Datacentric Semantics for Verification of Privacy Policy Compliance by Mobile Applications

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

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

Abstract. We introduce an enhanced information-flow analysis for tracking the amount of confidential data that is possibly released to third parties by a mobile application. The main novelty of our solution is that it can explicitly keep track of the footprint of data sources in the expressions formed and manipulated by the program, as well as of transformations over them, yielding a lazy approach with finer granularity, which may reduce false positives with respect to state-of-the-art information-flow analyses.
Keywords: Abstract Interpretation, Privacy, Information-flow Analysis.

1 Introduction

Mobile applications typically ask for permission to access personal (i.e. relevant with respect to privacy) information stored on the device. However, even in non-malicious applications, once these permissions are granted it is often the case that data concerning gender, sex, age, GPS location, smartphone ID, etc. is managed in a way that partially releases it to third parties (e.g. for advertising, profiling, analytics and social computing), with or without some degree of obfuscation, leaving the user unaware of how much confidential information actually leaked [29, 36]. Most systems, in fact, are designed to allow users to configure access control (e.g., by setting permissions), without enabling them to monitor the actual information flow of confidential data. In reality, users may trust an application to manage their personal information, but might be concerned about the obfuscation degree applied to that information before it is passed to other (possibly untrusted) actors. The key issue is to keep track of (and possibly restrict) the amount of confidential information that is released by an app, without compromising the usability of the app itself by enforcing overly conservative constraints.

In this challenging context, the aim of this work is to define a theoretical framework to support the design of tools that provide developers as well as end users with better control of how the values managed by the applications reveal confidential data stored on the device.

1.1 Background

Preserving confidentiality of sensitive information in software systems is a subject of intensive research. Various language-based information-flow security

analyses were proposed [18, 21, 32, 35]. Most of these works are based on the non-interference notion that says that a variation of confidential data given as input to a program does not cause any variation of publicly observable data [12]. The approaches are different (type systems [28, 32, 34], dependence graphs [18, 22], slicing [2, 6, 21, 35], etc.), and apply to different languages including imperative, object-oriented, functional and structured query languages [17, 18, 21, 30]. Recent works address in particular data protection from permission-hungry Android applications [4, 7, 15, 19, 24, 40], and data-leakage aggregation due to undesired inter-application dataflows [31].

However, in the scenario depicted above the crucial point is not just to discover if sensitive data is confined to private variables, but also to keep control of how the authorized access to confidential data is compliant with respect to a privacy policy, expressed in terms of minimal degree of obfuscation that should be applied to sensitive data in the exposed values. In this respect, the information-flow approach of the mentioned works yields overly conservative results, as the granularity of public/private variables is too coarse, just like the tampered/untampered granularity [1] assigned to data when declassification mechanisms are introduced for relaxing confidentiality policies [5].

1.2 Contribution

This work extends taint analysis, which is a popular variant of information-flow analysis [37–39] in order to trace the dependence flow of confidential information from data sources to data sinks. A finer granularity of the analysis is obtained by explicitly keeping track of the footprint of data sources in the expressions managed by the program, as well as of the obfuscation impact of the program operators. As the analysis is defined as an instance of the Abstract Interpretation framework [10], the tradeoff between accuracy and efficiency can be tuned by a suitable choice of the concrete domain abstraction.

The main contributions of this paper can be summarized as follows:

- We design an enhanced concrete semantics that makes explicit the dependence of values on local data sources.
- We define the notion of "confidentiality value of an expression" in terms of min/max confidentiality degree of its sources and of min/max obfuscation degree of the operators that are used to generate it.
- We lift the enhanced concrete semantics to (computable) abstract semantics according to the Abstract Interpretation framework.
- We show how a static analysis based on this framework can be used to verify the satisfaction of privacy policies.
- We provide practical evidence of the effectiveness of our approach.

In concrete implementations, tracking indirect information flows negatively impacts the effectiveness of the analysis due to the presence of exceptions. Therefore, our approach disregards them. However, the treatment of implicit flows can be further incorporated into our framework by infusing relational operators' footprint in the different conditional statements' branches.

```
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 }
25 public class UserInfo {
26  String language;
27  String country;
28  String id;
29  Location loc;
30
31  void updateInfo() {
32   Locale localLocale = Locale.getDefault();
33   language = localLocale.getLanguage();
34   country = localLocale.getCountry();
35   String androidId = Settings.Secure.getAndroidId();
36   id = MessageDigest.hashSHA1(androidId);
37   loc = LocationManager.getLastKnownLocation();
38  }
39 }
```

Fig. 1: Snippet of Code from the `Inmobi` Library

1.3 Structure of the Paper

The rest of paper is structured as follows: Section 2 presents some examples that motivate the main novelties of our approach. Sections 3, 4, and 5 describe the syntax, and the enhanced concrete and abstract semantics, respectively. Section 6 introduces the notion of confidentiality and obfuscation values for sources and operators. Section 7 shows how this framework can be applied to the verification of privacy compliance policies. Section 8 discusses related work, while Section 9 concludes.

2 Motivating Examples

2.1 `Inmobi`

Consider the motivating example in Figure 1. This code is extracted from the `Inmobi` library.[3] `Inmobi` is among the three most popular advertisement engines for Android apps [3]. This code sketches the main steps performed by the `Inmobi` library when loading an advertisement banner. This is performed by method `IMBanner.loadBanner()`, that first creates and updates a `UserInfo` object (lines 3-4), then creates and loads the advertisement banner view passing the information about the user (lines 5-6), and finally displays the banner (line 7). Even though, at a first glance, this method does not seem to access any

[3] This library is obfuscated, and some parts (and in particular `BannerView.loadNewAd()`) cannot be decompiled. For the sake of readability, we represented the main components of the library in this code snippet.

confidential information, `user.updateInfo()` collects and transforms various pieces of information about the user and the device, and in particular (i) the language and country of the user from the default `Locale` object (lines 32-34), (ii) the hashing of the Android ID (lines 35-36)[4], and (iii) the last known location (line 37). When a `UserInfo` object is passed to the constructor of `BannerView`, it is stored in a local field. The information contained in this field is then concatenated in the URL (lines 17-19) used to retrieve the advertisement banner. In this way, the data collected by `UserInfo.updateInfo()` is leaked to the advertisement server. This data aggregates various sources (language and country from the `Locale`, the android ID, and the location).

This example shows that we need to track complex flows of information. For instance, with a standard taint analysis [39] an alarm would be raised upon any flow from a source to a sink. In this particular example, we expect that it would be fine to release to the advertisement server some of the sources (e.g., the country and the language, but not the Android ID), so taint analysis could raise a false alarm in this scenario. Indeed, we are interested in computing the global amount of data that is released (that is, country, language, Android ID, and location), and to raise an alarm only if this amount exceeds a specified threshold. In addition, one needs to specify if the transformation performed on the confidential data (e.g., hashing the Android ID) is obfuscating the value sufficiently or not. For instance, the hashing of the ID might be used to track a user or device if the hash clashing is quite rare, and therefore the level of obfuscation performed by this transformation might be insufficient.

