MorphDroid: Fine-grained Privacy Verification

Pietro Ferrara
IBM Thomas J. Watson
Research Center
pietroferrara@us.ibm.com

Omer Tripp
IBM Thomas J. Watson
Research Center
otripp@us.ibm.com

Marco Pistoia
IBM Thomas J. Watson
Research Center
pistoia@us.ibm.com

ABSTRACT

Mobile devices are rich in sensors, such as a Global Positioning System (GPS) tracker, microphone and camera, and have access to numerous sources of personal information, including the device ID, contacts and social data. This richness increases the functionality of mobile apps, but also creates privacy threats. As a result, different solutions have been proposed to verify or enforce privacy policies. A key limitation of existing approaches is that they reason about privacy at a coarse level, without accounting for declassification rules, such that the location for instance is treated as a single unit of information without reference to its many fields. As a result, legitimate app behaviors — such as releasing the user's city rather than exact address — are perceived as privacy violations, rendering existing analyses overly conservative and thus of limited usability.

In this paper, we present MorphDroid, a novel static analysis algorithm that verifies mobile applications against fine-grained privacy policies. Such policies define constraints over combinations of fine-grained units of private data. Specifically, through a novel design, MorphDroid tracks flows of fine-grained privacy units while addressing important challenges, including (i) detection of correlations between different units (e.g. longitude and latitude) and (ii) modeling of semantic transformations over private data (e.g. conversion of the location into an address).

We have implemented MorphDroid, and present a thorough experimental evaluation atop a comprehensive benchmark suite for Android static and dynamic analyses (DroidBench), as well as the 500 top-popular Google Play applications in 2014. Our experiments involve a spectrum of 5 security policies, ranging from a strict coarse-grained policy to a more realistic fine-grained policy that accounts for declassification rules. The experiment on DroidBench shows that MorphDroid achieves precision and recall scores of over 90%. The experiments on popular apps show that the gap between policies is dramatic, the most conservative policy yielding warnings on 171 of the applications (34%), and the more realistic policy flagging only 4 of the applications as misbehaved ($< 1\%$). In addition, MorphDroid exhibits good performance with an average analysis time of < 20 seconds, where on average apps consist of 1.4M lines of code.

ACSAC '15, December 07 - 11, 2015, Los Angeles, CA, USA

ISBN 978-1-4503-3682-6/15/12…$15.00

DOI: http://dx.doi.org/10.1145/2818000.2818037

1. INTRODUCTION

Mobile applications offer increasingly richer and more contextual functionality. The provided features and services are derived from various mobile-specific sources, such as the user's (i) current and past locations; (ii) persistent identity, established via the device and Subscriber Identity Module (SIM) identifiers; (iii) digital media, such as microphone and camera data; (iv) contacts; and (v) social networks. Access to these sources of information enables advanced functionality, but at the same time raises privacy threats.

The balance between privacy and functionality is subtle, as the user clearly wishes to maximize both. For example, it stands to reason that an e-commerce application with offerings and promotions based on the user's location would send the user's city or zip code to its server side. The privacy loss is marginal, and in return useful contextual functionality is enabled. On the other hand, sharing with the server side fine-grained location information, such as the the user's exact GPS coordinates, is unjustified.

Clearly, as the scenario above illustrates, understanding the granularity at which private information is released to external observers is crucial to determine whether the appropriate balance between privacy and functionality is enacted. Extracting this level of detail from the code of a mobile application requires fine-grained modeling of the *units* of private information that the application accesses, as well as their *flow* and *transformation* throughout the application (e.g., location conversion into an address, then release of the city).

Existing Approaches Privacy testing, verification and enforcement are all problems that have received considerable attention from the research community, resulting in effective solutions. Notable examples are FlowDroid [2], which performs taint-based privacy verification of Android applications; TaintDroid [6] and BayesDroid [20], which feature low-overhead real-time privacy enforcement by detecting attempts to release sensitive data; and XPrivacy[1], a privacy management system that prevents applications from leaking confidential data by restricting the data an application can access.

Existing solutions model confidential information at the coarse level of privacy APIs. As an illustration, the return value of a `getLastKnownLocation` call is modeled and tracked (typically via taint analysis) as a coarse unit of private information. The same is true of the International Mobile Equipment Identifier (IMEI), which is the result of `getDeviceId`. In reality, however, judgments about the legitimacy of information release are finer grained, as illustrated above. If an application derives and releases only the ZIP code from the `Location` object due to `getLastKnownLocation`, then the analysis should accurately track information at this level of granularity. The same consideration applies to an application that releases only the manufacturer information from the IMEI (rather than the entire IMEI, which would reveal the user's identity).

[1] http://xprivacy.eu

Our Approach This paper presents MorphDroid, a novel security framework for structured data-leakage detection that bridges the gap highlighted above. Through careful design, MorphDroid is able to track the flow of fine-grained privacy units, including conversions between private objects, efficiently and scalably.

The two main contributions of MorphDroid are the following:

1. MorphDroid supports a declarative interface to express fine-grained privacy policies. The user can specify privacy constraints as combinations of fine-grained units of private data whose release to external observers is authorized. As an illustration, the user may specify that an application is allowed to release the device ID together with either the longitude or the latitude (but not both).
2. The customized policy compiled by the user is then discharged to a static analysis that tracks flows of fine-grained privacy units in search of violations. The analysis addresses several interesting challenges, including (i) sound and efficient enforcement of useful correlations between different units (such as longitude and latitude) and (ii) modeling of semantic transformations (for example, the conversion of a location into its corresponding address).

Thanks to the ability to express and enforce fine-grained privacy restrictions, MorphDroid is able to verify realistic policies with high accuracy. The declarative specification interface permits different users — including security specialists, administrators and information officers — to statically enforce custom policies with minimal effort.

We have implemented MorphDroid. In this paper, we formalize the language and the components of MorphDroid, and present a thorough experimental evaluation of MorphDroid atop DroidBench [2] [2] (the standard benchmark adopted to evaluate precision and recall of static and dynamic analyses of Android apps), and the 500 top-popular Google Play apps in 2014. The experiments on DroidBench shows that MorphDroid achieves a precision and recall above 90%, demonstrating its practical effectiveness. On top-popular apps, our experiments involve a spectrum of 5 security policies, ranging from a strict coarse-grained policy to a relaxed fine-grained policy. The gap is dramatic. The most conservative policy detects violations in 171 of the apps (34%), whereas the least conservative policy, which is also the most realistic policy, marks only 4 of the apps as misbehaved ($< 1\%$). This significant delta lends support to the MorphDroid design in practice. In addition, MorphDroid is demonstrated to be efficient with an average analysis time of < 20 seconds per app, where on average, our benchmark apps consist of 1.4M lines of code.

2. TECHNICAL OVERVIEW

In this section, we describe the MorphDroid algorithm, highlighting its unique features by comparison with existing techniques.

2.1 Running Example

In Figure 2.1, we present a small fragment from the code of the Sky Map app (package: `com.google.android.stardroid`).[3] This app, developed by Google, utilizes the user's GPS location as well as other sensors to provide rich information on stars and planets visible to the user. For readability, we simplified the code as well as the signatures of some of the invoked methods.

```
 1  Location loc = lm.getLocation();
 2  double lon = loc.getLongitude();
 3  double lat = loc.getLatitude ();
 4  showLocationToUser(new LatLong(lat, lon));
 5
 6  private void showLocationToUser(LatLong lal) {
 7  double lon = lal.lon;
 8    double lat = lal.lat;
 9    Address addr=new Geocoder().getAddress(lat, lon);
10    String locality = addr.getLocality ();
11    String msg = "Location set to " + locality ;
12    Log.d(TAG, msg);
13     ... }
```

Figure 2.1: Fragment from the Code of the Sky Map App

As part of its behavior, Sky Map obtains the user's longitude and latitude, which it uses to construct a `LatLong` instance. That object is transformed, via the `Geocoder` class, into an `Address` object `addr` whose `locality` field is persisted in clear form into the debug log (the `Log.d` call).

