DAPA: Degradation-Aware Privacy Analysis of Android Apps

Gianluca Barbon, Agostino Cortesi, Pietro Ferrara, Enrico Steffinlongo

DAPA: Degradation-Aware Privacy Analysis of Android Apps

Gianluca Barbon[1], Agostino Cortesi[2], Pietro Ferrara[3], and Enrico Steffinlongo[2]

[1] Université Grenoble Alpes - Inria - LIG, Grenoble, France
gianluca.barbon@inria.fr
[2] Università Ca' Foscari, Venezia, Italy
{cortesi, enrico.steffinlongo}@unive.it
[3] Julia Srl, Verona, Italy
pietro.ferrara@juliasoft.com

Abstract. When installing or executing an app on a smartphone, we grant it access to part of our (possibly confidential) data stored in the device. Traditional information-flow analyses aim to detect whether such information is leaked by the app to the external (untrusted) environment. The static analyser we present in this paper goes one step further. Its aim is to trace not only if information is possibly leaked (as this is almost always the case), but also how relevant such a leakage might become, as an under- and over-approximation of the actual degree of values degradation. The analysis captures both explicit dependences and implicit dependences, in an integrated approach. The analyser is built within the Abstract Interpretation framework on top of our previous work on datacentric semantics for verification of privacy policy compliance by mobile applications. Results of the experimental analysis on significant samples of the DroidBench library are also discussed.

1 Introduction

Mobile applications have access to a large variety of confidential information like geographical data, and user identifiers (e.g., IMEI and phone number). Often the access to this sensitive data is essential for the functionality of the mobile app: a navigation system needs access to the current position of the user, while a photo editor accesses the picture gallery of the user. In addition, the business model might exploit this confidential information for contextual advertisement. On the other hand, a malicious app might exploit confidential information to capture sensitive data. For instance, the user might be tracked by recording and leaking the location and identifier of the mobile device.

The current Android model guards the access to sensitive data through permissions. For instance, an app should obtain the `ACCESS_FINE_LOCATION` permission in order to access precise geographical information. Therefore, each Android app has to list the permissions it needs in the manifest. The user is then asked to accept this list before installing the app or during the first execution of the app. However, this model allows an app to get only full or no access to a resource, and it does not take into account how the resource is accessed, and the information manipulated. In particular, the

application might *degrade* the confidential information before leaking it. For instance, an app performing contextual advertisement exposing to the advertisement engine only the zip code of the user (instead of the full location) together with the user identifier. Degradation-unaware analysis would conservatively consider that the full location could be leaked, and the user could be precisely tracked.

Therefore, degradation-awareness is crucial to precisely infer what kind of and how much sensitive information is accessed, manipulated, and possibly leaked in a mobile app. In this scenario, we introduce a novel degradation-aware static analysis based on the abstract interpretation framework. Our approach tracks both explicit and implicit flows of information as well as the degradation levels of operators applied to the confidential data. We implemented our system in a prototype and applied to some representative examples taken from the DroidBench test suite [19]. Our experimental results show the practical interest of our solution.

The rest of the paper is structured as follows. In the rest of this Section, we introduce two motivating examples. Section 2 discusses the related work, while Section 3 formalizes the language, and the concrete and abstract semantics of our approach. The architecture of our tool is then described in Section 4, and Section 5 presents the experimental results. Finally, Section 6 concludes.

1.1 Motivating Examples

Consider the motivating example in Fig. 1, a simplified version of the `ImplicitFlow1` test case from the DroidBench application set [19], an open source standard benchmark suite for information flow analyses of Android apps. This set has been created and maintained by the Secure Software Engineering Group of the Technische Universität Darmstadt. This program reads the device identifier (IMEI), and leaks it after some obfuscation steps. Obfuscation is performed by applying functions `obfuscateIMEI` and `copyIMEI`. Both the functions contain loops that are using the data derived from the IMEI as condition. This generates implicit flows which partially reveal confidential information about the original IMEI. Furthermore, it is interesting to notice that the functions and operators applied to the IMEI in the two methods are obfuscating it in different way, thus releasing implicitly different quantities of information.

In Fig. 2 another motivating example is shown. The program reads the IMEI and a user password. Then it hashes the password and it uses it as key for the encryption of the IMEI. Finally, the encrypted IMEI is explicitly released. In this example we can notice that, even if the program is leaking the password and the IMEI, it would not be possible to extract any sensitive information from the released values. Indeed the obfuscation steps performed through the `hash` and the `encrypt` operators make the reconstruction of the original values from the leaked ones hardly feasible.

Both examples show the need of a sound and precise analysis able to track (i) implicit flows and (ii) how the confidential information is obfuscated, by collecting the operators and functions that are applied to the confidential datum.

```
class ImplicitFlow1 extends Activity {
  static String obfuscateIMEI(String imei){
    String result, tmp;
    int idx;
    String[] array;
    result = "";
    idx = 0;

    array = toCharArray(imei);
    while (idx < stdlib.length(imei)) {
      tmp = array[idx];
      if (tmp == "0")
        result = result ++ "a";
      elif (tmp == "1")
        result = result ++ "b";
      elif (tmp == "2")
        result = result ++ "c";
      elif (tmp == "3")
        result = result ++ "d";
      elif (tmp == "4")
        result = result ++ "e";
      elif (tmp == "5")
        result = result ++ "f";
      elif (tmp == "6")
        result = result ++ "g";
      elif (tmp == "7")
        result = result ++ "h";
      elif (tmp == "8")
        result = result ++ "i";
      elif (tmp == "9")
        result = result ++ "j";
      else
        skip;
      idx = idx + 1;
    }
    return result;
  }

  static String copyIMEI(String imei){
    //ASCII values for integer: 48-57

    String[] imeiAsChar, newOldIMEI;
    String res;
    int[] numbers;
    int idx;
    idx = 0;
    numbers = new int[58];
    while (idx < 58) {
      numbers[idx] = idx;
      idx = idx + 1;
    }

    imeiAsChar = toCharArray(imei);
    newOldIMEI = new String[18];
    idx = 0;
    while (idx < len(imeiAsChar)) {
      int tmp;
      tmp = numbers[stdlib.strToInt(imeiAsChar[idx])];
      newOldIMEI[idx] = stdlib.intToString(tmp);
      idx = idx + 1;
    }
    res = "";
    idx = 0;
    while (idx < len(newOldIMEI)) {
      res = res ++ newOldIMEI[idx];
      idx = idx + 1;
    }
    return res;
  }

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

  //@Override
  static void onCreate(Bundle savedInstanceState) {
    String imei;
    String obfuscatedIMEI;
    imei = TelephonyManager.getDeviceId(); //source
    obfuscatedIMEI = obfuscateIMEI(imei);
    writeToLog(obfuscatedIMEI);

    obfuscatedIMEI = copyIMEI(imei);
    writeToLog(obfuscatedIMEI);
  }
}
```