2.2 IMSI

The following snippet of code is extracted from internal Android library `com.android.internal.telephony.cdma.RuimRecords`:

```
1 String mImsi = telephonyManager.getDeviceId();
2 log("IMSI:" + mImsi.substring (0, 6) + "xxxxxxxxx");
```

It leaks a portion of the device identifier through the log. The fundamental question here is whether the first 6 characters of the International Mobile Subscriber Identity (IMSI) code contain confidential information that the user does not want to leak outside. The IMSI code is usually made by 15 characters, where the first 3 characters identify the country, the following 2 or 3 characters identify the mobile network, and the rest is used to identify the device.[5] Therefore, we assume that the first 6 characters do not contain confidential information.

[4] The Android ID is "randomly generated when the user first sets up the device and should remain constant for the lifetime of the user's device". Therefore, it is used to track a specific user (rather than a device) by advertisement engines. We simplify the API call to make the code more readable.

[5] http://en.wikipedia.org/wiki/International_mobile_subscriber_identity.

3 Syntax

At the lowest level of the language, we consider expressions on strings ($s \in \mathbb{S}$), integers ($n \in \mathbb{Z}$), and Boolean values ($b \in \mathbb{B}$). We denote by *bexp* and *nexp* Boolean and numerical expressions, respectively. In addition to basic numerical, textual, and Boolean expressions, we introduce label constants (that refer to datastore entries). Let `Lab` be the set of labels. We denote by ℓ, possibly subscripted, labels identifiers in `Lab`. Our language support a statement $read(\ell)$ that returns the value read from the datastore corresponding to the label ℓ. We define string expressions 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 $\circ$ denotes the concatenation of two strings, k denotes a key used to encrypt a textual value, $sub(s, n_1, n_2)$ computes the substring of s from the n_1-th to the n_2-th character, and $hash(s)$ computes the hash value of s. For the sake of simplicity, we focus our formalization on this minimal representative language, and on operations over strings. Our approach can be extended straigthforwardly to support other operations and types.

Finally, we define a standard minimal imperative set of statements. In particular, we support string assignment ($x := sexp$), concatenation ($c_1; c_2$), conditional `if` (`if` *bexp* `then` c_1 `else` c_2), and `while` loops (`while` *bexp* `do` c). In addition, we have a special statement $send(sexp)$ that leaks a string value.

4 Collecting Semantics

4.1 Domain

First of all, we define atomic data expressions by $\mathbb{D} = \{\langle \ell_i, L_i \rangle : i \in I\}$. Given a set of data labels, which identify the locations of the *read-only*[6] datastore a program interacts with, an atomic data expression *adexp* is a set of elements $\langle \ell_i, \{(op_j, \ell'_j) : j \in J\} \rangle$. An element $\langle \ell_i, \{(op_j, \ell'_j) : j \in J\} \rangle$ in *adexp* says that the value of *adexp* has been obtained from the datum stored in the location ℓ_i by combining it with data coming from the locations ℓ'_j through the corresponding operations op_j. In other words, an atomic data expression keeps track, for each source of the expression value, of the set of other data sources that were used to get that value from it. We denote by $\mathbb{D}$ the domain of atomic data.

We focus our collecting semantics on the variables referring to values coming from the datastore. Therefore, we define a data environment mapping local variables in `Var` to atomic data expressions ($D : \texttt{Var} \longrightarrow \wp(\mathbb{D})$). Note that each variable may contain data about different sources (e.g., the concatenation of the strings representing the Android identifier and the location), and therefore each variable is related to a set of atomic data expressions. In addition, the concrete state tracks value information as well ($V : \texttt{Var} \longrightarrow (\mathbb{Z} \cup \mathbb{S})$). Formally, $\Sigma = D \times V$.

[6] We restrict our focus to a read-only datastore, for the sake of simplicity. Extending the model to the general case brings about the problem of aliasing that should be studied further.

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

Fig. 2: Semantics of Expressions on Atomic Data

We then introduce a concrete datastore that contains all the possible atomic data that may be read by a program, where the special label $\star$ is used to represent data coming either from the input of the program or from the constant set of the program itself, i.e. data that is not contained in the datastore.

Definition 1 (Concrete Datastore). *A concrete datastore C is 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 $\ell_i \neq \star\}$.*

Given a program p, we will denote the concrete datastore associated with this program by C_p.
Example Consider the `Inmobi` example from Section 2. Method `updateInfo` accesses various data coming from the datastore. We represent by (i) $\langle$`Language`$,\emptyset\rangle$ the language returned by the `Default` object (line `30`), (ii) $\langle$`Country`$,\emptyset\rangle$ the country returned by the `Default` object (line `31`), and (iii) $\langle$`AndroidId`$,\emptyset\rangle$ the Android identifier. These three data sources are stable, that is, they always return the same values. For the locations (that is, line `34`) it may be the case that different calls of `getLastKnownLocation` retrieves different locations. Therefore, the concrete datastore contains $\langle \texttt{Location}_i, \emptyset\rangle : i \in \mathbb{N}$ as well. Instead, for the `IMSI` example we have only one datum $\langle$`IMSI`$,\emptyset\rangle$.

4.2 Semantics

We suppose that a standard concrete evaluation of numerical ($S_N : nexp \times V \to \mathbb{Z}$) and string ($S_S : sexp \times V \to \mathbb{S}$) expressions is provided, as well as the evaluation of Boolean conditions ($S_B : bexp \times V \to \{\mathsf{true}, \mathsf{false}\}$). In addition, we suppose that the semantic evaluation of label expressions ($S_L : lexp \times \Sigma \to \texttt{Lab}$) returns a data label given a label expression.

The evaluation of the expressions on atomic data $S_A : sexp \times \Sigma \to \wp(\mathbb{D})$ is defined in Fig. 2. Observe that this enhanced concrete semantics of expressions can be seen as an abstract representation of partial execution traces, where each expression tree is projected to the data associated to the labels in `Lab`.

Once the semantics of expression is formalized, the (concrete enhanced) semantics of statements can be expressed as depicted in Figure 3.

$$
\begin{aligned}
S[\![x := sexp]\!](a,v) &= (a[x \mapsto S_A[\![sexp]\!](a,v)], v[x \mapsto S_S[\![sexp]\!](v)]) \\
S[\![send(sexp)]\!](a,v) &= (a,v) \\
S[\![c_1; c_2]\!](a,v) &= S[\![c_2]\!](S[\![c_1]\!](a,v))) \\
S[\![\text{if } bexp \text{ then } c_1 \text{ else } c_2]\!](a,v) &= \begin{cases} S[\![c_1]\!](a,v) & \text{if } S_B[\![bexp]\!](v) \\ S[\![c_2]\!](a,v) & \text{otherwise} \end{cases} \\
S[\![\text{while } bexp \text{ do } c]\!](a,v) &= S[\![\text{ if } (bexp) \ \ (\text{c; while } bexp \text{ do } c)]\!](a,v)
\end{aligned}
$$

Fig. 3: Concrete Semantics of Statements

Example Consider again the `Inmobi` example of Section 2. After the execution of `updateInfo` (line `4`) we have that (i) $\texttt{user.language} \mapsto \{\langle \texttt{Language}, \emptyset \rangle\}$, (ii) $\texttt{user.country} \mapsto \{\langle \texttt{Country}, \emptyset \rangle\}$, (iii) $\texttt{user.id} \mapsto \{\langle \texttt{AndroidId}, \{(\texttt{hash.AndroidId})\}\rangle\}$, and (iv) $\texttt{user.loc} \mapsto \{\langle \texttt{Location}_1, \emptyset \rangle\}$. We then concatenate all this data in a string stored in `url` at line `17`. Therefore, we obtain the following atomic data expression with label `AndroidId`:

$\langle \texttt{AndroidId}, \{(\texttt{hash}, \texttt{AndroidId}), (\circ, \texttt{Language}), (\circ, \texttt{Country}), (\circ, \texttt{Location}_1)\}\rangle$ while for $\texttt{Location}_1$ we obtain $\langle \texttt{Location}_1, \{(\circ, \texttt{AndroidId})\}\rangle$ since this is the last element concatenated when building `url`. For the `IMSI` example, we obtain that the data expression leaked at line 2 is $\langle \texttt{IMSI}, \{([sub, 0, 6], \texttt{IMSI})\}\rangle$.