Assume that the privacy policy to be verified allows the release of only partial information about the location (like to locality) to the log. The challenge is to automatically check whether or not the policy is violated. To reach the correct conclusion, whereby the policy is satisfied, the analysis should account for the data transformations executed by the code, as visualized in Figure 2.2. In the figure, we convey not only the data transformation trace, but also the amount (or number of units) of private information flowing into, and out of, every transformation.

First the longitude and latitude fields are projected from the location. These fields are then combined in a `LatLong` object, which carries the *semantic* equivalent of the user's position (but not the complete location, as the altitude is missing). The position is transformed into an address, which is a *semantically* equivalent representation. Finally, the locality field is extracted and released.

2.2 Merit of Taint Analysis

A popular method to check integrity and confidentiality properties is via taint tracking [8, 18, 19]. Intuitively, taint analysis tracks information flows extending between designated *sources* (in the example, line 1 that reads the location) and *sinks* (in the example, line 12 that releases data to external observers) by propagating taint tags across statements.

As an example, assume that `getLocation` generates a `LOC` tag representing the complete location. This tag is initially attached to (the object pointed-to by) `loc`. It then reaches the `lat` and `lon` fields of the `LatLong` instance that is passed into `showLocationToUser`, and so on. Tracking the `LOC` tag alone yields the solution in Figure 2.3 (without the part in square brackets). This solution is overly conservative, suggesting that the entire location (and not merely the address locality field) is potentially persisted to the log.

Importantly, this problem is not fixed by exploding `LOCATION` into multiple finer-grained tags, e.g. `LOC.LAT` and `LOC.LON`. The above problem repeats itself in this case too, as illustrated in Figure 2.3 with the part in square brackets included. Though the longitude and latitude flow temporarily along distinct paths, and the finer-grained tags are able to disambiguate the flows, precision is lost once both tags flow into the `LatLong` constructor and subsequently transformed through `getAddress`.

A more expensive solution is to conduct the taint analysis in two phases. In the first phase, tainted `Address` objects are detected. In particular, the `getAddress` call at line 9 is treated as a sink. Then a second analysis is conducted, where the sources are the statements defining tainted `Address` objects (again, line 9), and the sinks are the original sinks (`Log.d` at line 12 in our example). This is illustrated in Figure 2.4 (without the part in square brackets). This too is insufficient, as `ADDR` is too coarse, suggesting that the

[2] https://github.com/secure-software-engineering/DroidBench

[3] https://play.google.com/store/apps/details?id=com.google.android.stardroid

entire address is potentially released through `Log.d`.

This leads to the fourth and final alternative, whereby (i) two rounds of analysis are conducted, as in the third alternative; and additionally, (ii) all taint tags are fine grained. We illustrate this alternative in Figure 2.4 with the part in square brackets included. Note that we have only annotated the figure with relevant tags. Other fields that would be tracked in practice include e.g. the postal code, phone, sub-administrative area, etc.

In addition, it should be observed that traditional taint analyses track and report data flows from sources to sinks not intercepted by sanitizers. Wherever a sanitizer intercepts flow of tainted data, those analyses abort data-flow tracking under the assumption that the threat has been addressed.

When the purpose of the static analysis is to enforce privacy (as opposed to integrity), a traditional taint-analysis approach would be insufficient to model the concept of declassification, which is predominant in privacy policies. For example, a declassification rule within a privacy policy may establish that revealing *either* the latitude *or* the longitude field of a `Location` object (but *not both*) would not constitute a privacy violation. This rule could not be properly modeled by simply aborting flow tracking at methods `getLatitude` and `getLongitude`, as that would prevent the analysis e.g. from rejecting a program that recombines the latitude and longitude to reconstruct the full `Location`. Rather, MorphDroid tracks what types of transformations take place.

To summarize, verification of fine-grained policies via taint analysis explodes the complexity of the analysis. It demands tracking of multiple fine-grained tags, and in addition, the analysis is divided into multiple stages that connect between intermediate sources and sinks without proper support for declassification. This is not a satisfactory solution, motivating *semantic flow tracking* as a direct means to recover the flow graph in Figure 2.2.

2.3 The MorphDroid Approach

The discussion above motivates an alternative design to taint tracking for the purpose of privacy analysis. In specific, there is the need to reason about release of private information in terms of fine-grained semantic units, as established by Tripp and Rubin [20]. Contrary to integrity threats, which reduce effectively to *qualitative* source/sink reachability queries, the *quantitative* and *structural* aspects — accounting for the number of atomic privacy units that the app releases as well as their relationships — play a major role in privacy analysis. Informed by this motivation, we have made two principal design choices. We discuss each in turn.

First, our abstraction of private data is not as a set of disparate taint tags. Instead, we track a rich semantic abstraction, in the form of a data structure consisting of multiple atomic privacy units, such that the relationships and correlations between the units are modeled explicitly. As an example, the location consists of fields like longitude, latitude and altitude, where the longitude and latitude together form the user's position. The longitude and latitude are themselves compound data structures, which consist of integral and fractional fields. This decomposition is visualized in Figure 2.5. Another example is the device ID, or IMEI number, which consists of 15 position/digit units, where the first 8 digits identify the carrier, and the last 7 digits the specific device. Therefore, we can account for partial release of the device ID, which might not be as severe as releasing the entire ID.

Second, building on the rich abstraction, the MorphDroid analysis algorithm further accounts for semantic transformations. To illustrate this, consider the longitude and latitude fields. If either of these fields is released having first been cast into an integer, as in `Log.d(Integer.parseInt(longitude))`, then the fractional unit has not been released. On the other hand, the same operation, when applied to the device ID, is the identity transformation. Drawing these semantic distinctions goes beyond statement-level taint propagation, where the data-flow equations are general and derive from the semantics of the programming language. Structural modeling of private fields lets us model concrete operations, like `parseInt`, differently in different contexts.

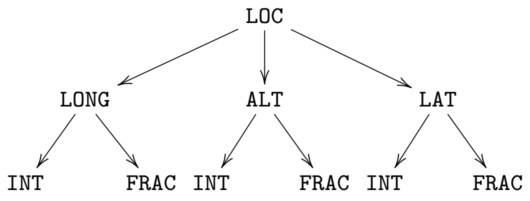

Figure 2.5: Semantic representation of the location field

MorphDroid tracks and transforms private fields interprocedurally atop a call-graph representation of the target program. Since private fields potentially flow into and out of the heap, MorphDroid utilizes a variant of Andersen's call-graph construction algorithm [1], such that pointer analysis is interleaved with resolution of virtual calls. Consequently, the call graph also conveys aliasing relationships over variables and (object/static) fields, thereby enabling flow tracking through the heap.

For interprocedural data-flow analysis, MorphDroid utilizes the IFDS (aka RHS) algorithm [13]. This algorithm guarantees polynomial time and space complexity under the assumption that the transformers are distributive. As we explain later, in Section 4.3, this assumption holds for the MorphDroid transformers. Intuitively, correlations between atomic fields are expressible thanks to the richness of our abstraction, and so distributivity is hardly a restriction.

3. SPECIFICATION LANGUAGE

In this section, we present the MorphDroid specification language, used to express fine-grained constraints on release of private data, and explain its semantics. We also describe semantic checks to catch likely specification errors.

Intuitively, a policy specifies what information coming from specific sources can be released to what specific sinks. The idea is that an approach like standard taint analysis (that is, no information coming from a source can flow to a sink) is too restrictive for mobile applications (e.g., an app might need to disclose the location in order to show where the user is inside a map). Therefore, the specification language allows a user to express what information can be released to whom.

3.1 Language Grammar and Semantics

The grammar of the MorphDroid specification language is given in Figure 3.1. A production of the grammar is illustrated in Figure 3.2. We have also used the second component of this policy in our experiments in Section 5.

The grammar is rooted at variable s, which unfolds into a specification, or policy, consisting of three sections. The sections are each enclosed in curly braces. In the grammar, we denote the sets of all privacy sinks and tags by Sinks and Tags, respectively.