Fig. 1: `ImplicitFlow1`

```
class ObfuscatedFlow extends Activity {

  static void onCreate() {
    String imei;
    String pwd;

    imei = TelephonyManager.getDeviceId();
    pwd = stdlib.hash(readlib.readUsrPwd("5v6QewbIOIMh"));
    log( stdlib .encrypt(imei, pwd));
  }
}
```

Fig. 2: `ObfuscatedFlow`

2 Related Works

Static [21] and dynamic [12] taint analyses have been deeply investigated to enforce integrity and confidentiality properties. The main idea of this approach is to check if information coming from a source (e.g., the input of the user or the method providing the IMEI of the device) flows into a sink (e.g., the execution of a SQL query or an internet connection) without being sanitized (e.g., removing or modifying special characters or encrypting it). Taint analysis has been widely applied to Android app as well. Flowdroid [1] models precisely the Android app lifecycle, and it performs a precise static taint analysis to discover leakages of information. Taintdroid [12] is instead a precise dynamic taint analysis with a low overhead. Amandroid [24] builds up a precise interprocedural call graph and data dependence graph, and it provides a framework to develop security analyses for Android apps. However, it can detect only explicit flows, and therefore it is not expressive enough to support our approach. Similarly, DroidSafe [14] proposes an accurate static information flow analysis, and HornDroid [6] introduces a fast and precise java bytecode analysis, but they do not track implicit flows. Taint analysis can track both implicit and explicit information flows, but it propagates only one bit of information (tainted or not). Instead, our approach tracks semantic information on how confidential data is processed and degraded. Implicit flows have been treated in [23, 7, 2, 15], but all these works are related to browsers vulnerabilities and focus on Java Script, while we apply the implicit flow notion to the Android environment. Instead, [26] tracks impicit flows on Java programs, but it does not consider degradation operators.

Various approaches have extended standard taint analysis to track more precise information for mobile software. MorphDroid [13] formalizes and implements a precise semantic analysis that infers what specific parts of confidential information are leaked (e.g., the zip code of the current location). However, it requires to manually define the semantics of degradation methods as well as tailored representation of each information of interest (e.g., IMEIs and locations), while our approach is agnostic on the type of information we deal with. BayesDroid [22] dynamically detects information leaks through Bayesian reasoning, that is, by comparing confidential data with leaked values. If the similarity among these values is above a given threshold, then BayesDroid infers that confidential data is leaked. While this approach is quite more efficient than existing taint tracking, it does not track how confidential information is degraded. Another dynamic approach is represented by AppIntent [25], a tool that records all GUI events and asks a security analyst if the data computed through a sequence of GUI events can be leaked.

A different approach was studied by Quantitative Information Flow [17]. Instead of tracking taints, this approach is aimed at inferring the quantity of information leaked by a program. On the one hand, we share with this field of research the belief that is crucial, especially for mobile applications, to track precisely the amount of information that is leaked. On the other hand, our approach targets the sequence of degradation operations applied to confidential data rather than an estimation of the quantity carried on by a value.

An orthogonal field of research has been the development of security oriented specification languages [20] to cover a large variety of aspects (e.g., access control). Some of these languages like [11] were focused on confidentiality properties. However,

these languages do not take into account how values are transformed and degraded, while this is the main focus of our work.

In our previous work [3], we introduced an information flow analysis that tracked the bit (i.e., quantity) of confidential information contained by each variable in a program. Instead, in this work we take a rather different approach by collecting the degradation operators (rather than a precise quantity) applied to the information stored in each variable. In this way, we overcome several limits of our previous solution, and in particular we track the implicit flow of strict equalities comparisons.

3 Concrete and Abstract Semantics

3.1 Syntax

As said in the Introduction, the target of our analyser are Android applications. For the sake of clarity, we will introduce our approach by restricting the view on a basic imperative language, supporting arithmetic, boolean and textual expressions, and arrays. Following [8], the formalization is focused on three types of data: strings ($s \in \mathbb{S}$), integers ($n \in \mathbb{Z}$) and Booleans ($b \in \mathbb{B}$). String, integer, and Boolean expressions are respectively denoted by *sexp*, *nexp*, and *bexp*. ℓ is used to represent (possibly sensitive) data-store entries, and *lexp* denotes label expressions. For instance, string expressions are defined by: $sexp ::= s \mid sexp_1 \circ sexp_2 \mid enc(sexp, k) \mid pre(sexp, k) \mid hash(sexp) \mid read(lexp)$, where $\circ$ represents concatenation, enc the encryption of a string with a key k, pre the prefix substring of $sexp$ of length k, $hash$ 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 explicit and implicit data sources of a specific expression. Formally, an atomic data expression *adexp* is a set of elements $\langle \ell, L_{dir}, D_{dir}, L_{imp}, D_{imp} \rangle$ where:

- `Lab` is the (finite) set of labels corresponding to possibly sensitive information sources stored in the device;
- $\ell \in$ `Lab`;
- $L_{dir} = \{(op_j, \ell'_j) : j \in J\}$, says that the datum corresponding to label ℓ has been combined with data corresponding to labels ℓ'_j through operators op_j to get the actual value of the expression.
- $D_{dir} = \{(op_j, v_j) : j \in J\}$ says that the actual value of the expression is obtained from the datum corresponding to label ℓ by applying the operator op_j with values belonging to the set v_j.
- $L_{imp} = \{(op_j, \ell'_j) : j \in J\}$, says that the actual value of the expression implicitly depends on the datum corresponding to label ℓ combined with data corresponding to labels ℓ'_j through the operator op_j .
- $D_{imp} = \{(op_j, v_j) : j \in J\}$ says that the actual value of the expression implicitly depends on the datum corresponding to label ℓ by applying the operator op_j with values belonging to the set v_j.

The set of atomic data expressions is defined by: $\mathbb{D} = \{\langle \ell_i, L^i_{dir}, D^i_{dir}, L^i_{imp}, D^i_{imp}\rangle : i \in I \subseteq \mathbb{N}, \ell_i \in \texttt{Lab}, L^i_k \subseteq \wp(\texttt{Op} \times \texttt{Lab}), D^i_k \subseteq \wp(\texttt{Op} \times \texttt{V})\}$, where Lab is the set of labels, and Op is the set of operators, and V contains sets of uniform values (integer intervals, sets of string , etc.). For $\ell \in \texttt{Lab}$, we denote by $\hat{\ell}$ the (constant) value stored in ℓ.

An environment relates variables to their values as well as to the set of atomic data expressions. Formally, $\Sigma = \Phi \times \Psi$, where (i) $\Phi : \mathbf{Var} \longrightarrow (\mathbb{Z} \cup \mathbb{S} \cup \{\text{true}, \text{false}\})$ is the usual environment that tracks value information, and (ii) $\Psi : \mathbf{Var} \longrightarrow \wp(\mathbb{D})$ maps local variables in $\mathbf{Var}$ to a set $\{\langle \ell_i, L^i_{dir}, D^i_{dir}, L^i_{imp}, D^i_{imp}\rangle : i \in I\}$. The special symbol $\star$ represents data coming from the user input and from the constants of the program.

Observe that the definition above refines [3], by introducing the explicit and implicit degradation sets that allow to keep track of the values the operators combine with the labels in Lab.

3.3 Concrete Semantics

We denote by $S_N : Nexp \times \Sigma \to \mathbb{Z}$, $S_S : Sexp \times \Sigma \to \mathbb{S}$, and $S_B : Bexp \times \Sigma \to \{\text{true}, \text{false}\}$ the standard concrete evaluations of numerical, string, and Boolean expressions. In addition, $S_L : Lexp \times \Sigma \to \texttt{Lab}$ returns a data label given a label expression. An array c of lenght n is represented by $n+1$ variables: $c_{length}, c[0], , \ldots c[n-1]$ where c_{length} stores the value of the lenght of the array c.

The semantics of expressions on atomic data $S_A : sexp \times \Sigma \to \mathbb{S} \times \wp(\mathbb{D})$ is described in Figure 1, for some basic unary and binary operators (like array selection, encryption *enc* and string prefix *pre* operators). Similar rules for numerical and boolean expressions are omitted here for the sake of space. Observe that the only new implicit flow is introduced when evaluating an array element, as the expression yielding the index may carry information that implicitly flows when accessing the corresponding array element.

The operator $\uplus$ is defined on the degradation elements as follows: $D_1 \uplus D_2 = \{(op, S_1 \cup S_2) : (op, S_1) \in D_1, (op, S_2) \in D_2\}$.

The operator $\diamond$ allows to inherit implicit dependences. Let $A = (a, \{\langle \ell^1_i, L^{1i}_{dir}, D^{1i}_{dir}, L^{1i}_{imp}, D^{1i}_{imp} : i \in I\}\rangle)$ and $B = (b, \{\langle \ell^2_j, L^{2j}_{dir}, D^{2j}_{dir}, L^{2j}_{imp}, D^{2j}_{imp}\rangle : j \in J\})$. $A \diamond B$ captures the fact that the expression represented by A implicitely reveals data contained in the expression represented by B. Formally, $A \diamond B = (a, \{\langle \ell^1_i, L^{1i}_{dir}, D^{1i}_{dir}, \bigcup_{j\in J}(L^{2j}_{dir} \cup L^{2j}_{imp}) \cup L^{1i}_{imp}, \biguplus_{j\in J}(D^{2j}_{dir} \uplus D^{2j}_{imp}) \uplus D^{1i}_{imp}\rangle : i \in I\}$.

Given a statement c, we denote by $\texttt{Def}(c)$ the set of variable that are assigned in the statement c.

The (concrete) semantics of statements is depicted in (Fig. 3). Observe that implicit flows are introduced in correspondence of *if* and *while* statements and arrays.