4.3 Canonical Form of Atomic Data

The definition of atomic data does not impose any constraint on the number of elements. In particular, the same source label can appear several times in an atomic datum, when its data have multiple impact on the expression's value. However, if we are just interested to observe the sources of an expression, and the set of operators applied to each source, a more compact representation of atomic data can be given, where each source label appears at most once.

Given an atomic datum $d = \{\langle \ell_j, L_j \rangle : j \in J\}$, we denote by $\texttt{src}(d)$ its source set $\{\ell_j : j \in J\}$. Moreover, given a label ℓ and an atomic datum d, we denote by $\texttt{links}(\ell, d)$ the links set of ℓ in d if $\ell \in \texttt{src}(d)$, and $\emptyset$ otherwise.

Definition 2. *We say that an atomic datum d is in* canonical form *if every label in* $\texttt{src}(d)$ *occurs as a source label exactly once in d. Given an atomic datum, its canonical form can be obtained by applying the following unary source collapse operator ρ:*

$$\rho(d) = \{\langle \ell, \cup \texttt{links}(\ell, d) \rangle : \ell \in \texttt{src}(d)\}$$

5 Abstract Semantics

There are two main ways to get abstractions of the concrete semantics defined so far: abstracting values, and abstracting labels. The abstract elements should be an overapproximation of the concrete values assigned to variables in the concrete computation steps.

5.1 Values Abstraction

Values can be abstracted by means of well-known either relational or non-relational domains for numerical and textual values [9, 26]. Therefore, we suppose that a value abstract domain V^a is provided, and it is equipped with the standard lattice and semantic operators.

5.2 Labels Abstraction

Labels can be abstracted by any abstract domain for categorical data, like a flat constant propagation domain. Observe that when dealing with data stored in relational form, i.e. by means of bi-dimensional tables, a relational abstract domain for array representation can be adopted, as defined in [11].
Example In the `Inmobi` example of Section 2, we do not need to apply any abstraction on `Language`, `Country`, and `AndroidId`, since these are persistent throughout the execution. Instead, we need to apply abstraction to the locations $\texttt{Location}_i : i \in \mathbb{N}$, since the statement at line `37` may produce many values. Therefore, we abstract together all the locations produced by the same program point `pp` with $\texttt{Location}^{\texttt{pp}}$. In our example, this means that we abstract the data source at line `37` with $\texttt{Location}^{37}$.

5.3 Atomic Data Abstraction

We are now in position to formalize the atomic data abstract domain $\mathbb{AD}$.

Definition 3 (Abstract Atomic Data). *Given a set of atomic data, an abstract element will be a set of tuples* $\{\langle \ell^a_j, L^{a\sqcap}_j, L^{a\sqcup}_j \rangle : j \in J\} \in \mathbb{AD}$, *where*

- ℓ^a_j *is an element of an abstract domain that abstracts labels in* `Lab`
- $L^{a\sqcap}_j = \{(op^a_{ij}, \ell^a_{ij}) : i \in I\}$ *is an under-approximation of the set of operators applied to the sources represented by* ℓ^a_j *with values coming from sources represented by* ℓ^a_{ij}
- $L^{a\sqcup}_j = \{(op^a_{ij}, \ell^a_{ij}) : i \in I'\}$ *is an over-approximation of the set of operators applied to the sources represented by* ℓ^a_j *with values coming from sources represented by* ℓ^a_{ij}
- $L^{a\sqcap}_j \subseteq L^{a\sqcup}_j$.

The order on the abstract elements is given by the order on the Cartesian product of the components' domain, and the least upper bound and greatest lower bound operators are defined accordingly.

Definition 4 (Partial Order on Abstract Atomic Data). *Given two abstract atomic data* $d_1 = \{\langle \ell^a_{1i}, L^{a\sqcap}_{1i}, L^{a\sqcup}_{1i} \rangle : i \in I_1\}$ *and* $d_2 = \{\langle \ell^a_{2i}, L^{a\sqcap}_{2i}, L^{a\sqcup}_{2i} \rangle : i \in I_2\}$ *on the same abstract domains for values and labels,*

$$d_1 \sqsubseteq d_2 \Leftrightarrow \forall i \in I_1\ \exists j \in I_2 : \ell^a_{1i} = \ell^a_{2j},\ L^{a\sqcap}_{1i} \supseteq L^{a\sqcap}_{2j},\ L^{a\sqcup}_{1i} \subseteq L^{a\sqcup}_{2j}$$

Given an abstract atomic datum $\{\langle \ell^a_j, L^{a\sqcap}_j, L^{a\sqcup}_j \rangle : j \in J\}$, we denote by $\texttt{src}(d)$ its source set $\{\ell^a_j : j \in J\}$.

Definition 5 (Least Upper bound of Abstract Atomic Data). *Given two abstract atomic data* $d_1 = \{\langle \ell^a_{1i}, L^{a\sqcap}_{1i}, L^{a\sqcup}_{1i}\rangle : i \in I_1\}$ *and* $d_2 = \{\langle \ell^a_{2i}, A, L^{a\sqcap}_{2i}, L^{a\sqcup}_{2i}\rangle : i \in I_2\}$ *on the same abstract domains for values and labels, the least upper bound of* d_1 *and* d_2 *is the atomic datum*

$$d_1 \sqcup d_2 = \bigcup_{\ell^a \in \mathtt{src}(d_1)\cup\mathtt{src}(d_2)} \left\{ \begin{array}{ll} \langle \ell^a, L^{a\sqcap}_1, L^{a\sqcup}_1\rangle & \textit{if } \ell^a \in \mathtt{src}(d_1) \setminus \mathtt{src}(d_2) \\ \langle \ell^a, L^{a\sqcap}_2, L^{a\sqcup}_2\rangle & \textit{if } \ell^a \in \mathtt{src}(d_2) \setminus \mathtt{src}(d_1) \\ \langle \ell^a, L^{a\sqcap}_1 \cap L^{a\sqcap}_2, L^{a\sqcup}_1 \cup L^{a\sqcup}_2, \rangle & \textit{otherwise} \end{array} \right\}$$

Let $\mathtt{Lab}^a$ and $\mathtt{A}$ be complete lattices featuring Galois Connections with the concrete domains of labels and values, respectively. Let $(\gamma_{\mathtt{Lab}}, \alpha_{\mathtt{Lab}}), (\gamma_{\mathtt{A}}, \alpha_{\mathtt{A}})$ be the corresponding concretization and abstraction functions. When applied to a set of links $\{(op_i, \ell_i) : i \in I\}$, the function $\alpha_{\mathtt{Lab}}$ returns the set $\{(op^a_i, \alpha_{\mathtt{Lab}}(\ell_i)) : i \in I\}$, where op^a_i is the abstract operator that safely approximates op_i in the abstract domain $\mathtt{A}$.

