, \star)\} \rangle \big\} \Big\rangle
\end{aligned}
$$

Quantitative Analysis The interval analysis approximates the minimum and maximum number of iterations of the loop. Through these bounds, we compute an interval of quantities, that infers the minimum and the maximum amount of information revealed through implicit flows. However, in this particular example the number of iterations can be precisely inferred because it is performed on a fixed value (the dimension of the `IMEI`). Thus, we infer that the loop is iterated 14 times. Each iteration of the loop leaks one bit of information. We now infer the number of bits using the method described in the Section 5.2:

$$\mathit{quantity} = \lfloor \mathit{log}_2(\mathit{n\_iterations}) \rfloor + 1 \ = 4\ \mathit{bits}$$

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

6.2 ImplicitFlow2

In this example, the user has to type a password. Then, this password is compared to the correct one, that comes from the data-store. After this evaluation, a message is saved to a log file. This operation leaks information through an implicit flow, since the logged message depends on the correctness of the password.

```
1 public class ImplicitFlow2 extends Activity {
2
3   protected void onCreate(...){
4     // ...
5   }
6
7   public void checkPassword(View view){
8     String userInputPassword = //user input
9     String superSecure = //secret password
10
11    if (checkpwd(superSecure,userInputPassword))
12      passwordCorrect = true;
13    else
14      passwordCorrect = false;
15
16    if (passwordCorrect)
17      Log.i("INFO", "Password is correct");
18    else
19      Log.i("INFO", "Password is not correct");
20  }
21 }
```

Concrete Analysis We reuse the semantics of *Log* defined for the previous example ($S_A[\![Log(sexp)]\!](a,v) = (a,v)$). The abstract semantics is similar. We now shows the results of the concrete semantics, assuming that the password provided by the user is not correct. The concrete data-store contains only the label that corresponds to the variable *superSecure*, that is, $\langle \ell_1, \emptyset \rangle$.

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

Abstract Analysis We now present the results of the abstract semantics. We adopt the same label of the concrete semantics.

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

Quantitative Analysis We abstract quantities with an interval. In this case study, there is an implicit flow in the first `if` statement:

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

This value is then propagated until the *Log* statement that leaks it.

6.3 ImplicitFlow3

Like in the previous example, this example checks if a password provided by a user matches the correct password. However, in this case the information is leaked through the creation of objects.

```
1  public class ImplicitFlow3 extends Activity {
2
3    protected void onCreate(...) {
4      // ...
5    }
6
7    public void leakData(View view){
8      String userIntPwd = //user input
9      String superSecure = //secret password
10
11     Interface classTmp;
12     if (checkpwd(superSecure,userIntPwd))
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 }
```

Concrete Analysis We reuse the Log semantic described in the previous example. We assume that the password typed by the user is correct. As in the previous example, the concrete data-store contains only one label $\{\langle \ell_1, \emptyset \rangle\}$ corresponding to *superSecure*.

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

Abstract Analysis We adopt the same label abstraction.

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

Quantitative Analysis In this example, there is only one if statement that generates implicit flow. This statement exposes *1 bit* of quantity of information:

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

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

6.4 Discussion

As emphasized by the examples above, the adoption of a quantitative analysis allows the evaluation of quantities of confidential data that might be released. In ImplicitFlow1, the analysis tells that an implicit flows exists, that confidential labels are contained in it, and it also estimate the potential quantities of data released. This quantity is calculated by the operations implemented in the code, so by the operations that obfuscate the confidential label. The application of given policies [4] will allow to establish whether the released quantities are allowed or not. Any considerations about the safeness of the analyzed application are thus referred to the type of applied policy. In particular, as for ImplicitFlow1, we are able to calculate how much the while loop affects the produced quantity of data. Indeed, it allows to understand that, given a fixed (and possibly low) number of loop iterations, the quantity of confidential data that might be released will not be high. In conclusion, this analysis is not only capable of locating implicit flows, but also to evaluate their importance. In fact, if the released values will be low, with respect to a given policy, the implicit flow will be negligible.