Example 1. Consider the following program:

(1) $x = read(\ell)$;
(2) $input(y)$;
(3) if $(x < y)\{$
(4) $\quad x = x + y;\}$
(5) $z = x + 1$;

$$
\begin{array}{rl}
S_A[\![s]\!](v,a) & = (S_S[\![s]\!](v), \{\langle \star, \emptyset, \emptyset, \emptyset, \emptyset \rangle\}) \\
S_A[\![n]\!](v,a) & = (S_N[\![n]\!](v), \{\langle \star, \emptyset, \emptyset, \emptyset, \emptyset \rangle\}) \\
S_A[\![x]\!](v,a) & = (v(x), a(x)) \\
S_A[\![read(lexp)]\!](v,a) & = \text{let } \ell = S_L[\![lexp]\!](v,a) \text{ in } (\hat{\ell}, \{\langle \ell, \emptyset, \emptyset, \emptyset, \emptyset \rangle\}) \\
S_A[\![\, c[nexp] \,]\!](v,a) & = \text{let } (n,d) = S_A[\![nexp]\!](v,a) \text{ in } (v(c[n]), a(c[n]) \diamond d) \\
S_A[\![enc(sexp,n)]\!](v,a) & = \text{let } (t, \{\langle \ell_i, L^i_{dir}, D^i_{dir}, L^i_{imp}, D^i_{imp} \rangle : i \in I\}) = S_A[\![sexp]\!](v,a) \text{ in} \\
& \quad (enc(t,n), \\
& \quad \{\langle \ell_i, L^i_{dir} \cup \{(enc, \ell_i)\}, D^i_{dir} \uplus \{(enc, \{n\})\}, L^i_{imp}, D^i_{imp} \rangle : i \in I\}) \\
S_A[\![sexp_1 \circ sexp_2]\!](v,a) & = \text{let } (t_1, \{\langle \ell^1_i, L^{1i}_{dir}, D^{1i}_{dir}, L^{1i}_{imp}, D^{1i}_{imp} \rangle) : i \in I\}) = S_A[\![sexp_1]\!](v,a) \text{ and} \\
& \quad \text{let } (t_2, \{\langle \ell^2_j, L^{2j}_{dir}, D^{2j}_{dir}, L^{2j}_{imp}, D^{2j}_{imp} \rangle) : j \in J\}) = S_A[\![sexp_2]\!](v,a) \text{ in} \\
& \quad (t_1 \circ t_2, \\
& \quad \bigcup_{i \in I, j \in J} (\{\langle \ell^1_i, L^{1i}_{dir} \cup \{(\circ, \ell^2_j)\}, D^{1i}_{dir} \uplus \{(\circ, \{t_2\})\}, L^{1i}_{imp}, D^{1i}_{imp} \rangle\} \cup \\
& \qquad \{\langle \ell^2_j, L^{2j}_{dir} \cup \{(\circ, \ell^1_i)\}, D^{2j}_{dir} \uplus \{(\circ, \{t_1\})\}, L^{2j}_{imp}, D^{2j}_{imp} \rangle\})) \\
S_A[\![pre(sexp,n)]\!](v,a) & = \text{let } (t, \{\langle \ell_i, L^i_{dir}, D^i_{dir}, L^i_{imp}, D^i_{imp} \rangle : i \in I\}) = S_A[\![sexp]\!](v,a) \text{ in} \\
& \quad (pre(t,n), \\
& \quad \{\langle \ell_i, L^i_{dir} \cup \{(pre, \ell_i)\}, D^i_{dir} \uplus \{(pre, \{n\})\}, L^i_{imp}, D^i_{imp} \rangle : i \in I\})
\end{array}
$$

Table 1: Semantics of Textual Expressions on Atomic Data

$$
\begin{array}{rl}
S[\![x := sexp]\!](v,a) & = \text{let } (t,d) = S_A[\![sexp]\!](v,a) \text{ in} \\
& \quad (v[x \mapsto t], a[x \mapsto d]) \\
S[\![\, c[nexp] := sexp \,]\!](v,a) & = \text{let } (t,d_1) = S_A[\![sexp]\!](v,a) \text{ and let } (n,d_2) = S_A[\![nexp]\!](v,a) \text{ in} \\
& \quad (v[c[n] \mapsto t], a[c[n] \mapsto d_1 \diamond d_2]) \\
S[\![input(x)]\!](v,a) & = \text{let } v_{read} \text{ be the input value in} \\
& \quad (v[x \mapsto v_{read}], a[x \mapsto \{\langle \star, \emptyset, \emptyset, \emptyset, \emptyset \rangle\}]) \\
S[\![skip]\!](v,a) & = (v,a) \\
S[\![send(sexp)]\!](v,a) & = (v,a) \\
S[\![c_1; c_2]\!](v,a) & = S[\![c_2]\!](S[\![c_1]\!](v,a)) \\
S[\![\text{if } b \text{ then } c_1 \text{ else } c_2]\!](v,a) & = \text{let } (v_1,a_1) = S[\![c_1]\!](v,a) \text{ and let } (v_2,a_2) = S[\![c_2]\!](v,a) \text{ in} \\
& \quad \begin{cases} (v_1, a_1') \textit{ if } S_A[\![b]\!](v,a) = (\text{true}, d) \\ (v_2, a_2') \textit{ if } S_A[\![b]\!](v,a) = (\text{false}, d) \end{cases} \\
& \quad \text{where } \begin{cases} a_1'(x) = a_1(x) \diamond d & \text{if } x \in \texttt{Def}(c_1) \\ a_1'(x) = a_1(x) & \text{otherwise} \end{cases} \\
& \quad \text{and } \begin{cases} a_2'(x) = a_2(x) \diamond d & \text{if } x \in \texttt{Def}(c_2) \\ a_2'(x) = a_2(x) & \text{otherwise} \end{cases} \\
S[\![\text{while } c_1 \text{ do } c_2]\!](v,a) & = S[\![\text{ if } (c_1) \text{ then } (c_2; \text{ while } c_1 \text{ do } c_2) \text{ else } \textit{skip} \,]\!](v,a)
\end{array}
$$