Definition 6 (Abstraction function). *The abstraction function* $\alpha : \wp(\mathbb{D}) \longrightarrow \mathbb{AD}$ *is first defined on singletons and then extended to sets by applying the least upper bound operator.*

$$\alpha_s(\{\langle \ell_i, L_i\rangle : i \in I\}) = \{\langle \alpha_{\mathtt{Lab}}(\ell_i), \alpha_{\mathtt{Lab}}(L_i), \alpha_{\mathtt{Lab}}(L_i)\rangle : i \in I\}$$
$$\alpha(\{d_j \in \mathbb{D} : j \in J\}) = \bigsqcup_{j\in J} \alpha_s(d_j).$$

Notice that in the definition above, when considering a single atomic datum, in its abstract representation the under- and over-approximations of the link sets are equal. The gap among these sets is introduced in fact by the least upper bound operator.

Definition 7 (Concretization function). *The concretization of abstract atomic data is defined as an adjoint of the abstraction function:* $\gamma(ad) = \{d \in \mathbb{D} : \alpha(d) \sqsubseteq ad\}$

Theorem 1 (Galois Connection). *The functions* α *and* γ *defined above form a Galois Connection between* $\wp(\mathbb{D})$ *and* $\mathbb{AD}$*, i.e.:*

i) α *and* γ *are monotone,*
ii) $\forall ad \in \mathbb{AD} : \alpha(\gamma(ad)) \sqsubseteq_{\mathbb{AD}} ad$
iii) $\forall S \subseteq \mathbb{D} : S \subseteq \gamma(\alpha(S))$.

Proof. $\mathbb{AD}$ is the Cartesian product of abstract domains featuring Galois Connections with the concrete domain $\wp(\mathbb{D})$, and the functions α and γ are defined in canonical way w.r.t. the Cartesian product [8].

We define by $AD^a : \mathtt{Var} \mapsto \wp(\mathbb{AD})$ the component of the abstract domain tracking information on atomic data expressions. The partial order, the upper bound and the concretization function are defined as pointwise application of the operators defined on $\mathbb{AD}$.

5.4 Abstract Domain

Our abstract domain is the Cartesian product of the Atomic Data abstract domain (AD^a), and the value domain (V^a).

5.5 Data-Store Abstraction

The analysis of a program P aimed at verifying that it satisfies a given confidentiality policy with respect to data stored in devices running P can be defined either as a "datastore-aware" analysis, i.e. running the analysis on the actual data contained in the device, or in a "datastore-unaware" way, i.e. running the analysis on a generic datastore that represents the actual datastores under a suitable abstraction of labels and values.

A "datastore-aware" analysis has the advantage of being in general more accurate, as it allows to deal with the actual values that are leaked by the program. However, this scenario requires the analysis being applied only once the program is installed on the device, as an app that runs on the device itself or on a third-party verifier that should be given access permission to the device's datastore.

The accuracy of a "datastore-unaware" analysis heavily relies on the datastore abstraction, but it has the advantage of being applicable to the program with no need to access the actual confidential data when running the analysis itself.

Definition 8 (Abstract Datastore). *Given a Galois connection between* $\wp(\mathbb{D})$ *and* $\mathbb{AD}$*, an abstract datastore is a set* $D^a = \{\{\langle \ell^a_j, \emptyset, \emptyset\rangle\} : j \in J\} \subseteq \mathbb{AD}$ *such that* $\forall i, j \in J : i \neq j \Rightarrow \ell^a_i \neq \ell^a_j$ *.*

An abstract datastore $D^a = \{\{\langle \ell^a_j, \emptyset, \emptyset\rangle\} : j \in J\}$ is an abstraction of all concrete datastores $D = \{\{\langle \ell_i, \emptyset\rangle\} : i \in I\}$ that satisfies the following conditions: $\bigcup_{i\in I}\{\alpha(\ell_i)\} = \bigcup_{j\in J}\{\ell^a_j\}$, and $\forall i \in I\ \exists! j \in J$ such that $\ell_i \in \gamma_{\texttt{Lab}}(\ell^a_j)$.

Example Consider the `Inmobi` example introduced in Section 2. In particular, we have the following atomic data expressions: $\langle \texttt{Language}, \emptyset, \emptyset\rangle$, $\langle \texttt{Country}, \emptyset, \emptyset\rangle$, and $\langle \texttt{AndroidId}, \emptyset, \emptyset\rangle$. The only case that is slightly different regards `Location` since we abstract with $\langle \texttt{Location}^{37}, \emptyset, \emptyset\rangle$ all the concrete data expressions in $\langle \texttt{Location}_i, \emptyset\rangle : i \in \mathbb{N}$. Similarly, for the `IMSI` example we have $\langle \texttt{IMSI}, \emptyset, \emptyset\rangle$.

5.6 Abstract Semantics of Statements

Figure 4 depicts the abstract semantics of statements[7]. We omit the abstract semantics of expressions, as it can be easily formalized by mimicking the concrete semantics, the only difference being that (1) every operation has impact on both link sets associated to an abstract label, and (2) abstract atomic data are kept in canonical form by systematically applying the following normalization operator in presence of multiple occurrences of the same label in the source set of an abstract atomic datum: $\rho(ad) = \{\langle \ell^a, \cap\texttt{links}(\ell^a, ad), \cup\texttt{links}(\ell^a, ad)\rangle : \ell^a \in \texttt{src}(ad)\}$

Example The abstract semantics of the `Inmobi` example does not substantially differ from the concrete semantics for the the example of Section 2. After the execution of `updateInfo` (line 4), we have the same information described in Section 4.2, the only difference being that the abstract label for the location is

[7] Observe that this semantics does not capture indirect information flow.

$$
\begin{aligned}
S^a[\![x := sexp]\!](a^a, v^a) &= (a^a, S^a_v[\![x := sexp]\!](v^a)) \\
S^a[\![send(sexp)]\!](a^a, v^a) &= (a^a, v^a) \\
S^a[\![c_1; c_2]\!](a^a, v^a) &= S^a[\![c_2]\!](S^a[\![c_1]\!](a^a, v^a)) \\
S^a[\![\text{if } bexp \text{ then } c_1 \text{ else } c_2]\!](a^a, v^a) &= S^a[\![c_1]\!](a^a, S^a_B[\![bexp]\!](v^a)) \sqcup S^a[\![c_2]\!](a^a, S^a_B[\![\neg bexp]\!](v^a)) \\
S^a[\![\text{while } bexp \text{ do } c]\!](a^a, v^a) &= fix(S^a[\![\text{ if } (bexp) \text{ (while } bexp \text{ do } c)]\!](a^a, v^a))
\end{aligned}
$$

Fig. 4: (Abstract) Semantics of Statements

$\texttt{Location}^{37}$ instead of the concrete label $\texttt{Location}_1$[8]. The same consideration applies to the results of the concatenation at line 17.

For the IMSI example, the abstract semantics tracks that we log the abstract datum $\langle \texttt{IMSI}, \{([sub, 0, 6], \texttt{IMSI})\}, \{([sub, 0, 6], \texttt{IMSI})\}\rangle$.

The following theorem formalizes the soundness of the analysis.