The first section (part (1) in the policy in Figure 3.2), rooted at variable st, combines sink (or release) APIs into groups. This is not merely syntactic sugar, as the meaning of enforcing a privacy restriction on a group is that it must hold collectively over all the grouped sinks. For example, given the `INTERNET` group in the illustrative policy in Figure 3.2, if the restriction is that either the IMEI

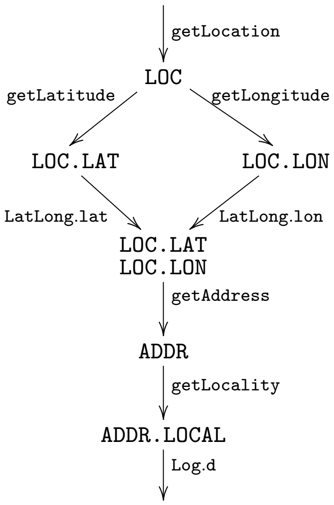

Figure 2.2: Our objective

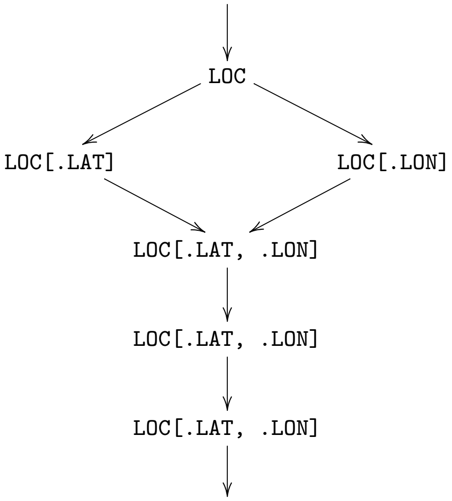

Figure 2.3: Standard taint analysis, and with multiple labels

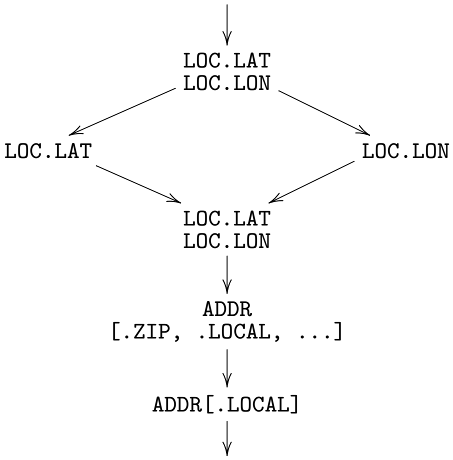

Figure 2.4: Taking into account transformations, and with multiple labels for an address

$$
\begin{array}{rcll}
s & ::= & \{st\}\{tt\}\{[sp]\} & \textbf{(spec)} \\
id & ::= & [a-zA-Z0-9]^{\star} & \textbf{(identifier)} \\
st & ::= & st\ ;\ st \mid id\ :=\ si & \textbf{(m-lists)} \\
si & ::= & si, si \mid_{\{\mathsf{m}:\ \mathsf{Sinks} \cup id\}} \mathsf{m} & \textbf{(m-list)} \\
tt & ::= & tt\ ;\ tt \mid id\ :=\ ta & \textbf{(u-conds)} \\
ta & ::= & ta, ta \mid c_c & \textbf{(u-cond)} \\
sp & ::= & sp\ ;\ sp \mid id\ \models\ c & \textbf{(prop)} \\
c & ::= & c_c \mid c \otimes c & \textbf{(o-cond)} \\
c_c & ::= & (c_c \oplus c_c) \mid_{\{\mathsf{t}:\ \mathsf{Tags} \cup id\}} \mathsf{t} & \textbf{(i-cond)}
\end{array}
$$

Figure 3.1: Grammar of the MorphDroid specification language

$$
(1)\quad \left\{ \begin{array}{l} \texttt{INTERNET} := \texttt{URL.openConnection()}, \\ \qquad\qquad\quad \texttt{WebView.loadUrl(String)} \end{array} \right\}
$$

$$
(2)\quad \left\{ \begin{array}{l} \texttt{ID} := \texttt{IMEI} \oplus \texttt{IMSI} \oplus \texttt{PHONE\_NUMBER}; \\ \texttt{PARTIAL\_LOC} := \\ \qquad \texttt{LOC.LON} \oplus \texttt{ADDR.COUNTRY} \oplus \texttt{ADDR.LOCAL}, \\ \qquad \texttt{LOC.LAT} \oplus \texttt{ADDR.COUNTRY} \oplus \texttt{ADDR.LOCAL}; \\ \texttt{PARTIAL\_ID} := \texttt{IMSI.CARR} \oplus \texttt{IMSI.COUNTRY} \oplus \\ \qquad \texttt{PHONE\_NUMBER.PREFIX} \oplus \texttt{IMEI.MODEL} \end{array} \right\}
$$

$$
(3)\quad \left\{ \begin{array}{l} \texttt{INTERNET} \models (\texttt{PARTIAL\_ID} \oplus \texttt{ADDR}) \otimes \\ \qquad\qquad\qquad (\texttt{ID} \oplus \texttt{PARTIAL\_LOC}) \end{array} \right\}
$$

Figure 3.2: A policy specification

or the location are released to INTERNET (but not both), then a violation would occur if the IMEI is released via openConnection while the location is released via loadUrl.

Note that overlap between sink groups is disallowed, or more accurately, two sink groups may only overlap each other if at most one of them is referenced at the third section (which places constraints on groups). This design choice is for two reasons: to simplify policy verification as well as to prevent user error due to subtle ambiguities. This restriction is enforced via syntactic checking of the first section (which occurs at load time).

The middle section (part (2)), rooted at *tt*, plays an analogous role for privacy tags. The user can express custom tags as a set of tag sequences combined via the $\oplus$ operator. In the illustrative policy, for instance, PARTIAL_LOC might be either the combination of the longitude, country and locality or the combination of the latitude and the latter two fields.

Finally, the third and last section (part (3)) defines constraints on sink groups (as declared in the first section) using tags in Tags or tag aggregations (as declared in the second section). First of all, a combination of tags with $\oplus$ represents that it is allowed to release *at most* all the tags contained in the concatenation. For instance, in part (3) of Figure 3.2, PARTIAL_ID $\oplus$ ADDRESS represents that an app is allowed to release the partial identifier and the full address.

A constraint on a sink group can contain many of these combinations aggregated via the $\oplus$ operator. $c_1 \oplus c_2$ denotes that an app is allowed to release what is specified either in c_1 *or* in c_2. Therefore, the policy should be interpreted to mean that each formula represents the *highest degree of authorization* granted to the application. Releasing more than what is specified is a violation. For instance, in part (3) of Figure 3.2, an app is allowed to release (i) the full identifier and a partial location or (ii) a partial identifier and the full address, but not both the full identifier and the full address. Note that if this component is not present, then we default to the standard (taint) behavior of forbidding any leakage of information.

3.2 Semantic Checks

The policy, as compiled by the user, expresses restrictions on release of private information that are particular to the user at hand. Grouping sink APIs and combining private fields into compound conditions are ways of modularizing the specification while also making it more concise and thus also simpler to reason about.

Still, inconsistencies might occur. As an example, the user may authorize release of the full location, but not the address, to INTERNET. To avoid inconsistencies or ambiguities in the specification, MorphDroid runs built-in semantic checks on the policy.

The checks performed by MorphDroid are in the form of inequalities between private fields, such as LOC $\geq$ ADDR. The meaning of this expression is that authorization to release the full location logically entails authorization to release the full address. If the specification — as authored by the user — contradicts this judgment, then a warning is presented to the user. Another example is the combination of IMEI $\geq$ IMSI with IMSI $\geq$ IMEI, which assert the "equivalence" between the IMEI and IMSI identifiers. If the specification refers to only one of these fields, but not the other, then a warning is flagged.