7 Conclusions

In this paper, we extended the framework for tracking explicit flows introduced in [4] with respect to implicit flows and quantitative analysis, showing the effectiveness of this approach on significant examples. This framework can support complex hybrid policies, i.e. policies that can grant both qualitative and quantitative thresholds.

Aknowledgments Work partially supported by the Italian MIUR project "Security Horizons".

References

1. S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.*, 49(6):259–269, June 2014.
2. D. Clark, S. Hunt, and P. Malacaria. Quantitative analysis of the leakage of confidential data. *Electronic Notes in Theoretical Computer Science*, 59(3):1 – 14, November 2002. Quantitative Aspects of Programming Languages (Satellite Event for PLI 2001).
3. D. Clark, S. Hunt, and P. Malacaria. Quantified interference for a while language. In *Electr. Notes Theor. Comput. Sci. 112: 149-166*. 2005.
4. A. Cortesi, P. Ferrara, M. Pistoia, and O. Tripp. Datacentric semantics for verification of privacy policy compliance by mobile applications. In *Proc. Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, volume 8931 of *LNCS*, pages 61–79. Springer, 2015.
5. A. Cortesi and M. Zanioli. Widening and narrowing operators for abstract interpretation. *Computer Languages, Systems & Structures*, 37(1): 24-42, 2011.

6. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.
7. D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19:236–243, 1976.
8. W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst.*, 32(2):5:1–5:29, June 2014.
9. W. Enck, D. Octeau, P. Mcdaniel, and S. Chaudhuri. A study of android application security. In *In Proc. USENIX Security Symposium*, 2011.
10. C. Fritz, S. Arzt, and al. Highly precise taint analysis for android application. Technical report, EC SPRIDE Technical Report TUD-CS-2013-0113. http://www. bodden. de/pubs/TUD-CS-2013-0113. pdf, 2013.
11. C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8:399–422, 2009.
12. P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *Proc. 18th ACM Conf. on Computer and Communications Security*, pages 639–652, New York, NY, USA, 2011. ACM.
13. International Data Corporation. Worldwide Quarterly Mobile Phone Tracker 3q14. http://www.idc.com/tracker/showproductinfo.jsp?prod-id=37. Accessed: Jan. 2015.
14. G. Lowe. Quantifying information flow. In *In Proc. IEEE Computer Security Foundations Workshop*, pages 18–31, 2002.
15. McAfee Labs. Threats Report. http://www.mcafee.com/ca/resources/reports/rp-quarterly-threat-q3-2014.pdf. Accessed: Dec. 2014.
16. S. Mccamant and M. D. Ernst. A simulation-based proof technique fordynamic information flow, 2007.
17. S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. *SIGPLAN Not.*, 43(6):193–205, June 2008.
18. S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden. Droidforce: Enforcing complex, data-centric, system-wide policies in android. In *Proceedings of the 9th International Conference on Availability, Reliability and Security (ARES)*. IEEE, Sept. 2014.
19. A. Russo, A. Sabelfeld, and K. Li. Implicit flows in malicious and nonmalicious code. In *Logics and Languages for Reliability and Security*, volume 25 of *NATO Science for Peace and Security Series*, pages 301–322. IOS Press, 2010.
20. Secure Software Engineering Group - Ec Spride. DroidBench. http://sseblog.ec-spride.de/tools/droidbench/. Accessed: Feb. 2015.
21. G. Smith. Principles of secure information flow analysis. In M. Christodorescu and al., editors, *Malware Detection*, volume 27 of *Advances in Information Security*, pages 291–307. Springer, 2007.
22. M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg. F4f: taint analysis of framework-based web applications. In *OOPSLA*. ACM, 2011.
23. O. Tripp, P. Ferrara, and M. Pistoia. Hybrid security analysis of web javascript code via dynamic partial evaluation. In *Proc. of the 2014 Int. Symposium on Software Testing and Analysis*, ISSTA 2014, pages 49–59, New York, NY, USA, 2014. ACM.
24. O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. Taj: effective taint analysis of web applications. In *ACM PLDI*, pages 87–97. ACM, 2009.
25. O. Tripp and J. Rubin. A bayesian approach to privacy enforcement in smartphones. In *USENIX Security*, 2014.