Theorem 2. *The abstract semantics of a program P with an abstract datastore D^a is a conservative (sound) over-approximation of the enhanced concrete semantics of P with a concrete datastore $D \in \gamma(D^a)$.*

Proof. By induction on the lenght of the trace as in [23], by lifting the local correctness of the operations to the Cartesian product [8].

We observe in particular that for each execution of P with input I, if a value assigned to a variable v in the store is obtained from values coming from local data stored in ℓ through operations in R, then there is a corresponding abstract trace of P with input $\alpha(\texttt{I})$, assigning v an abstract atomic datum ad such that (i) $\ell^a \in \texttt{src}(ad)$, and (ii) $R \subseteq \{\texttt{op} : (\texttt{op}^a, \ell^a_j) \in \texttt{links}(\ell^a, ad)\}$ (where $\ell^a = \alpha(\ell)$ is the label in the abstract datastore representing ℓ).

6 Confidentiality and Obfuscation

So far, we made no distinctions among data contained in the data-store, with respect to their confidentiality level. In general, we can consider a lattice of confidentiality levels S, and we can associate to each label ℓ in Lab an element $s_\ell \in \texttt{S}$. Confidentiality levels are assigned to labels, and values corresponding to these labels will inherit from them the same confidentiality level.

On the operation side, we introduce the notion of *obfuscation degree*. The intuitive idea is that if you know which operation has been applied to get an expression, and the expression itself, you can look at the amount of information which is necessary to recover the sources the operation applied to. This leads us to assume the existence of a partial-order relation among operations that captures their different obfuscation impact.

[8] For the sake of simplicity, we ignore the issues related with heap abstraction. It has been demonstrated [16] that value domains (like AD^a and V^a) can be combined with heap abstractions relying on standard operators of value domains.

This can be seen as a generalization of the all-or-nothing tainting approach [37, 39], where only declassification operators (e.g., encryption) are tracked.

The obfuscation degree of an operator can be seen as a measure of the complexity of the brute-force analysis needed by an external observer in order to detect the actual source data when knowing just the result of the operation and the applied operator.

Definition 9 (Obfuscation Degree). *Consider a complete lattice* $(\mathtt{O}, \sqsubseteq_{\mathtt{O}})$*, and a map* $\zeta : Op \to \mathtt{O}$*, such that* $\zeta(op_1) \sqsubseteq_{\mathtt{O}} \zeta(op_2)$ *if the obfuscation power of* op_1 *is smaller than the obfuscation power of* op_2*. We say that the obfuscation degree of an operator* $op \in Op$ *is* $\zeta(op)$*.*

Example The string operators of *sexp* introduced in Section 3 have different obfuscation degrees. For instance, *encrypt* obfuscates more that *hash*, while the power of obfuscation of *substring* may depend on the indexes used to compute the substring, and the particular information contained in the string. For instance, in the `IMSI` example of Section 2 the `substring` operator at line 2 has a high obfuscation degree, but this relies on the value information tracked on the indexes passed to `substring`.

6.1 Confidentiality of Atomic Data

Given an atomic datum, a confidentiality value can be assigned to it by considering an under- and over-approximation of the confidentiality levels of source data, and by considering an under- and over- approximation of obfuscation power of the operations applied to them.

We first define it at a concrete level, on top of our instrumented atomic data semantics, and then we can lift this notion to the abstract case.

Definition 10 (Confidentiality of Atomic Data for Monotonic Operators). *Let* $\mathtt{S}$ *be a lattice representing confidentiality levels of labels. Let* $\mathtt{O}$ *be a lattice representing the obfuscation power of operators. Finally, let* η *and* ζ *be functions assigning confidentiality/obfuscation values in* $\mathtt{S}$ *and* $\mathtt{O}$ *to labels and operators, respectively.*

If the combination of operators in $\bigcup_{i \in I} L_i$ *is monotonic with respect to the obfuscation order in the lattice* $\mathtt{O}$*, the confidentiality value of an atomic datum* $\{\langle \ell_i, L_i \rangle : i \in I\}$ *with respect to* (η, ζ) *is the tuple* $(sc_{min}, sc_{max}, lc_{min}, lc_{max})$*, where:*

$$sc_{min} = \sqcap_{\mathtt{D}}\{\eta(\ell_i) : i \in I\} \qquad lc_{min} = \sqcap_{\mathtt{O}}\{\zeta(op_{ij}) : (op_{ij}, \ell_j) \in L_i, i \in I\}$$
$$sc_{max} = \sqcup_{\mathtt{D}}\{\eta(\ell_i) : i \in I\} \qquad lc_{max} = \sqcup_{\mathtt{O}}\{\zeta(op_{ij}) : (op_{ij}, \ell_j) \in L_i, i \in I\}$$

Example Imagine that we have $\mathtt{L} < \mathtt{M} < \mathtt{H}$ as both the confidentiality and obfuscation lattice. We then establish that *encrypt* has `H` obfuscation level, and *hash* has level `M`, whereas the obfuscation level of substring depends on the parameters: $[sub, k_1, k_2]$ has level `L` if $k_1 = 6$ and $k_2 = 9$, it has level `M` if $6 < k_1 + k_2 < 15$, and it has level `H` if $k_1 + k_2 \leq 6$. Consider then the concrete labels of the `Inmobi` example introduced in Section 4.1. We define as `L` both `Language` and `Country`, since they do not contain particularly confidential information. Instead, we define as `H` `AndroidId`, since this datum allows to uniquely identify our Android account,

and track our activity. Finally, $\texttt{Location}_i : i \in \mathbb{N}$ are all $\texttt{M}$, since these locations allow to identify our geographical location at a given point, but do not uniquely identify us. For the IMSI example, we obtain that for the data expression leaked at line 2, i.e. $\langle \texttt{IMSI}, \{([sub, 0, 6], \texttt{IMSI})\}\rangle$, we get $sc_{min} = sc_{max} = \texttt{H}$ and $lc_{min} = lc_{max} = \texttt{H}$. This says that even if sensitive data items are leaked, a powerful obfuscation is definitely applied to them before releasing them.

Notice that Definition 10 is explicitly restricted to the case of operators whose combination is monotonic with respect to the obfuscation order in $\texttt{O}$. If we are interested to consider also programs where the combination of operators is non-monotonic, we just need to give an obfuscation value to sets of operators instead of singletons.

Definition 11 (Confidentiality of Atomic Data - General Case). *Let* $\texttt{S}$ *be a lattice representing confidentiality levels of labels. Let* $\texttt{O}$ *be a lattice representing the obfuscation power of operators. Finally, let* η *be a function assigning confidentiality values in* $\texttt{S}$ *to labels, and let* ζ *be a function assigning to each set of operators an interval in* $\texttt{O} \times \texttt{O}$ *representing its min and max obfuscation power.*

The confidentiality value of an atomic datum $\{\langle \ell_i, L_i\rangle : i \in I\}$ *with respect to* (η, ζ) *is the tuple* $(sc_{min}, sc_{max}, lc_{min}, lc_{max})$*, where:*

$$sc_{min} = \sqcap_{\texttt{D}}\{\eta(\ell_i) : i \in I\} \qquad lc_{min} = \sqcap_{\texttt{O}}\{\pi_1(\zeta(\{op_{ij} : (op_{ij}, \ell_j) \in L_i\})) : i \in I\}$$
$$sc_{max} = \sqcup_{\texttt{D}}\{\eta(\ell_i) : i \in I\} \qquad lc_{max} = \sqcup_{\texttt{O}}\{\pi_2(\zeta(\{op_{ij} : (op_{ij}, \ell_j) \in L_i\})) : i \in I\}$$

where π_1 *and* π_2 *denote the on the min and max element of the interval, respectively.*

Notice that keeping track of mimimal confidentaility and maximal obfuscation allows us (when they are equal to maximal confidentiality and minimal obfuscation, respctively) to be aware of the precision of these values.