The user is free to dismiss the warnings issued by MorphDroid. As such, they are indeed warnings and not errors. Their only pur-

```
st ::=
    x.f = y | x = y.f                                   (heap)
    | x = getLocation() | x = getDeviceId() | ...   (sources)
    | Log.d(x) | loadURL(x) | ...                      (sinks)
    | x = y.getLocal() | x = y.getLatitude() |      (transf.)
      | x = getAddress(y,z) | ...
    | x = y.toString() | x = y + z |                 (strings)
      | x = y.substring(i1, i2) | ...
```

Figure 4.1: A Java-like language

pose is to address cases of oversight.

4. FORMALIZATION

In this section, we present a Java-like language exposing all the relevant features supported by MORPHDROID, based on which we define the abstract domain and semantics of our static analysis.

4.1 Language

Figure 4.1 defines a Java-like language that will be used to define the abstract semantics of MORPHDROID. We restrict ourselves to a core language with representative statements to focus on the essential details of our analysis, though we emphasize that the implementation supports the full Java language and Android APIs.

Our language supports heap accesses (both reads and writes), various privacy sources (like `getLocation` and `getDeviceId`) and sinks (like `Log.d` and `loadURL`). It also includes transformations (e.g. projections like `getLocal` and conversions like `getAddress`) as well as string operations (like `substring` and concatenation).

We assume that the program type checks. This means in specific that (i) variables and heap locations are consistent with their assigned value, (ii) the receiver of `getLocal` is an `Address` object, (iii) the parameters of `getAddress` are double values, (iv) the receiver of `substring` is a string and the parameters are integers values, and (v) string concatenation operates on strings.

Table 1 organizes the statements of the programming language in Figure 4.1, except for sources and sinks, into categories by the type of data transformation they potentially apply. In Section 4.3, we formalize their abstract semantics. For each statement, Table 1 refers to the equation in Section 4.3 that defines its abstract semantics. Conceptually, there are three types of transformations: projection, aggregation and morphing. We have three different types of projection: (i) high-level methods of APIs (like `getLatitude`), (ii) substring operators, and (iii) field accesses (since we interpret an object as a set of tags, and a field access refers only to a subset of these). Similarly, tag aggregation is accomplished through (i) high-level methods (like `getAddress(x,y)`, where `x` contains the latitude and `y` the longitude of a location); (ii) string concatenation; or (iii) heap assignments. Finally, data may be morphed using the `toString` operator (transforming the secret into a string) or with APIs like `getAddress` (transforming a location into an address).

4.2 Abstract Domain

We assume that we can distinguish string values and variables from other types of information (integers, addresses, etc). Therefore, we represent by StrId and Id the set of string and other identifiers, respectively. These identifiers comprise variables as well as abstract heap locations.

Our analysis is aimed at tracking, for each identifier, the confidential information it may carry in the form of a set of privacy tags. In addition, for string identifiers we maintain a flag to track if the tag is stored at the beginning of the string. This is important to define a precise semantics of the `substring` operation, which is often used to extract partial information out of a whole privacy tag. Our abstract domain is composed of a set of facts. Each fact relates an identifier to a secret the identifier *might* contain. Formally, $\Sigma : (\mathsf{StrId} \times \{\mathsf{true}, \mathsf{false}\} \times \mathsf{Tags}) \cup (\mathsf{Id} \times \mathsf{Tags})$.

For instance, $(\mathtt{x}, \mathtt{ADDR})$ represents that variable `x` contains secret `ADDR`, while $(\mathtt{y}, \mathsf{true}, \mathtt{IMEI})$ represents that variable `y` is a string that contains the secret `IMEI` at its beginning.

The abstract domain consists of a set of facts, and the lattice structure is based on set operators. Formally, our abstract domain is $\langle \wp(\Sigma), \subseteq, \cup, \cap \rangle$.

4.3 Abstract Semantics

As is standard with the IFDS algorithm [13], the analysis starts from a set of seeds. Therefore, we define a function *createSeeds* that, given a statement, returns the set of facts that might have been created by source statements. For the language in Figure 4.1, this is formalized as follows:

$$\begin{aligned} \mathit{createSeeds}(\mathtt{st}) = & \{(\mathtt{x}, \mathtt{LOC}) \text{ if } \mathtt{st} = \mathtt{x} = \mathtt{getLocation}()\} \cup \\ & \{(\mathtt{x}, \mathtt{IMEI}) \text{ if } \mathtt{st} = \mathtt{x} = \mathtt{getDeviceId}()\} \end{aligned}$$

In the running example in Figure 2.1, this creates the tag `LOC` at line 1 and assigns it to `loc`, thereby creating the fact $(\mathtt{loc}, \mathtt{LOC})$.

We now define the semantics $\mathbb{S}'$ that, given a statement and a fact, returns all the *new facts* that may have been generated by a statement ($\mathbb{S}' : \Sigma \to \wp(\Sigma)$). This semantics is defined on all the statements of the language in Figure 4.1 excepts sources, since these create the seeds of the analysis and do not depend on input facts.

Transformations have to take into account (i) if the input fact contains the information about the parameters or the receiver of the transformation method, and (ii) whether the secret type is conformant with the expected type. If both of the conditions are satisfied, then the input secret is propagated to the assigned variable transforming the tag in accordance with the semantics of the statement. Note that while `getLatitude` may generate a fact in $\mathsf{Id} \times \mathsf{Tags}$, `getLocal` returns a string and therefore generates a fact in $\mathsf{StrId} \times \{\mathsf{true}, \mathsf{false}\} \times \mathsf{Tags}$, where the secret appears at the beginning of the string (and therefore the second component is equal to `true`). This is formalized as follows:

$$\begin{aligned} & \mathbb{S}'[\![\mathtt{x} = \mathtt{y.getLocal}(), (\mathtt{z}, \mathtt{v})]\!] = \\ & \quad = \{(\mathtt{x}, \mathsf{true}, \mathtt{ADDR.LOCAL}) : \mathtt{z} = \mathtt{y} \wedge \mathtt{v} = \mathtt{ADDR}\} \\ & \mathbb{S}'[\![\mathtt{x} = \mathtt{y.getLatitude}(), (\mathtt{z}, \mathtt{v})]\!] = \\ & \quad = \{(\mathtt{x}, \mathtt{LOC.LAT}) : \mathtt{z} = \mathtt{y} \wedge \mathtt{v} = \mathtt{LOC}\} \end{aligned} \tag{4.1}$$

At lines 2 and 3 of the running example in Figure 2.1, the abstract semantics generates $(\mathtt{lat}, \mathtt{LOC.LAT})$ and $(\mathtt{lon}, \mathtt{LOC.LON})$.

In order to soundly overapproximate the heap, our abstract semantics is parameterized on a standard pointer analysis [1] that models heap accesses conservatively. In particular, given heap access `x.f`, the pointer analysis returns an identifier in $\mathsf{StrId} \cup \mathsf{Id}$. Formally, the pointer analysis defines function *heap* : $\mathtt{x.f} \to (\mathsf{Id} \cup \mathsf{StrId})$ that provides this information.

The semantics of heap reads and writes simply propagates the input secret to the assigned variable or heap location if it refers to the read variable or heap location. Formally,

$$\mathbb{S}'[\![\mathtt{x} = \mathtt{y.f}, (\mathtt{z}, \mathtt{v})]\!] = \{(\mathtt{x}, \mathtt{v}) : \ \mathtt{z} = \mathit{heap}(\mathtt{y.f})\} \tag{4.2}$$

$$\mathbb{S}'[\![\mathtt{x.f} = \mathtt{y}, (\mathtt{z}, \mathtt{v})]\!] = \{(\mathit{heap}(\mathtt{x.f}), \mathtt{v}) : \mathtt{z} = \mathtt{y}\} \tag{4.3}$$

In the running example in 2.1, the constructor of `LatLong` at line 4 assigns tags `LOC.LON` and `LOC.LAT` to fields `lon` and `lat`, respectively. Assume that the pointer analysis produces identifiers #1 and #2 for the location pointed-to by fields `lon` and `loc`, respectively. Then the abstract semantics of heap assignment produces the facts $(\#1, \mathtt{LOC.LON})$ and $(\#2, \mathtt{LOC.LAT})$.