Fig. 3: Concrete Semantics of Statements

By applying the rules above, assuming that $\hat{\ell} = 3$ and that the input value assigned to y is 4, we get:

$$
\begin{array}{rl}
x_{(1)} & \mapsto (3, \{\langle \ell, \emptyset, \emptyset, \emptyset, \emptyset \rangle\}) \\
y_{(2)} & \mapsto (4, \{\langle \star, \emptyset, \emptyset, \emptyset, \emptyset \rangle\}) \\
(x < y)_{(3)} & \mapsto (\text{true}, \{\langle \ell, \{(<, \star)\}, \{(<, \{4\})\}, \emptyset, \emptyset \rangle, \langle \star, \{(<, \ell)\}, \{(<, \{3\})\}, \emptyset, \emptyset \rangle\} \\
(x + y)_{(4)} & \mapsto (7, \{\langle \ell, \{(+, \star)\}, \{(+, \{4\})\}, \emptyset, \emptyset \rangle, \{\langle \star, \{(+, \ell)\}, \{(+, \{3\})\}, \emptyset, \emptyset \rangle\} \\
z_{(5)} & \mapsto (8, \{\langle \ell, \{(+, \star)\}, \{(+, \{1,4\})\}, \{(<, \star), (<, \ell)\}, \{(<, \{3,4\})\} \rangle, \\
& \qquad \langle \star, \{(+, \ell)\}, \{(+, \{1,3\})\}, \{(<, \star), (<, \ell)\}, \{(<, \{3,4\})\} \rangle\}
\end{array}
$$

3.4 Abstract Semantics

By following the Abstract Interpretation framework, in order to lift the concrete semantics to an abstract semantics, suitable abstractions of the domains of concrete values should be

given, as well as operators on such abstractions that safely over-approximate the effects of the corresponding concrete ones. In our actual implementation, numerical values are abstracted in the lattice of `Intervals` [18], while textual values are abstracted by the `Prefix` domain [9, 10]. The abstract semantics of expressions and statements strictly follows the concrete one, with the usual exceptions: (1) in the evaluation of the conditional and iterative statements the least upper bound operator is applied when the truth value of the conditional expression cannot be inferred, and (2) a threshold widening operator is applied on intervals when evaluating while loops to guarantee termination of the analysis, as the domain of intervals does not satisfy the ascending chain condition.

The least upper bound operator of abstract atomic data is $\sqcup$ [8]. Abstraction and concretization functions are inherited from [8]. The $join$ operator is used to define the least upper bound of abstract values (numerical, boolean and string). The abstract semantics of statements are depicted in (Fig. 4). For simplicity, the abstract semantics of expressions are not explicitly described, since they corresponds to the concrete ones lifted to the abstract environment. Only the abstract semantic for the array *get* is defined here:

$$S^a_A[\![\, c[nexp] \,]\!](v^a, a^a) = \text{let } (n^a, d^a) = S_A[\![nexp]\!](v^a, a^a) \text{ in } \\ \left(\bigsqcup_i v^a(c[i]), \left(\bigsqcup_i a^a(c[i])\right) \diamond d^a\right) \mid i \in \gamma(n^a) \wedge i \in [0, \ldots, c_{length}]$$

Given an abstract element $(\tilde{a}, \{\langle \ell_i, \tilde{L}^i_{dir}, \tilde{D}^i_{dir}, \tilde{L}^i_{imp}, \tilde{D}^i_{imp}\rangle : i \in I\})$, it represents concrete expressions whose value is represented by $\tilde{a}$, and that may contain fingerprints of values stored in ℓ_i. An over-approximation of the operations and values under which such (direct or implicit) fingerprints may be hidden in that value are collected in the last four components.

$$\begin{array}{rl}
S^a[\![x := sexp]\!](v^a, a^a) = & \text{let } (t^a, d^a) = S^a_A[\![sexp]\!](v^a, a^a) \text{ in} \\
& (v^a[x \mapsto t^a], a^a[x \mapsto d^a]) \\
S^a[\![\, c[nexp] := sexp \,]\!](v^a, a^a) = & \text{let } (t^a, d^a_1) = S^a_A[\![sexp]\!](v^a, a^a) \text{ and} \\
& \text{let } (n^a, d^a_2) = S^a_A[\![nexp]\!](v^a, a^a) \text{ in} \\
& \left(\bigsqcup_i v(c[i] \mapsto t^a), \bigsqcup_i a(c[i] \mapsto (d^a_1 \diamond d^a_2))\right) \\
& \quad \mid i \in \gamma(n^a) \wedge i \in [0, \ldots, c_{length}] \\
S^a[\![input(x)]\!](v^a, a^a) = & \text{let } v^a_{read} \text{ be an abstract input value in} \\
& (v^a[x \mapsto v^a_{read}], a^a[x \mapsto \{\langle \star, \emptyset, \emptyset, \emptyset, \emptyset\rangle\}]) \\
S^a[\![skip]\!](v^a, a^a) = & (v^a, a^a) \\
S^a[\![send(sexp)]\!](v^a, a^a) = & (v^a, a^a) \\
S^a[\![c_1; c_2]\!](v^a, a^a) = & S[\![c_2]\!](S[\![c_1]\!](v^a, a^a)) \\
S^a[\![\text{if } b \text{ then } c_1 \text{ else } c_2]\!](v^a, a^a) = & \text{let } (v^a_1, a^a_1) = S^a[\![c_1]\!](v^a, a^a) \text{ and let } (v^a_2, a^a_2) = S^a[\![c_2]\!](v^a, a^a) \\
& \text{in } (join(v^a_1, v^a_2), a') \\
& \text{where } S^a_A[\![b]\!](v^a, a^a) = (t, d^a) \\
& \text{and } \begin{cases}
a'(x) = \{(a^a_1(x) \diamond d^a) \sqcup a^a_2(x)\} & \text{if } x \in \texttt{Def}(c_1) \\
a'(x) = \{a^a_1(x) \sqcup (a^a_2(x) \diamond d^a)\} & \text{if } x \in \texttt{Def}(c_2) \\
a'(x) = \{(a^a_1(x) \diamond d^a) \sqcup (a^a_2(x) \diamond d^a)\} & \\
\quad \text{if } x \in \texttt{Def}(c_1) \wedge x \in \texttt{Def}(c_2) & \\
a'(x) = \{a^a_1(x) \sqcup a^a_2(x)\} & \text{otherwise}
\end{cases} \\
S^a[\![\text{while } c_1 \text{ do } c_2]\!](v, a) = & fix(S[\![\text{ if } (c_1) \text{ then } (c_2; \text{ while } c_1 \text{ do } c_2) \text{ else } \textit{skip} \,]\!](v, a))
\end{array}$$