6.2 Confidentiality of Abstract Atomic Data

In order to define the confidentiality value of abstract atomic data, we need to assign a confidentiality value to abstract labels. As abstract labels may represent concrete labels with different confidentiality values, the confidentiality function η^a returns an interval min/max of values in $\texttt{S} \times \texttt{S}$ instead of a single value. On the obfuscation side, we just lift the value, as we can assume there is a always a one-to-one correspondence between concrete and abstract operators.

As in the concrete setting, we distinguish the case in which all operator combinations behave monotonically with respect to the obfuscation order, from the general case, that takes into account non-monotonic behaviors, the price to pay being to assign obfuscation values to sets of operators instead of single ones.

Definition 12 (Confidentiality Value of Abstract Atomic Data - Monotonic Operators). *Let* $\texttt{S}$ *and* $\texttt{O}$ *be the lattices representing the labels' confidentiality and the obfuscation power of operators, respectively. Let* η *and* ζ *be functions assigning confidentiality/obfuscation values in* $\texttt{S}$ *and* $\texttt{O}$ *to (concrete) labels and operators, respectively. Let* $\eta^a : \texttt{Lab}^{\texttt{a}} \rightarrow \texttt{S} \times \texttt{S}$ *such that* $\eta^a(\ell^a) = [\sqcap\{\eta(\ell) : \ell \in \gamma(\ell^a)\}, \sqcup\{\eta(\ell) : \ell \in \gamma(\ell^a)\}]$*. Let* ζ^a *be the function assigning to each abstract operator the same obfuscation value assigned by* ζ *to the concrete operator it corresponds.*

If the combination of operators in Op *appearing in* $\bigcup_{i \in I} L_i^{a\sqcup}$ *is monotonic with respect to the obfuscation order in* O*, then the confidentiality value of an abstract atomic datum* $\{\langle \ell_i^a, val_i^a, A_i, L_i^{a\sqcap}, L_i^{a\sqcup} \rangle : i \in I\}$ *is a tuple* $(sc_{min}^a, sc_{max}^a, lc_{min}^a, lc_{max}^a)$*, where:*

$$sc_{min}^a = \sqcap_{\mathsf{S}}\{\pi_1(\eta^a(\ell_i^a)) : i \in I\} \qquad lc_{min}^a = \sqcap_{\mathsf{O}}\{\zeta^a(op_{ij}^a) : (op_{ij}^a, \ell_j^a) \in L_i^{a\sqcap}, i \in I\}$$
$$sc_{max}^a = \sqcup_{\mathsf{S}}\{\pi_2(\eta^a(\ell_i^a)) : i \in I\} \qquad lc_{max}^a = \sqcup_{\mathsf{O}}\{\zeta^a(op_{ij}^a) : (op_{ij}^a, \ell_j^a) \in L_i^{a\sqcup}, i \in I\}$$

where π_1 *and* π_2 *denote the min and max element of the interval, respectively.*

Observe that lc_{min}^a is obtained as the greatest lower bound of the obfuscation values corresponding to operators that are surely applied to compute the value, while lc_{max}^a is obtained as the least upper bound of the obfuscation values corresponding to operators that are possibly applied to compute the value. As $L_i^{a\sqcap} \subseteq L_i^{a\sqcup}$ for each $i \in I$, we get that $lc_{min}^a \sqsubseteq_{\mathsf{O}} lc_{max}^a$.

Definition 13 (Confidentiality Value of Abstract Atomic Data - General Case). *Let* S *and* O *be the lattices representing the labels' confidentiality and the obfuscation power of operators, respectively. Let* η *and* ζ *be functions assigning confidentiality/obfuscation values in* S *and* O *to (concrete) labels and operators, respectively. Let* $\eta^a : \mathsf{Lab}^{\mathsf{a}} \rightarrow \mathsf{S} \times \mathsf{S}$ *such that* $\eta^a(\ell^a) = [\sqcap\{\eta(\ell) : \ell \in \gamma(\ell^a)\}, \sqcup\{\eta(\ell) : \ell \in \gamma(\ell^a)\}]$*. Finally, let* ζ^a *be a function assigning to each set of (abstract) operators an interval in* $\mathsf{O} \times \mathsf{O}$ *representing the min and max obfuscation power.*

The confidentiality value of an abstract atomic datum $\{\langle \ell_i^a, val_i^a, A_i, L_i^{a\sqcap}, L_i^{a\sqcup} \rangle : i \in I\}$ *is a tuple* $(sc_{min}^a, sc_{max}^a, lc_{min}^a, lc_{max}^a)$*, where:*

$$sc_{min}^a = \sqcap_{\mathsf{S}}\{\pi_1(\eta^a(\ell_i^a)) : i \in I\} \quad lc_{min}^a = \sqcap_{\mathsf{O}}\{\pi_1(\zeta^a(S)) : S \subseteq \{op_{ij}^a : (op_{ij}^a, \ell_j^a) \in L_i^{a\sqcup}\}, i \in I\}$$
$$sc_{max}^a = \sqcup_{\mathsf{S}}\{\pi_2(\eta^a(\ell_i^a)) : i \in I\} \quad lc_{max}^a = \sqcup_{\mathsf{O}}\{\pi_2(\zeta^a(S)) : S \subseteq \{op_{ij}^a : (op_{ij}^a, \ell_j^a) \in L_i^{a\sqcup}\}, i \in I\}$$

where π_1 *and* π_2 *denote the on the min and max element of the interval, respectively.*

Notice that lc_{min}^a and lc_{max}^a in the general case are both obtained as the greatest lower bound and least upper bound, respectively, of the obfuscation values corresponding to operators that are possibly applied to compute the value. This conservative approach guarantees the soundness of the result also in presence of operators whose combination does not behave monotonically with respect to obfuscation, i.e. the monotonicity of confidentiality with respect to the partial order in the domain of abstract atomic data.

7 Privacy Compliance Policies

Definition 14 (Confidentiality Policy). *Given the set of data source labels* Lab*, and the confidentiality/obfuscation lattices* S *and* O *for labels and operations, respectively, a confidentiality policy is a tuple* $\pi = (\eta, \zeta, \kappa_{sc\_max}, \kappa_{lc\_min})$ *such that*

- η, ζ *assign each label and each operator a corresponding value in the confidentiality lattices* S *and* O*, respectively.*
- κ_{sc_max} *is a source confidentiality threshold (the max confidentiality level allowed for sources).*

– κ_{lc_min} *is an obfuscation threshold (the min obfuscation level required for operators).*

Given a program P, let X be the set of concrete/abstract atomic data P generated as an output. We say that P satisfies the confidentiality policy $\pi = (\eta, \zeta, \kappa_{sc\_max}, \kappa_{lc\_min})$ if:

$\forall d \in X$, if $(sc_{min}, sc_{max}, lc_{min}, lc_{max})$ is the confidentiality value of d with respect to (η, ζ), then, $sc_{max} \sqsubseteq_{\mathtt{S}} \kappa_{sc\_max}$ and $lc_{min} \sqsupseteq_{\mathtt{O}} \kappa_{lc\_min}$.