These tags are then retrieved from the heap and assigned to local variables at lines 7 and 8. Since the pointer analysis is sound, it will

	projection	**aggregation**	**morphing**
fields	`x = y.f` (Eq. 4.2)	`x.f = y` (Eq. 4.3)	
strings	`x = y.substring(`i_1`,`i_2`)` (Eq. 4.7)	`x = y + z` (Eq. 4.6)	`x = y.toString()` (Eq. 4.5)
methods	`x = y.getLatitude(),x = y.getLocal()` (Eq. 4.1)	`x = getAddress(y,z)` (Eq. 4.4)	

Table 1: Organization of the Language Statements into Categories According to the Data Transformation They Potentially Apply

retrieve #1 and #2 at line 7 and 8, respectively. This generates the facts $(\mathtt{lon},\mathtt{LOC.LON})$ and $(\mathtt{lat},\mathtt{LOC.LAT})$ after line 8.

The semantics of the transformation `getAddress` propagates the longitude and latitude received as parameters as specific instances of an `ADDR` tag. We augment Tags with $\mathtt{ADDR}_{\mathtt{LAT}}$ and $\mathtt{ADDR}_{\mathtt{LON}}$. This is needed in order to enforce the distributivity of our analysis, since in this way we can propagate the secrets in isolation, and combine them only at the end of the analysis when checking what is leaked to sinks. Formally,

$$\begin{aligned} &\mathbb{S}'[\![\mathtt{x} = \mathtt{getAddress}(\mathtt{y},\mathtt{z}),(\mathtt{w},\mathtt{v})]\!] = \\ &\quad = \{(\mathtt{x},\mathtt{ADDR}_{\mathtt{LAT}}) : \mathtt{w} = \mathtt{y} \wedge \mathtt{v} = \mathtt{LOC.LAT}\} \cup \\ &\quad\quad \{(\mathtt{x},\mathtt{ADDR}_{\mathtt{LON}}) : \mathtt{w} = \mathtt{z} \wedge \mathtt{v} = \mathtt{LOC.LON}\} \end{aligned} \tag{4.4}$$

When applied to the running example in Figure 2.1, given the input facts $(\mathtt{lon},\mathtt{LOC.LON})$ and $(\mathtt{lat},\mathtt{LOC.LAT})$, the abstract semantics creates, at line 9, the output facts $(\mathtt{addr},\mathtt{ADDR}_{\mathtt{LON}})$ and $(\mathtt{addr},\mathtt{ADDR}_{\mathtt{LAT}})$, respectively. Then the `getLocality` transformation at line 10 generates the facts $(\mathtt{locality},\mathsf{true},\mathtt{ADDR}_{\mathtt{LON}}.\mathtt{LOCAL})$ and $(\mathtt{locality},\mathsf{true},\mathtt{ADDR}_{\mathtt{LAT}}.\mathtt{LOCAL})$.

We now define the semantics $\mathbb{S}'$ of string operations. The receiver of `toString` may be an identifier in either StrId or Id. In the first case, if the input fact refers to the receiver, we simply propagate the input secret to the assigned variable. In the second case, we create a secret that represents a string containing at the beginning the sensitive input. Formally,

$$\begin{aligned} &\mathbb{S}'[\![\mathtt{x} = \mathtt{y}.\mathtt{toString}(),(\mathtt{z},\mathsf{b},\mathtt{v})]\!] = \{(\mathtt{x},\mathsf{b},\mathtt{v}) : \mathtt{z} = \mathtt{y}\} \\ &\mathbb{S}'[\![\mathtt{x} = \mathtt{y}.\mathtt{toString}(),(\mathtt{z},\mathtt{v})]\!] = \{(\mathtt{x},\mathsf{true},\mathtt{v}) : \mathtt{z} = \mathtt{y}\} \end{aligned} \tag{4.5}$$

String concatenation propagates the input secrets to the assigned variable. If the secret is contained in the left operand, then we propagate the boolean flag as well. Instead, if it is in the right operand, then we set the boolean flag to false to conservatively represent that the portion of the string containing the secret may not be at the beginning. Formally,

$$\begin{aligned} &\mathbb{S}'[\![\mathtt{x} = \mathtt{y} + \mathtt{z},(\mathtt{w},\mathsf{b},\mathtt{v})]\!] = \\ &\quad = \{(\mathtt{x},\mathsf{b},\mathtt{v}) : \mathtt{w} = \mathtt{y}\} \cup \{(\mathtt{x},\mathsf{false},\mathtt{v}) : \mathtt{w} = \mathtt{z}\} \end{aligned} \tag{4.6}$$

At line 11 of our running example in Figure 2.1, we infer that $(\mathtt{msg},\mathsf{false},\mathtt{ADDR}_{\mathtt{LON}}.\mathtt{LOCAL})$ and $(\mathtt{msg},\mathsf{false},\mathtt{ADDR}_{\mathtt{LAT}}.\mathtt{LOCAL})$.

The `substring` operator has to be tailored to the secret type flowing into it. It further needs precise information on the indexes used to cut the string to know what portion of the secret is propagated. Therefore, we parameterize our analysis on a standard numerical constant propagation analysis, represented by a function *getConst* that returns either the constant value of a numerical variable or $\top$. $\geq_\top$ and $\leq_\top$ represent the extension of numerical comparisons $\geq$ and $\leq$ to support $\top$, respectively. In particular, these comparisons returns true if and only if the relation *definitely* holds. For instance $i_1 \leq_\top i_2$ iff both i_1 and i_2 are not $\top$ and $i_1 \leq i_2$.

We then define a function *cut* that, given a secret and two variables used as indices in `substring`, returns the portion of the secret that is projected. This function takes into account the specification of the secret structure (e.g., the first eight characters of the IMEI represent the carrier). *cut* is defined as follows:

$$\begin{aligned} &\mathit{cut}((\mathsf{b},\mathsf{d}),i_1,i_2) = \{(\mathsf{false},\mathsf{d}) : \mathsf{b} = \mathsf{false}\} \cup \\ &\left\{ \begin{array}{l} (\mathsf{false},\mathsf{d}') : \mathsf{b} = \mathsf{true} \wedge \mathsf{d} = \mathtt{IMEI} \wedge \\ \quad \mathsf{d}' \in \left\{ \begin{array}{ll} \mathtt{IMEI.CARRIER} & \text{if } v_1 \geq_\top 0 \wedge v_2 \leq_\top 7 \\ \mathtt{IMEI.NUMBER} & \text{if } v_1 \geq_\top 8 \\ \mathtt{IMEI} & \text{otherwise} \end{array} \right\} \end{array} \right\} \cup \\ &\left\{ \begin{array}{l} (\mathsf{false},\mathsf{d}') : \mathsf{b} = \mathsf{true} \wedge \mathsf{d} = \mathtt{IMSI} \wedge \\ \quad \mathsf{d}' \in \left\{ \begin{array}{ll} \mathtt{IMSI.COUNTRY} & \text{if } v_1 \geq_\top 0 \wedge v_2 \leq_\top 2 \\ \mathtt{IMSI.CARR} & \text{if } v_1 \geq_\top 3 \wedge v_2 \leq_\top 4 \\ \mathtt{IMSI.NUMBER} & \text{if } v_1 \geq_\top 5 \\ \mathtt{IMSI} & \text{otherwise} \end{array} \right\} \end{array} \right\} \cup \\ &\quad \ldots \end{aligned}$$

where $v_1 = \mathit{getConst}(i_1)$, $v_2 = \mathit{getConst}(i_2)$. Finally, $\mathbb{S}'$ has to apply *cut* and propagates the result to the assigned variable only if the input fact contains information about the receiver of `substring`. Formally,

$$\begin{aligned} &\mathbb{S}'[\![\mathtt{x} = \mathtt{y}.\mathtt{substring}(\mathtt{i}_1,\mathtt{i}_2),(\mathtt{z},\mathsf{b},\mathtt{v})]\!] = \\ &\quad = \{(\mathtt{x},\mathsf{false},\mathtt{v}') : \mathtt{z} = \mathtt{y} \wedge \mathtt{v}' = \mathit{cut}(\mathsf{s}(\mathtt{y}),i_1,i_2)],\mathsf{b},\mathtt{v})\} \end{aligned} \tag{4.7}$$

