Principles of Software Security Paper Summary:

*Privacy Analysis of Android Apps: Implicit Flows and Quantitative Analysis (Barbon, Cortesi, Ferrar, Pistoia, Tripp (2015))*

Jose Cambronero
November 23, 2015

Abstract
This summary paper provides an overview of the ideas presented in Barbon et al (2015), highlights some of the interesting points raised, and provides some slight suggestions for future work.

1 Motivation

1.1 Problem definition
The rising popularity of mobile applications, particularly in the Android space, along with increasing need of those applications to access user data (both personally and non-personally identifying information) has increased the need for ways to verify that applications make use of data according to appropriate policies.

While applications request that users approve their use of certain types of data (e.g. location)[1], there is a need to further verify that the source code of the application takes the necessary steps to ensure its use is appropriate.

In a prior paper[2], the authors introduced a data-centric approach to privacy policy verification, which in contrast to other approaches allows for a refined approach to data leaking. As highlighted in[1], a complete absence of data leakage would make many applications unusable. This paper focuses on developing a framework that captures leakage in implicit flows and quantifies the amount of information leaked.

1.2 Existing Work
As is usually the case, approaches to confidentiality analysis can be grouped into dynamic and static. In the case of the latter, [1] highlights examples such as TaintDroid[5], that monitors sensitive data used. Another interesting example is BayesDroid[8], which turns the privacy verification problem into a classification problem and creates a statistical classifier to address this. However, like other dynamic methods, these require that a) the application actually run (thus exposing any underlying dangers), b) is influenced by realized traces.

2 Overview of approach
[1] extends the work in [2], which presents a abstract interpretation framework to track and analyze data sources for operations involving "labels" from a device's datastore. The original framework combined explicit flows with confidentiality and obfuscation levels, properties which can roughly be described as : how confidential a given data element is considered, and how much a given operation has done to eliminate the ability to recover the original data element to which it was applied by observing the resulting element, respectively.

The current paper extends this work with implicit flows and provides a quantitative measure for how much data is leaked.
3 Concrete Analysis

3.1 Syntax

The paper defines a simple syntax covering basic operations over strings, integers and booleans. Additionally, the concept of labels is introduced. A label $l_i$ belongs to a set of labels $\text{Lab}$, which represent entries in a program’s datastore. The current paper [2] assumes the datastore is a read-only store, so as to avoid issues with aliasing. Given a program $p$, it has a datastore $C_p$ such that every entry $i$ is associated with a unique label $l_i$, which is not a program constant or user input.

3.2 Atomic Data Expression (with novel extensions)

An atomic data expression $\text{adexp}$ is the data structure that is used to keep track of the source of data used in calculating a given value. An $\text{adexp}$, as introduced in [2] as a set of elements of the form $(l_i, \{(o_j, l'_j) : j \in J\})$, which simply states that a value came from combining $l_i$ with a series of other labels, $l'_j$, using operations $o_j$. Furthermore, it is important to note that an $\text{adexp}$ is a set because as a value could come from interaction with multiple labels (i.e. multiple sources from the data store). This allows the framework to keep track of “where” a value came from.

The paper extends this with implicit flow information and quantitative information.

3.2.1 Implicit Flow extension

Rather than solely considering explicit flow information for an $\text{adexp}$ as in [2], the “source” of data can be broken down into a tuple of 2 sets: explicit and implicit flows. The authors define an extended $\text{adexp}$ $d$ to be of the form $d = (d_e, d_i)$, where the first provides the set of explicit $\text{adexp}$ and the second provides the set of implicit $\text{adexp}$. The implicit portion $d_i$ allows us to combine the information flow captured by implicit flows in conditional statements. For example, if we have a conditional with a test $c$ and a true branch $b_1$, and assume that the true branch is taken, the concrete extended $\text{adexp}$ would be composed of $(d_e(b_1), d_i(b_1) \cup d_e(c) \cup d_i(c))$, where we abuse notation a bit and create functions $d_e, d_i$ which extract the explicit and implicit portions of an $\text{adexp}$.