Theorem 3. *Consider a program P, an abstract datastore A, and a confidentiality policy $\pi = (\eta, \zeta, \kappa_{sc\_max}, \kappa_{lc\_min})$. If the program P terminates, and the output of the analysis on P and A satisfies π, then any actual execution of program P on a concrete datastore in $\gamma(A)$ satisfies the confidentiality policy π.*

Proof. By Theorem 2, and by the monotonicity of confidentiality values with respect to the partial order on the domain of abstract atomic data.

Example A reasonable privacy policy for the `Inmobi` example of Section 2 may be that a datum can be released only if its obfuscation level is equal or higher than its confidentiality level. This program satisfies this model for `Country` and `Language` (whose confidentiality level is `L` and they are released without any obfuscation), but not for $\mathtt{Location_i}$ (with confidentiality level `M` and released without any obfuscation) and `AndroidId` (whose confidentiality level is `H` and it is released after invoking *hash*, that is, with obfuscation level `M`).

7.1 Sources' Confidentiality Policies

The definition of confidentiality policy of atomic data discussed in the previous sections allows to capture the min and max levels of confidentiality/obfuscation carried by the values returned by a program, or shared with other applications.

As an orthogonal approach, we may define (at the concrete level) a confidentiality policy as a propositional formula that captures constraints on the allowed releasing levels of confidential data in the datastore, and verify if the atomic data returned by the concrete execution of the program satisfy that formula.

Let `Lab` denote as usual the set of data source labels, and `Op` denote the set of operators in the program. Consider a set of propositional variables `V` (with empty intersection with the set of program variables), and a function λ that maps elements of V into either labels or links.

$$\lambda : \mathtt{V} \longrightarrow \mathtt{Lab} \cup \{(op, \ell) : op \in \mathtt{Op}, \ell \in \mathtt{Lab}\}$$

A *policy formula* is a positive propositional formula on `V`, i.e. a propositional formula using only $\wedge, \vee$ and $\leftrightarrow$ logical operators[9].

Example For instance, we can express the fact that we can leak the Android ID if encrypted and the location, or the hashed Android ID, or the first six

[9] The advantage of using positive formulas, i.e. formulas that are satisfied by assigning true to all its propositional variables, is that they well capture monotonic behaviors [42].

characters of the IMSI, by means of the formula $\varphi = (x \wedge y) \vee z \vee w$, where $\lambda(x) = ([encrypt, k], \texttt{AndroidId})$, $\lambda(y) = \texttt{Location}$, and $\lambda(z) = (hash, \texttt{AndroidId})$, $\lambda(w) = ([sub, 0, 6], \texttt{AndroidId})$.

Given a set of atomic data S, a set of propositional variables $\texttt{V}$ and an assignment λ on $\texttt{V}$, we say that S satisfies the policy formula φ on $\texttt{V}$ if $S, \lambda \models \varphi$, as defined inductively as follows:

$$
\begin{array}{ll}
S, \lambda \models v \in \texttt{V} & \Leftrightarrow \begin{cases} \lambda(v) \in \bigcup_{d \in S} \texttt{src}(d) & \text{if } \lambda(v) \in \texttt{Lab} \\ \lambda(v) \in \bigcup_{d \in S} \texttt{links}(d) & \text{otherwise.} \end{cases} \\
S, \lambda \models \varphi_1 \vee \varphi_2 & \Leftrightarrow S, \lambda \models \varphi_1 \text{ or } S, \lambda \models \varphi_2 \\
S, \lambda \models \varphi_1 \wedge \varphi_2 & \Leftrightarrow S, \lambda \models \varphi_1 \text{ and } S, \lambda \models \varphi_2 \\
S, \lambda \models \varphi_1 \leftrightarrow \varphi_2 & \Leftrightarrow S, \lambda \models \varphi_1 \text{ iff } S, \lambda \models \varphi_2.
\end{array}
$$

Observe that, by construction, if the data resource denoted by $\lambda(x)$ contributes to any of the values represented by the atomic data S, then $S, \lambda \models x$.

Example The formula φ in the Example above is satisfied by the data expressions leaked by either Inmobi and IMSI as described in the Example in Section 3.2.

Observe that checking the policy reduces to checking the satisfiability of the propositional assignment [42].

When we lift to the abstract setting, three-valued models of propositional formulas should be used in order to preserve soundness. Let S^a be a set of abstract atomic data, and $\lambda^a : \texttt{V} \longrightarrow \texttt{Lab}^{\texttt{a}} \cup \{(op^a, \ell^a) : op \in \texttt{Op}, \ell^a \in \texttt{Lab}^{\texttt{a}}\}$ be a function mapping propositional variables into abstract labels and links. Consider the three value assignment $assign(S^a, \lambda) : \texttt{V} \rightarrow \{\texttt{true}, \texttt{false}, \top\}$ defined by

$$
assign(S^a, \lambda^a)(v) = \begin{cases} \texttt{true} & \text{if } \lambda^a(v) \in \texttt{Lab}^{\texttt{a}} \text{ and } \lambda^a(v) \in \bigcup_{d \in S} \texttt{links\_lab}(\texttt{L}^{\sqcap}_d) \\ & \text{or} \\ & \text{if } \lambda^a(v) \notin \texttt{Lab}^{\texttt{a}} \text{ and } \lambda^a(v) \in \bigcup_{d \in S} \texttt{L}^{\sqcap}(d) \\ \texttt{false} & \text{if } \lambda^a(v) \in \texttt{Lab}^{\texttt{a}} \text{ and } \lambda^a(v) \notin \bigcup_{d \in S} \texttt{src}(d) \\ & \text{or} \\ & \text{if } \lambda^a(v) \notin \texttt{Lab}^{\texttt{a}} \text{ and } \lambda^a(v) \notin \bigcup_{d \in S} \texttt{L}^{\sqcup}(d) \\ \top & \text{otherwise} \end{cases}
$$

(where $\texttt{L}^{\sqcap}(d)$ and $\texttt{L}^{\sqcup}(d)$ denote the last two components of d, respectively), and consider the logical operators extended to the top value $\top$ by:

$\wedge$	true	false	$\top$
$\top$	$\top$	false	$\top$

$\vee$	true	false	$\top$
$\top$	true	$\top$	$\top$

$\leftrightarrow$	true	false	$\top$
$\top$	$\top$	$\top$	$\top$

Theorem 4. *Let φ be a positive formula on a set $\texttt{V}$ of propositional variables, S^a be a set of abstract atomic data, and $\lambda^a : \texttt{V} \longrightarrow \texttt{Lab}^{\texttt{a}} \cup \{(op^a, \ell^a) : op \in \texttt{Op}, \ell^a \in \texttt{Lab}^{\texttt{a}}\}$ be a function mapping propositional variables into abstract labels and links. If $assign(S, \lambda)(\varphi) = \texttt{true}$, then there is a set of atomic data $S \subseteq \bigcup_{d \in S^a} \gamma(d)$ and a function λ satisfying $\forall v \in \texttt{V} : \lambda(v) \in \gamma_{\texttt{Lab}}(\lambda^a(v))$, such that $S, \lambda \models \varphi$.*

Proof. By structural induction on the formula φ, and by Theorem 1.