Fig. 4: Abstract Semantics of Statements

Example 2. Consider again the program of Example 1. Assume that the value of ℓ is bounded in the interval $[2, 4]$, and that the input value assigned to y is bounded by the interval $[3, 5]$. By applying the abstract semantics rules we get:

$$
\begin{aligned}
x_{(1)} &\mapsto ([2,4], \{\langle \ell, \emptyset, \emptyset, \emptyset, \emptyset \rangle\}) \\
y_{(2)} &\mapsto ([3,5], \{\langle \star, \emptyset, \emptyset, \emptyset, \emptyset \rangle\}) \\
(x < y)_{(3)} &\mapsto (\top, \{\langle \ell, \{(<, \star)\}, \{(<, [3,5])\}, \emptyset, \emptyset \rangle, \langle \star, \{(<, \ell)\}, \{(<, [2,4])\}, \emptyset, \emptyset \rangle\} \\
(x + y)_{(4)} &\mapsto ([5,9], \{\langle \ell, \{(+, \star)\}, \{(+, [3,5])\}, \emptyset, \emptyset \rangle, \{\langle \star, \{(+, \ell)\}, \{(+, [2,4])\}, \emptyset, \emptyset \rangle\} \\
z_{(5)} &\mapsto ([3,10], \{\langle \ell, \{(+, \star)\}, \{+, [1,5]\})\}, \{(<, \star), (<, \ell)\}, \{(<, [3,5]\})\}\rangle, \\
&\qquad \langle \star, \{(+, \ell)\}, \{(+, [1,4])\}, \{(<, \star), (<, \ell)\}, \{(<, [3,5])\}\rangle\}.
\end{aligned}
$$

Observe that the value of the expression $(x < y)_{(3)}$ is the top element of the boolean lattice $\{\bot, \text{true}, \text{false}, \top\}$ representing the fact that the abstraction does not allow to predict the truth value of such expression. Therefore, while computing $z_{(5)}$ both the branches of the conditional statement will be considered.

If we look at the abstract evaluation of $z_{(5)}$, we can say that under the mentioned initial conditions:

- the value of z at point (5) will definitely belong to the interval $[3, 10]$;
- the value of z may depend on either the value stored in ℓ, or from program constants, or from input values, but no other confidential information stored in $\ell_j \neq \ell$ may have affected the value of z, neither directly nor by implicit information flow;
- the only operator that might have been used to get the value of z out of ℓ is the numerical addition, with arguments that were never out of the interval $[1, 5]$;
- the possible implicit information flow from ℓ to the value of z can be only due to a strict ordering comparison with values in the interval $[3, 5]$.

4 The DAPA tool

We developed DAPA, a static analysis tool based on the abstract interpretation framework. We adopted the Scala language for the development of the tool. Our tool is able to analyse both explicit and implicit flows and obtain a set of Degradation Elements from an Android app source code (translated in our language). It also computes results of functions and expressions in the form of abstract values, in order to collect them through the Degradation Elements. One of the main strength of the analyser is the capability to collect conditions of *If* and *While* constructs as implicit statements. This allows to propagate the implicit information flow throughout the analysis and to check whether confidential labels are present in the form of implicit information in sink points.

The analyser is conceived in such a way to be modular and easily extensible in case of future improvements. We adopted Scala traits to mask the underlying implementation of the atomic data expression and lattices. This means that it is possible to modify the used abstraction by just providing an alternative implementation of the abstract types to the analyser, ensuring the modularity of the whole project. Standard join, meet and widening methods from the abstract interpretation framework are provided. In addition to these methods, a *union* function for atomic data expression is introduced, which allows to collect different behaviours for the over and under approximation. It performs

the meet for the under approximation and the join for the over approximation of the elements of the atomic data expression. Operators present in statements are collected through an update function, which ensures the insertion of the operator in the atomic data expressions of all the involved labels. This method implements the $\uplus$ operator defined in Section 3. The $\diamond$ operator is also implemented in order to produce implicit flows.

Three types of basic abstract values are implemented: boolean, numerical and string, following the ones defined in the theory and adopting solutions described in [4, 16, 18]. Also abstract values implementation is hidden through the use of common interface, ensuring modularity.

The output of DAPA consists in a list of *adexp* as defined in Section 3. This list of *adexp* highlights the degradation applied to every confidential label in the analysed app, by listing all the functions and operators applied to it, along with other labels involved as parameters. Specific information about the implicit flows will also appear in the output. A detailed explanation of the output is provided in Section 5.

Multiplicity of the Degradation Element In the tool the *Degradation Element* is extended with a multiplicity notion, in order to associate the times it appears in a loop, allowing to track the number of repetitions of the element. These ones are tracked through an abstract interval, allowing proper handling during loop widening. This will contain the abstract number of times that such operator has been evaluated, giving a useful information to be associated to the degradation of the current label. This allows to abstract the operators present inside the scope of the loop. We also introduce information regarding position in the code of the statement in order to have an unambiguous element. The tool is thus more precise with respect to the semantics.

Widening of While loops While loops are analysed through the use of widening from the abstract interpretation framework. A threshold (or guard) can be modified by the user in order to stop loop iterations and start widening of the remaining ones. The under approximation in the resulting expressions will contain the smallest possible number of iterations (possibly, no iterations at all). Instead, the over approximation will be the set of all the possible iterations (the maximum possible number of iterations) of the loop.

5 Results

In this Section we present a qualitative analysis of the DAPA tool. Quantitative evaluation has also been made, in order to evaluate performances.