3.2.2 Quantitative extension

The second main idea introduced in the paper is the extension of the framework with quantitative information. Specifically, information is represented in binary form, so depending on its data type, a label $l_i$ will generate some value according to its size in memory. The idea is that this will help us determine the maximum amount of information that might have been leaked by an operation. This is akin to the work referenced from [7], which studies the quantity of information released by measuring the bits that are released when observing the execution of a program (e.g. a conditional that branches true, or false, releases 1 bit of information).
In the author’s formulation the data expression is now extended to \( d = (d_e, d_i, d_q) \), where \( d_q \) represents the quantitative information for implicit flows. Formally, \( d_q = \{ (l_k, q_k) : k \in J \} \), where \( J \) represents indices for implicit flows. The value \( q_k \) associated with a label \( l_k \) is updated for a statement that has implicit flows, for statements that don’t generate implicit flows \( q_k \) is passed along unchanged.

So for example in Section 3.2.1, the \( q_k \) for any labels \( l_i \) in the \( adexp \) for the conditional test (which generated the implicit flows) would be updated using a quantitative function \( \phi \) such that if a label \( l_i \) had a quantitative value \( q_i \) prior to the statement, it will have \( q_i + \phi(l_i) \) after.

The use of this extended atomic data expression (along with its suitable abstraction) represents the major contribution of this work building on [2].

Note that we elide the collecting semantics described in the paper for the environment, consisting out of a mapping for the atomic data expression and normal values (the traditional environment). However, we note that it is constructed inductively on a simple language’s syntax. These collecting semantics can then be used for static analysis in two-stages: a) analyzing flows, b) quantifying the information leaked in implicit flows.

4 Abstract Analysis

The abstraction technique provided assume a suitable abstraction for labels and normal values.

The authors build on the abstractions in [2] and extend them to handle implicit flows and quantification of the implicit flows, as appropriate for their extended \( adexp \). The gist of the idea is that the quantitative value is abstracted into intervals, representing the least and most amount of information that could be released through implicit flows. Meanwhile, implicit flows are abstracted similarly to the way explicit flows were abstracted in [2], which is to say: labels are abstracted using a label abstraction, and the set of \( \langle o, l_i \rangle \in L_i \) tuples identifying operation and labels that are combined to obtain values, are abstracted into under- and over-approximations of the set of operators applied to labels. We refer the reader to [1] for a detailed explanation of the abstraction function, but provide a rough sketch below.

**Definition Sketch 4.1.** An abstract function \( \alpha_Q \) takes a set \( \{ (l_i, k_i) \} \), where \( k_i \) might be the explicit/implicit flow information or the quantitative values, and abstracts into \( \{ (l_i^\wedge, k_i^{\wedge}), (l_i^\vee, k_i^{\vee}) \} \), where each element is abstracted with an appropriate function (e.g. labels use the provided label abstraction, normal values use the normal value abstraction provided, and quantitative information values abstract into interval min/max bounds) into the abstract domain \( Q \).

Since the paper focuses on the implicit flow extension and quantification of information, it elides the abstract semantics for expressions, since they don’t create implicit flows. The abstract semantics for the statements presented, however, are fairly straightforward. As usual, we consider all possible executions in control statements, so the abstract semantics for if-else and while statements appropriately join the possible explicit/implicit flows (respectively) and update the bounds of the implicit flow quantification with under/over-approximations of the updated quantification bounds.
5 Interesting points Expanded

5.1 Estimating Information Leaked in Loops

The approach to estimating information leaked inside loops is particularly interesting and reduces to a two-step solution. In the first step, the framework adds a simple counter that is increased in each iteration of the loop. The counter is analyzed using interval analysis, providing bounds on the number of times the loops executes. The second step involves analyzing the quantitative value of information leaked through implicit flows in each iteration of the loops. The lower/upper bounds on this information, combined with the upper bound on iterations, allows us to infer the lower/upper bounds that the entire construct could leak. So if our loop calculates a value using $l_i$ at each iteration, iterates 16 times and leaks 3 bit of information in each iteration, our abstraction becomes $\langle l_i, 3 \times \log_2(16), 3 \times \log_2(16) \rangle$, which allows us to quantify and analyze the information leaked. Each iteration test provides 1 bit of information, since we know if the conditional is true or false. We apply $\log_2$ and multiply by the number of bits released per iteration to arrive at a quantification of the information leaked through this implicit flow.