Instead, sink statements do not generate any new fact. Therefore, their semantics simply returns an empty set. Formally,

$$\begin{aligned} &\mathbb{S}'[\![\mathtt{Log.d}(\mathtt{x}),(\mathtt{z},\mathtt{v})]\!] = \emptyset \\ &\mathbb{S}'[\![\mathtt{loadURL}(\mathtt{x}),(\mathtt{z},\mathtt{v})]\!] = \emptyset \end{aligned}$$

We are now in position to extend the semantics $\mathbb{S}'$ to a set of facts in $\wp(\Sigma)$. However, input facts are propagated if and only if (i) the identifier in the secret is not assigned by the statements, or (ii) it is assigned but it represents an abstract heap location (since we adopt weak updates to preserve the soundness of the analysis). This is formalized by the following *project* function.

$$\begin{aligned} &\mathit{project}(\mathtt{st},\mathsf{V}) = \{(\mathtt{y},\mathtt{v}) \in \mathsf{V} \text{ iff} \\ &\quad \left\{ \begin{array}{l} \mathtt{st} \in \{\mathtt{Log.d}(\mathtt{x}),\mathtt{loadURL}(\mathtt{x})\} \\ \mathtt{st} = \mathtt{x} = \ldots \wedge \mathtt{x} \neq \mathtt{y} \\ \mathtt{st} = \mathtt{x.f} = \ldots \end{array} \right. \quad \} \end{aligned}$$

Finally, we define $\mathbb{S} : \wp(\Sigma) \rightarrow \wp(\Sigma)$ as the pointwise application of $\mathbb{S}'$ to all the partitions contained in an abstract state, the creation of secrets from sources, and the propagation of input secrets that are not overwritten by the analyzed statement. Formally,

$$\mathbb{S}[\![\mathtt{st},\mathsf{V}]\!] = \bigcup_{\mathsf{v} \in \mathsf{V}} \mathbb{S}'[\![\mathtt{st},\mathtt{v}]\!] \cup \mathit{project}(\mathtt{st},\mathsf{V}) \cup \mathit{createSeeds}(\mathtt{st})$$

4.4 Policy Verification

At the end of the analysis, MORPHDROID computes a set of input and output facts for each statement. Therefore, each sink statement is related to a set of input facts. First of all, we collect for each sink the tags that may have been leaked. Then, we group together all the sinks following the specification of sink groups. Since in MORPHDROID enforces that groups of sinks in a policy cannot overlap as explained in Section 3.1, each sink belongs to exactly one of these groups. At the end of this phase, we produce a function in $\mathsf{SinkIds} \rightarrow \wp(\mathsf{Tags})$ that, for each sink-group identifier, returns the set of privacy tags that might have been leaked through the given sinks.

For instance, imagine that we want to check if the policy constraint $\texttt{LOG} \models \texttt{PARTIAL\_LOC}$ (following the policy specification in Figure 3.2) is validated by our running example in Figure 2.1. In this first phase, we collect the function $\texttt{LOG} \mapsto \{\texttt{ADDR}_{\texttt{LON}}.\texttt{LOCAL}, \texttt{ADDR}_{\texttt{LAT}}.\texttt{LOCAL}\}$ representing that the locality of the current address might have been leaked through logging.

We are now in position to define a predicate that, given a policy specification and the analysis results of a program, holds if and only if the program respects the privacy policy. This is formalized by the following *satisfyPolicy* predicate, where c represents a privacy policy, and r the results of the analysis represented by a function in $\mathsf{SinkIds} \to \wp(\mathsf{Tags})$.

$$\mathit{satisfyPolicy}(\mathsf{c},\mathsf{r})$$
$$\Updownarrow$$
$$\begin{array}{ll} (1) & \forall \mathsf{s} \in \mathit{dom}(\mathsf{r}) : \mathsf{r}(\mathsf{s}) \neq \emptyset \Rightarrow \mathsf{s} \in \mathit{dom}(o), o(\mathsf{s}) = \bigotimes_{i\in[0..n]} c_i \\ (2) & \quad \exists j \in [0..n] : c_j = \bigoplus_{l\in[0..n']} \mathsf{t}_l, \\ (3) & \quad \forall \mathsf{t} \in \mathsf{r}(\mathsf{s}) : \exists j \in [0..n'] : \mathsf{t}_j \Rightarrow \mathsf{t} \end{array}$$

If a sink is not part of our specification, this means that nothing is allowed to be leaked through it. Therefore, if MORPHDROID inferred that a tag was released through a sink, but this is not part of the policy specification (point (1) of the formal definition), the policy is not satisfied. Otherwise, we need to check what is released through each sink aggregation, and if this is covered by the policy specification. This means that for each sink aggregation we check that *all* the leaked privacy tags (point (3)) are covered by at least *one* combination of tags in the formula (point (2)) related to that particular aggregation. Given two tags $\mathsf{t}_1, \mathsf{t}_2 \in \mathsf{Tags}$ we denote by $\mathsf{t}_1 \Rightarrow \mathsf{t}_2$ that t_1 *covers* t_2, that is (i) $\mathsf{t}_1 = \mathsf{t}_2$, (ii) t_1 is the identifier of an aggregation whose t_2 is part, or (iii) $\mathsf{t}_1 \geq \mathsf{t}_2$ according to the inequalities described in Section 3.2.

For our running example and the policy $\texttt{LOG} \models \texttt{PARTIAL\_LOC}$, we consider the combination $c_j = \texttt{PARTIAL\_LOC}$ at point (2). Given this formula, for all the leaked tags (namely, $\texttt{ADDR}_{\texttt{LON}}.\texttt{LOCAL}$ and $\texttt{ADDR}_{\texttt{LAT}}.\texttt{LOCAL}$), the specification PARTIAL_LOC covers them, since ADDR.LOCAL is one of the tags in the combination of the definition of PARTIAL_LOC (as defined in Figure 3.2), and so we have

$$\texttt{PARTIAL\_LOC} \Rightarrow \texttt{ADDR}_{\texttt{LON}}.\texttt{LOCAL} \wedge$$
$$\texttt{PARTIAL\_LOC} \Rightarrow \texttt{ADDR}_{\texttt{LAT}}.\texttt{LOCAL}$$

Therefore, the running example respects this privacy policy.

4.5 Algorithm

```
Data: Privacy policy specification c and program p
Result: true if and only if p respects c
1 begin
2     Seeds ⟵ collect all the seeds of program p
3     foreach sr ∈ Seeds do
4         r ⟵ r ∪ fixpoint solution of 𝕊 considering seed sr
5     end
6     t ⟵ aggregate all the leaked tags
              per the sink groups used in policy c
7     return satisfyPolicy(c, t)
8 end
```

Algorithm 1: MORPHDROID algorithm

We present the complete algorithm of MORPHDROID in Algorithm 1. We start by collecting all the seeds contained in the given program (line 2). Then, for each seed (line 4) we compute the fixpoint of the abstract semantics $\mathbb{S}$ (line 5) defined in Section 4.3. After collecting the results for all the seeds, we aggregate them per the sink groups defined in the given policy (line 7). Finally, we return the result of the policy check (line 8) defined in Section 4.4.

5. IMPLEMENTATION AND EVALUATION

In this section, we describe our implementation of MORPHDROID. We then present experiments that we have conducted to evaluate the efficacy of MORPHDROID and the viability of its underlying usage scenario.