5.1 Motivating Examples

Results of the analysis of the motivating example ImplicitFlow1, introduced in Section 1, are here described. A special *star* label is used to collect every degradation that does not belong to any labels. The IMEI_0 label is the only confidential label used in this example. It contains the IMEI value associated to a device.

```
Explicit: []
Implicit: [IMEI_0 ->
    S: [{(length_1, IMEI_0), (<_1, star), (toCharArray_1, IMEI_0), (==_2, star)}:
        {(length_1, IMEI_0), (<_1, star), (toCharArray_1, IMEI_0), (==_2, star)}]
    D: [[(length_1, ImplicitFlow1.java@10.22) -> ({"*"}, [1,1]),
        (<_1, ImplicitFlow1.java@10.20) -> ([0,+oo], [1,1]),
        (toCharArray_1, ImplicitFlow1.java@9.17) -> ({"*"}, [1,1]),
        (==_2, ImplicitFlow1.java@12.21) -> ({"0"}, [1,1])]:
        [(toCharArray_1, ImplicitFlow1.java@9.17) -> ({"*"}, [1,1]),
        (==_2, ImplicitFlow1.java@12.21) -> ({"0"}, [1,1]),
        (<_1, ImplicitFlow1.java@10.20) -> ([0,+oo], [1,1]),
        (length_1, ImplicitFlow1.java@10.22) -> ({"*"}, [1,1]),
        (==_2, ImplicitFlow1.java@22.23) -> ({"5"}, [1,1])]]]

Explicit: []
Implicit: [IMEI_0 ->
    S: [{(toCharArray_1, IMEI_0), (strToInt_1, IMEI_0), (<_1, star)}:
        {(toCharArray_1, IMEI_0), (strToInt_1, IMEI_0), (<_1, star)}]
    D: [[(toCharArray_1, ImplicitFlow1.java@53.22) -> ({"*"}, [1,1]),
        (strToInt_1, ImplicitFlow1.java@58.27) -> ({"*"}, [1,1]),
        (<_1, ImplicitFlow1.java@56.20) -> ([0,+oo], [1,1])]:
        [(toCharArray_1, ImplicitFlow1.java@53.22) -> ({"*"}, [1,1]),
        (strToInt_1, ImplicitFlow1.java@58.27) -> ({"*"}, [1,1]),
        (<_1, ImplicitFlow1.java@56.20) -> ([0,+oo], [1,1])]]]
```

Fig. 5: ImplicitFlow1 Results

Fig. 5 contains the output of the analysis of ImplicitFlow1. The upper part of the figure corresponds to the first release of the IMEI with method `writeToLog`, while the lower one corresponds to the second release. The two calls to this methods correspond to two different obfuscation methods usage, the `obfuscateIMEI` and the `copyIMEI` respectively. These two methods have different obfuscation powers. Our tool was able to track these differences by reporting the usage of different operators.

Results are split into two sections, *S* and *D*, which in turn are divided into two parts by a colon, for under and over approximation respectively. The *S* part contains the atomic data expression composed by couples $(operator, label)$, as defined in [3]. This allows to know which are the label that were combined through the related operator. The *D* part contains the degradation elements. It shows the operator, the position in the code, the abstract content and the number of iterations. Operators are annotated with an index. For instance, in the line 56 of the motivating example, the condition

```
idx < len(imeiAsChar)
```

will degradate IMEI by `< _1` of `idx`, that is in position 1 of the arguments of $<$. While `idx` will be degradated by `< _2` of imeiAsChar, but this is not listed in Fig. 5 since it belongs to the *star* label. The same holds for the *S* part. This allows to obtain an accurate explanation of the obfuscation. The position is composed of the name of the file, the number of row and the column in the code. Abstract values are related to the type of the label: numerical values are described through intervals, contained in square brackets, while strings are contained in braces. In this example, iterations are interval `[1,1]`, because there were no loops, thus no repeated elements. The user can in this way know which were the applied methods and operators used to obfuscate the IMEI.

Please notice that the only existing explicit flow is related to the special label *star* (not reported in the figure for the sake of space). This is because the value returned by the obfuscating methods does not explictly contains the IMEI label. On the other hand, this value contains implicit information about the IMEI label. This is correctly tracked by the DAPA analyser.

```
Explicit:
[IMEI_0 -> S: [{(encrypt_2, pwd_0)}:
               {(encrypt_2, pwd_0)}]
          D: [[(encrypt_2, main/resources/StmDegr.java@12.13) -> ({"*"}, [1,1])]:
              [(encrypt_2, main/resources/StmDegr.java@12.13) -> ({"*"}, [1,1])]],

  pwd_0 -> S: [{(hash_1, pwd_0), (encrypt_1, IMEI_0)}:
               {(hash_1, pwd_0), (encrypt_1, IMEI_0)}]
          D: [[(hash_1, main/resources/StmDegr.java@10.15) -> ({"*"}, [1,1]),
               (encrypt_1, main/resources/StmDegr.java@12.13) -> ({"*"}, [1,1])]:
              [(hash_1, main/resources/StmDegr.java@10.15) -> ({"*"}, [1,1]),
               (encrypt_1, main/resources/StmDegr.java@12.13) -> ({"*"}, [1,1])]]]
```

Fig. 6: ObfuscatedFlow Results

In a similar way, in Fig. 6 the analysis results for the example in Fig. 2 are presented. Since no implicit flow is produced, only the explicit one is reported. The atomic data expression part *S* shows that an hash operator is applied to the password label. Then this hashed password is used by the encrypt operator as key to obfuscate the IMEI label.

These examples show that our analysis is able to track every degradation operation applied to confidential data.