We can then decide if 12 bits is a sufficiently low number or if this is a risk. Additionally, the counting of bits should account for any “obfuscation” [2], such as encryption or hashing.

5.2 Performance and Example

The authors evaluate performance of their framework by using examples from DroidBench [6] and show that they are able to identify places with implicit flows, the source of those flows, and the amount of information released.

We consider a simplified example sketch inspired by their last 2 examples “ImplicitFlow 2” and “ImplicitFlow 3”.

```plaintext
1 x = read (l1)
2 y = user_input()
3 if (x == y)
4    then
5      print (‘yes’)
6    else
7      print (‘no’)
```

We skip the concrete analysis for length purposes

Abstract analysis and quantitative analysis:

- $x_1 : \langle \{(l_1, \emptyset), \emptyset\} \rangle$
- $(x == y)_3 : \langle \{(l_1, \{(==, \ast)\}), \{(==, \ast)\}) \rangle, \emptyset\rangle$
- print$_{5,7} : \langle \emptyset, \{(l_1, \{(==, \ast), (\neq, \ast)\}), \{(==, \ast), (\neq, \ast)\}) \rangle$)

The if-statement in line 3 creates an implicit flow at the print statements in lines 5/7. In this case, this exposes 1 bit of information (as we now know if the equality check is true or not), thus the abstract quantitative $qadexp$ associated with line 3 is $\langle l_1, 1, 1 \rangle$. 


6 Suggestions For Future Work/Improvements

6.1 Explicitly state steps for verification

The authors leave this implicit from prior work [2]. However, given that the extensions are meaningful, it stands to reason that their verification section should be at the very least duplicated here. As it stands, it is currently the reader’s responsibility to find the relevant prior reference and study it separately. An extension similar to the form presented in [2] might be helpful for the new reader:

Definition 6.1. Given the set of data source labels Lab, confidentiality/obfuscation lattices $S$ and $O$ for labels and operations, respectively, and a quantitative threshold for information leaked via implicit flows, a confidentiality policy is a tuple of $\pi = (\eta, \zeta, \phi, \kappa_{sc,max}, \kappa_{lc,min}, \kappa_{q,max})$, where $\eta, \zeta$ assign each label/operator a corresponding value from lattices $S$ and $O$, and $\phi$ calculates a quantification value used to update labels in statements with implicit flow. $\kappa_{sc,max}$ is a maximum confidentiality level, $\kappa_{lc,min}$ is a minimum obfuscation level, and $\kappa_{q,max}$ is a maximum amount of information leaked through implicit flows (measured in bits).

If a program $P$, terminates and generates a set $X$ of atomic data expressions, and $\forall x \in X : (sc_{min}, sc_{max}, lc_{min}, lc_{max}, q_{min}, q_{max}) : sc_{max} \sqsubseteq \kappa_{sc,max} \land \kappa_{lc,min} \sqsubseteq lc_{min} \land q_{max} \sqsubseteq \kappa_{q,max}$, then any actual execution of $P$ satisfies the policy $\pi$.

6.2 Statistical Integration: Combining with BayesDroid

In separate work[8], one of the authors implements a statistical approach to privacy analysis by creating a bayesian classifier that labels data released as “legitimate” (negative) and “illegitimate” (positive) (i.e. violating some normal use), given features of the program and data, such as similarity between the source and the released value.

[1] explicitly mentions that their framework allows identification of implicit flows missed by the statistical approach in [8] (where these instances were false negative classifications). It would be interesting to provide output from the static analysis proposed here in the form of features for the statistical learner. This might be able to reduce the false negative rate further and improve an existing dynamic analysis tool. In future work [8] references the advantage of using static analysis results as input, but doesn’t suggest using these specifically for implicit flow analysis.

References