8 Related Work

In this section, we discuss in details how our work compares to some similar approaches in the area in addition to the high-level overview on the state of the art of Section 1.1.

Quantitative Information Flow (QIF) [25] is aimed at measuring the quantity of information that is leaked by a program. A given confidential datum might be manipulated by the program, that at the end releases only partial information. Then, the analysis checks if the quantity of released information is below a given threshold. Our approach shares with QIF the intuition that is crucial to estimates the quantity of information revealed, since ofter it is necessary to partially disclose a part of the information. Nevertheless, instead of measuring a quantity, our approach tracks the set of operators that have been applied to the datum before its release, and then we check if this matches what specified on the policy. We believe that QIF can be seen as an abstraction of a concrete semantics tracking the exact order of operators applied to a datum (instead of a set of operators as we do). In addition, QIF can track implicit flows, while we explicitly ignored these flows as we believe they lead to many false alarms.

Declassification-based approaches [33] suppose that a list of declassifier operators is given, and as soon as one of these operators is applied to a datum, then it can be sent to a sink. Our analysis can represent declassifiers through a policy stating that it is allowed to release the data on which at least one of these operators has been applied.

Decentralized Information Flow Control [20, 27] systems represent a finer grained and more expressive model, in which each process can declassify information, rather than a central authority as in centralized system and classical declassification-based approaches. Nevertheless, our analysis can support this more complex scenarios by defining specific policies per process.

Another approach that has gained relevant results recently is Differential Privacy [13, 14]. Given a data store, the goal of Differential Privacy is to discover if the variation of a query over the data set stays below a given threshold when an entry is added. Usually, some statistical noise is added in order to ensure differential privacy. Our approach may track how the data from the data set is aggregated and noise added, and this may be a first step towards proving differential privacy. However, how to relate this information on the operators applied to data and ϵ-differential privacy is not straightforward at all, and it requires further investigation.

9 Conclusion

Our semantic framework for fine-grained information-flow analysis captures how the values released by an application may partially reveal confidential data stored on the device through different levels of obfuscation. The enhanced concrete semantics and the generic abstract domain we presented provide a workbench for (static) analysis of mobile apps that can be tuned by setting a few parameters: the domain representing values, the domain representing data

locations, and the confidentiality and obfuscation values for data and operators. This data-centric approach may be utilized to refine existing tools like [15, 41, 40] aimed at enforcing privacy policies, providing the user with more accurate privacy control.

The problem of formalizing how the semantics of operations reflects on the corresponding obfuscation values, as well as the problem of assisting the user in the definition of privacy compliance policies remain of course, as there is a tradeoff between the amount of sensitive information that she allows the device to release and the accuracy and efficiency of some functionalities.

Aknowledgments Work partially supported by PRIN "Security Horizons".

References

1. M. S. Alvim, A. Scedrov, and F. B. Schneider. When not all bits are equal: Worth-based information flow. In *POST*. Springer, 2014.
2. T. Amtoft and A. Banerjee. A logic for information flow analysis with an application to forward slicing of simple imperative programs. *Science of Compututer Programming*, 64:3–28, 2007.
3. AppBrain. Adnetwork stats. http://www.appbrain.com/stats/libraries/ad.
4. S. Arzt, S. Rasthofer, and al. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI*. ACM, 2014.
5. A. Askarov and A. Myers. A semantic framework for declassification and endorsement. ESOP. Springer, 2010.
6. S. Cavadini. Secure slices of insecure programs. In *ASIACCS*. ACM Press, 2008.
7. A. Chaudhuri. Language-based security on android. In *PLAS*. ACM, 2009.
8. A. Cortesi, G. Costantini, and P. Ferrara. A survey on product operators in abstract interpretation. *EPTCS*, 129:325–336, 2013.
9. G. Costantini, P. Ferrara, and A. Cortesi. Static analysis of string values. In *ICFEM*. Springer, 2011.
10. P. Cousot and R. Cousot. Abstract interpretation: Past, present and future. In *CSL-LICS*. ACM, 2014.
11. P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL*. ACM, 2011.
12. D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19:236–243, 1976.
13. C. Dwork. Differential privacy: A survey of results. In *TAMC*. Springer, 2008.
14. H. Ebadi, D. Sands, and G. Schneider. Differential privacy: Now it's getting personal. In *POPL*. ACM, 2015.
15. W. Enck, P. Gilbert, and al. Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones. *Comm. of the ACM*, 57(3):99–106, 2014.
16. P. Ferrara. Generic combination of heap and value analyses in abstract interpretation. In *VMCAI*. Springer, 2014.
17. R. Halder, M. Zanioli, and A. Cortesi. Information leakage analysis of database query languages. In *SAC*. ACM, 2014.
18. 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.
19. P. Hornyack, S. Han, J. Jung, S. E. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *CCS*. ACM, 2011.

20. M. N. Krohn and E. Tromer. Noninterference for a practical DIFC-based operating system. In *IEEE S&P*. IEEE, 2009.
21. B. Li. Analyzing information-flow in java program based on slicing technique. *SIGSOFT Softw. Eng. Notes*, 27:98–103, 2002.
22. A. Lochbihler and G. Snelting. On temporal path conditions in dependence graphs. *Journal of Automated Software Engineering*, 16:263–290, 2009.
23. F. Logozzo. Class invariants as abstract interpretation of trace semantics. *Computer Languages, Systems & Structures*, 35:100–142, 2009.
24. L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *CCS*. ACM, 2012.
25. S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *PLDI*. ACM, 2008.
26. A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
27. A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP*. ACM, 1997.
28. A. Nanevski, A. Banerjee, and D. Garg. Dependent type theory for verification of information flow and access control policies. *ACM TOPLAS*, 35(2):6:1–6:41, 2013.
29. I. Omoronyia, L. Cavallaro, and al. Engineering adaptive privacy: on the role of privacy awareness requirements. In *ICSE*. IEEE / ACM, 2013.
30. F. Pottier and V. Simonet. Information flow inference for ml. *ACM Transactions on Programming Languages and Systems*, 25:117–158, 2003.
31. S. Rasthofer, E. Lovat, and E. Bodden. Droid force: Enforcing complex, data-centric, system-wide policies in android. In *ARES*, 2014.
32. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21:5–19, 2003.
33. A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17:517–548, 2009.
34. 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.
35. S. F. Smith and M. Thober. Refactoring programs to secure information flows. In *PLAS*. ACM, 2006.
36. S. Spiekermann and L. F. Cranor. Engineering privacy. *IEEE Trans. Software Eng.*, 35(1):67–82, 2009.
37. 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.
38. O. Tripp, P. Ferrara, and M. Pistoia. Hybrid security analysis of web javascript code via dynamic partial evaluation. In *ISSTA*. ACM, 2014.
39. O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. Taj: effective taint analysis of web applications. In *PLDI*. ACM, 2009.
40. O. Tripp and J. Rubin. A bayesian approach to privacy enforcement in smartphones. In *USENIX Security*, 2014.
41. X. Xiao, N. Tillmann, M. Fähndrich, J. de Halleux, and M. Moskal. User-aware privacy control via extended static-information-flow analysis. In *ASE*. ACM, 2012.
42. M. Zanioli, P. Ferrara, and A. Cortesi. Sails: static analysis of information leakage with sample. In *SAC*. ACM, 2012.