5.2 Quantitative Evaluation: the DroidBench test set

The DAPA analyser has been tested using a set of simple Android-like apps from the DroidBench application set [19]. Where needed, the original Java code has been modified in order to be recognised by the analyser, since it still lacks support for actual libraries and some typical Java elements. Similar considerations must be taken into account for the Android libraries. The table in Fig. 7 describes analyser results in terms of time performances and detected leakage (explicit and implicit). The computer used for tests execution was equipped with an Intel i5 450M processor and 8GB of DDR2 RAM.

Even if the code in test cases contains some small differences with the original ones (with the exceptions of ArrayCopy1 and ImplicitFLow2, that were heavily modified), the DAPA tool was able to discover correctly explicit or implicit leaks when present.

5.3 Results Discussion

A quantitative test comparison with other existing tools, such as FlowDroid, DroidSafe, BayesDroid, was not possible. This is because the testing step has been made on simple basic applications. Nevertheless, the results obtained by DAPA are richer from a qualitative point of view when comparing to other tools. This is because we are performing a

file name	time	explicit leakage	implicit leakage
ArrayAccess1	151ms	no	no
ArrayAccess2	212ms	no	no
ArrayCopy1*	200ms	yes	no
ArrayToString1	237ms	yes	no
DirectLeak1	166ms	yes	no
ImplicitFlow1	1838ms	no	yes
ImplicitFlow2*	300ms	no	yes
LoopExample1	488ms	yes	yes
LoopExample2	682ms	yes	yes
SourceCodeSpecific1	392ms	yes	no
StringToCharArray1	424ms	yes	yes
UnreachableCode	33ms	no	no

Fig. 7: DroidBench Results

privacy degradation aware analysis, while common tools are more focused on pure taint analysis. This means that DAPA collects all the operations applied to sensible data, when such data are released. Moreover, when no degradations are applied to the confidential data, DAPA returns in any case richer results, since it is also capable to track implicit flows.

6 Conclusions

We introduced a new static analyser for information flow analysis of Android apps, that captures both explicit and implicit leakage and support degradation awareness. Our preliminary experimental results show the effectiveness of this approach, and the modularity of the analyser allows to tune the accuracy and efficiency of the analysis by plugging in more or less sophisticated abstract domains.

Future improvements will consist in implementing objects in our language. A complete move to Java will be considered too, since it will introduce the possibility to analyse real Android app, without the need of conversions. Moreover, the evaluation of policies based on confidentiality and obfuscation notions [8], already captured by the current analyser, should be considered in the future.

Finally, we would also consider to reuse the bit quantity introduced in [3] in order to define a function able to compute the final exported explicit and implicit quantity as a result of the degradation. Since the definition of this function would have required a considerable research effort about operators information release, it was outside the scope of this work, and we planned it only as possible future improvement.

References

1. S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *PLDI*. ACM, 2014.
2. S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett. Vex: Vetting browser extensions for security vulnerabilities. In *USENIX Security*. USENIX Association, 2010.
3. G. Barbon, A. Cortesi, P. Ferrara, M. Pistoia, and O. Tripp. Privacy analysis of android apps: Implicit flows and quantitative analysis. In *CISIM*. Springer, 2015.
4. G. Bohlender and U. W. Kulisch. Definition of the arithmetic operations and comparison relations for an interval arithmetic. *Reliable Computing*, 15(1):36–42, 2011.
5. C. Braghin, A. Cortesi, and R. Focardi. Control flow analysis of mobile ambients with security boundaries. In *FMOODS*. Springer, 2002.
6. S. Calzavara, I. Grishchenko, and M. Maffei. Horndroid: Practical and sound static analysis of android applications by smt solving. In *EuroS&P*. IEEE, 2016.
7. R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. *SIGPLAN Not.*, 44(6):50–62, June 2009.
8. A. Cortesi, P. Ferrara, M. Pistoia, and O. Tripp. Datacentric semantics for verification of privacy policy compliance by mobile applications. In *VMCAI*. Springer, 2015.
9. G. Costantini, P. Ferrara, and A. Cortesi. Static analysis of string values. In *ICFEM*. Springer, 2011.
10. G. Costantini, P. Ferrara, and A. Cortesi. A suite of abstract domains for static analysis of string values. *Softw., Pract. Exper.*, 45(2):245–287, 2015.
11. F. Cuppens and R. Demolombe. A deontic logic for reasoning about confidentiality. In *DEON*. ACM, 1996.
12. W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI*, 2010.
13. P. Ferrara, O. Tripp, and M. Pistoia. Morphdroid: Fine-grained privacy verification. In *ACSAC*. ACM, 2015.
14. M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, , and M. Rinard. Information-flow analysis of android applications in droidsafe. In *NDSS*. ACM, 2015.
15. S. Just, A. Cleary, B. Shirley, and C. Hammer. Information flow analysis for javascript. In *PLASTIC*. ACM, 2011.
16. U. W. Kulisch. Complete interval arithmetic and its implementation on the computer. In *Numerical Validation in Current Hardware Architectures, Dagstuhl Seminar*, 2008.
17. S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *PLDI*. ACM, 2008.
18. A. Miné. *Weakly relational numerical abstract domains*. PhD thesis, École Polytechnique, Dec. 2004. http://www-apr.lip6.fr/ mine/these/these-color.pdf.
19. Secure Software Engineering Group - Ec Spride. DroidBench. http://sseblog.ec-spride.de/tools/droidbench/.
20. N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. In *S&P*. IEEE, 2009.
21. O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: Effective Taint Analysis of Web Applications. In *PLDI*, 2009.
22. O. Tripp and J. Rubin. A Bayesian Approach to Privacy Enforcement in Smartphones. In *USENIX Security*, 2014.
23. P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS*. The Internet Society, 2007.

24. F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *CCS*. ACM, 2014.
25. Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection. In *CCS*. ACM, 2013.
26. M. Zanioli, P. Ferrara, and A. Cortesi. SAILS: static analysis of information leakage with sample. In *SAC*. ACM, 2012.