5.1 Implementation Details

MORPHDROID is a client security analysis built on top of the T. J. Watson Library for Analysis (WALA) framework.[4] From an implementation perspective, MORPHDROID involves three phases: (1) Call-graph and points-to-graph construction, (2) Taint analysis, and (3) Detection of policy infringements.

Phase 1 relies on WALA in order to build the call graph and points-to graph modeling the execution of the application and its supporting libraries. In particular, we rely on the 0-CFA [16] call-graph construction algorithm with the standard settings for points-to analysis. In addition, we rely on the entrypoints defined by Scandroid [7] with standard options (that is, taking into account callbacks, overridden functions and constructors of Android components) to model the possible executions of an app, and in particular the control flow between different events.

Phase 2 performs data-flow analysis in order to detect any flow of data from a source to a sink while taking into account how data is morphed by the application. This is instantiated as a standard IFDS [13] problem in WALA. We manually defined synthetic models for Android and Java libraries. Our models concern only data (or information) flow (e.g., StringBuilder.append flows information from the parameter to the receiver), and they are secret independent. We usually defined synthetic models on the library methods called by an app, rather than on internal native methods, since their semantics is well documented, and the definition of data flow is straightforward. Therefore, these synthetic models define also how the information flow through containers, JSON objects, etc.. We have created the models lazily by forcing it to halt if a model is missing. Altogether, this entailed manual creation of synthetic models for about 200 library APIs. For the most part, the semantics of these APIs was well-known and properly documented, so it did not require manual inspection of library code. Therefore, all together the annotation of these 200 APIs required about 2 hours.

At the end of Phase 2, MORPHDROID serializes the results of the analysis. In Phase 3, MORPHDROID only needs to operate on the serialized data, rather than the application's code. Thanks to this optimization, the same app can be analyzed against different policies without having to repeat the more expensive phases (Phases 1 and 2).

5.2 Evaluation Setup

The MORPHDROID implementation supports four main types of sources, and in particular the IMEI (e.g., TelephonyManager.getDeviceId()), the IMSI (e.g., TelephonyManager.getSub−scriberId()), the phone number (e.g., TelephonyManager.get Line1Number()), and the location (e.g., LocationManager.get−LastKnownLocation()). In addition, the tag set represents the address obtained by passing the location to Geocoder.getFrom Location. These tags are aggregated into 3 types, as formalized in part (2) of the policy in Figure 3.2, to represent generic identifiers ID, partial identifiers PARTIAL_ID, and partial locations PARTIAL_LOC.

For sinks, we support 60 method signatures aggregated into six different types: logging (LOG), internet (INTERNET), shared preferences (SHPREF), intent (INTENT), file system (FILE) and database

[4] http://wala.sf.net

pol1	$\emptyset$
pol2	$\texttt{INTERNAL} \models \texttt{ID} \otimes \texttt{ADDRESS}$
pol3	$\texttt{INTERNAL} \models \texttt{ID} \oplus \texttt{ADDRESS}$
pol4	$\texttt{INTERNAL} \models \texttt{ID} \oplus \texttt{ADDRESS}$ $\texttt{EXTERNAL} \models \texttt{ID} \otimes \texttt{ADDRESS}$
pol5	$\texttt{INTERNAL} \models \texttt{ID} \oplus \texttt{ADDRESS}$ $\texttt{EXTERNAL} \models (\texttt{PARTIAL\_ID} \oplus \texttt{ADDRESS}) \otimes$ $(\texttt{ID} \oplus \texttt{PARTIAL\_LOCATION})$

Table 2: Policies Used in Experimental Evaluation

(DB). Our implementation further aggregates sinks into outer categories INTERNAL (representing all the sinks keeping the data on the device, namely log, shared preferences, files and database), and EXTERNAL (representing sinks that release data outside through the Internet or intents). This is defined by the following rules:

$$\texttt{INTERNAL} := \texttt{LOG}, \texttt{SHPREF}, \texttt{FILE}, \texttt{DB}$$
$$\texttt{EXTERNAL} := \texttt{INTERNET}, \texttt{INTENT}$$

To evaluate the effectiveness of MorphDroid, we implemented five different privacy policies using the MorphDroid privacy specification language, as shown in Table 2. The policies are decreasingly restrictive, representing the common requirement to enforce privacy without penalizing the functionality of the application:

- pol1 is the strictest policy, equivalent to a standard taint analysis: if anything flows from any source to any sink, the program will not be validated.
- pol2 allows the application to release, *at most*, *either* the device ID *or* the address, but not both, and only to internal sinks, such as log files and local databases. Neither the ID nor the address can be released to external sinks, such as Internet servers. Since policies represent the highest degree of authorization granted to an application, an application that, for example, releases neither the ID nor the address will pass validation.
- pol3 allows the application to release, *at most*, *both* the ID *and* the address, as long as the release is to internal sinks. For example, an application that releases internally just the ID will pass validation.
- pol4 relaxes pol3 by allowing additionally the external release of, *at most*, *either* the ID *or* the address.
- pol5 allows the release of, *at most*, *either* a partial version of the ID and the full address, *or* a partial version of the location and the full ID, as defined in Figure 3.2.

Note that, given the semantic checks described in Section 3.2, ADDR is equivalent to $\texttt{LOC.LAT} \oplus \texttt{LOC.LON}$, so our policies consider also the case in which the latitude and longitude are released instead of the full address. Given that pol$i$ is more restrictive than pol$(i+1)$, for $i = 1,\ldots,4$, if a program does not satisfy pol$i$, then it does not satisfy pol$j$ for any $j < i$. In this sense, the above represents an increasingly permissive chain of policies, all dealing with combination of internal and external releases of data obtained from the device ID and location. In this evaluation, we chose these two pieces of private information because they are representative of numerous interesting privacy challenges. Indeed, releasing the device ID and location in their entirety would uniquely disclose the geographical position of a user. On the other hand, partial releases of this information—as permitted, for example, by pol5—would anonymize the link between the user's identity and location, which in most cases would be sufficient to protect the user's privacy without impacting the application's functionality.

5.3 Experimental Hypotheses

Before we present the raw experimental data output by our testbed, we lay out the experimental hypotheses guiding our research.

H1: Analysis Precision and Recall Static analysis is approximate to guarantee termination. However, rough approximation might seriously affect the quality of the results, yielding many false alarms. In addition, building a static model for an Android application is a hard challenge, since it requires abstractions for (i) the app lifecycle, (ii) the heap structure, and (iii) the call graph. Abstracting each of these components might introduce unsoundnesses due to the limitations of existing tools. Therefore, our first experimental hypothesis concerns the precision and recall of MorphDroid.

H1 MorphDroid correctly models the important aspects of the Android system, achieving precision and recall scores of above 90% when applied to microbenchmarks that test the analysis w.r.t. the main components of Android apps supported by MorphDroid.

H2: Policy Precision Intuitively, the *precision* of a sound analysis is inversely proportional to the number of false positives it reports. While a sound static analysis will inevitably suffer from false positives due to the over-approximations it adopts during the modeling phase, from a security perspective false positives may also arise from modeling privacy rules in an overly conservative manner. For example, reporting any flow from a source to a sink as a potential vulnerability, as in policy pol1, will likely generate numerous false positives due to the fact that declassification is not taken into any account. We expect the number of such false positives to be significantly reduced when the analysis *properly* models declassification rules: For this to happen, the analysis has to soundly model the various correlations between different units of private data as well as any semantic transformation on that data. This leads to the following analysis-precision hypothesis:

H2 Given a fine-grained policy, enforcing it at a coarse level yields many more false positives ($> 50\%$) than fine-grained tracking does.

H3: Analysis Performance For a static security analysis to be accepted by security officers, precision is not the only concern that needs to be addressed. Users of a security-analysis tool, especially for mobile applications —which have a fast turnaround with new versions made available sometimes on a weekly basis—expect quick responses in order to detect and correct vulnerabilities as early as possible in the application-development lifecycle. Security officers also expect a reliable analysis tool—one that completes the analysis of real-world commercial applications without crashing. We optimized and perfected the design and implementation of MorphDroid to satisfy these important usability requirements, which lead to the formulation of the following analysis-performance hypothesis:

H3 The MorphDroid algorithm is efficient, with a running time between 1 and 5 minutes on average, at most 10 minutes in the worst case, and scales to real-world mobile applications.

5.4 Evaluation of the Results

In the experiments, MorphDroid, which is a Java program, was developed and executed on top of a Java Development Kit (JDK) V1.7.0_55, running on an Apple MacBook Pro with a 2 GHz Intel Core i7 processor, a 16 GB 1600 MHz DDR3 RAM, and operating system MacOS X Yosemite 10.10.2.

The experiments we conducted involved the DroidBench test suite, and the top 500 applications available on Google Play (the most popular Android marketplace) in the United States in June

2014. The version of the Android libraries included in the analysis scope was that of Android 4.4 KitKat with wearable extensions (API level 20).

5.4.1 Evaluation of H1

We evaluated the precision and recall of MorphDroid on DroidBench, an independent open-source benchmark suite for static and dynamic Android analyses that is considered highly comprehensive and challenging. We utilized policy `pol1` to check if MorphDroid correctly detects the data flows of interest.

DroidBench is aimed at checking a wide range of problematic scenarios, some of which fall outside our assumed threat model. As such, we focused on the relevant DroidBench test suites, and in particular those that contain (i) explicit flows, (ii) the privacy sources and sinks in the MorphDroid specification and entrypoints that are detectable by Scandroid, (iii) the full intra-app Android lifecycle, and (iv) standard object-oriented features and data structures. This filtering process still left us with the majority of the DroidBench library, 70 benchmarks to be exact.

MorphDroid obtained the following results:

cases: 70 precision: 91% recall: 96% F-score: 93%

This validates Hypothesis 1, since both precision are recall are above 90%. MorphDroid produced 7 false alarms. 4 of the alarms were caused by our heap abstraction (and in particular, our handling of inductive data structures), while the other 3 were due to the call-graph construction algorithm, which is imprecise w.r.t. dynamic dispatch. False positives are unavoidable in static program analysis, and so the ratio we obtained of 7 false alarms relative to 70 test cases is encouraging, rendering MorphDroid's precision comparable with state-of-the-art static analyzers.

MorphDroid also has 3 misses (or false negatives). These are all due to limitations of the WALA framework in supporting Android. In particular, 2 of the misses are due to built-in sources of unsoundness in heap analysis, and the remaining false negative is because of unsoundness in the type hierarchy. We emphasize that none of these misses is directly related to the MorphDroid algorithm. The misses are all due to the design and/or implementation of the WALA framework. Still, we intend to investigate these problems further to push our recall score above 96%.

5.4.2 Evaluation of H2

Out of 500 applications, MorphDroid infers that (i) 171 apps violates `pol1`, (ii) 97 `pol2`, (iii) 79 `pol3`, (iv) 23 `pol4`, and (v) only 4 `pol5`. Therefore, there is indeed an overall decrease of 2 orders of magnitude in the number of the reports from `pol1` to `pol5`. This flexibility greatly simplifies the work of the analyst because it reduces the time the analyst has to spend on evaluating the results of the analysis. The precision of the analysis would be severely affected if the analysis only took into account flows from sources to sinks—similar to a traditional taint analysis, which corresponds to `pol1`.

Specifically, as conjectured by Hypothesis 2, given a fine-grained policy, such as `pol5`, enforcing it at a coarse level yields *many more* false positives than fine-grained tracking does. We can more precisely quantify the precision improvement by observing that, out of the 500 Android applications analyzed, MorphDroid reported 171 (34%) as vulnerable according to the most restrictive policy, `pol1`, whereas the least conservative policy, which is also the most realistic one in that it accounts for real-world declassification rules, marks only 4 of the apps as misbehaved ($< 1\%$). This significant delta justifies the motivation behind the creation of MorphDroid. This result shows also that the most popular applications usually access and carefully manage confidential data by applying various transformations and leaking it to precise sinks. In this way, they preserve both the functionality of the app and the confidential information of the user. In Hypothesis 2, we had conjectured a reduction in the total number of the false positives of at least 50% when switching from a policy with coarse granularity to a fine-grained policy that accounts for declassification rules. Our evaluation show that the number of results reported by the analysis when accounting for declassification rules decreases by 97.7%.

5.4.3 Evaluation of H3

The programs we considered are all real-world commercial applications from Google Play, with an average size of 1.4 MLOC (more accurately: 1,435,141 LOC), for a total of more than 700 MLOC. As for the time MorphDroid took to complete the analysis, we broke the time into three segments:

- Phase 1—which we defined as the part of the analysis in which WALA builds the call graph and points-to graph modeling the execution of the program—takes $7.6''$ on average.
- Phase 2—the taint analysis part of MorphDroid that also accounts for declassification by modeling the various correlations between different units of private data as well as any semantic transformation on that data—takes $10.8''$ on average.
- Finally, Phase 3—the policy-infringement analysis part of MorphDroid, which, thanks to our optimization, only needs to work on the serialized data output at the end of Phase 2—takes 4 msec on average

Therefore, MorphDroid takes on average 18.4” to analyze an Android app. In addition, in the slowest case MorphDroid took less than 7 minutes to terminate the analysis. These results validate Hypothesis 3.

6. RELATED WORK

We survey related research in the area of mobile information-flow-based privacy analysis. This includes both static and dynamic solutions, and related security/privacy policy languages.

Static analysis. In Section 2.2, we discuss the technical and conceptual differences between our approach and taint analysis [8, 18, 19]. Beyond this general discussion, we now refer to recent static privacy solutions with unique analysis features.

The FlowDroid tool [2] performs static taint analysis of Android applications. It features a precise model of the Android application lifecycle, which enables accurate handling of callbacks. FlowDroid reduces the number of false positives via a combination of context, flow, field and object sensitivity [14]. Unlike MorphDroid, however, FlowDroid performs taint analysis (cf. Section 2.2), and does not support fine-grained privacy policies.

Lin, *et al.* [11] investigate the feasibility of identifying a small set of privacy profiles, via clustering techniques, as a way of helping users manage their privacy preferences. Similarly to MorphDroid, their approach relies on static code analysis and is motivated by the need for expressive privacy preferences. Unlike MorphDroidhowever, where the role of static analysis is to verify policy compatibility, the static analysis of Lin, *et al.* aims to distinguish between use of permissions for core functionality versus advertising and social netweoks. Also, the inferred profiles are not at the granularity of atomic privacy units.

Cortesi, *et al.* [3] introduce a theoretical model for information-flow analysis that tracks the amount of confidential data that is possibly leaked by a mobile application. The main novelty of their 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. This gives rise to a lazy approach with finer granularity, which may reduce false positives with respect to state-of-the-art information-flow analyses.

Dynamic analysis. The state-of-the-art system for real-time privacy enforcement on mobile devices is TaintDroid [6, 15]. TaintDroid features tolerable run-time overhead of about 10%, and can track taint flow not only through variables and methods but also through files and messages passed between applications. TaintDroid has been used, extended and customized by several follow-up research projects [10]. One of these extensions, the Kynoid system [1], augments TaintDroid with user-defined security policies, which include for example temporal constraints on data processing as well as restrictions on destinations to which data is released. Neither TaintDroid nor any of its extensions support granular privacy specifications.

The BayesDroid system for real-time privacy enforcement by Tripp and Rubin [20] replaces taint tracking with Bayesian reasoning about information flows, where privacy judgments are based on value similarity between private and released values. Analogously to MORPHDROID, BayesDroid also accounts for value transformations, though the techniques are fundamentally different and do not rely on information-flow tracking.

AppIntent [21] also aims at detecting privacy violations in Android applications. AppIntent is also informed by the observation that simply enforcing information-flow security may lead to overly conservative results. The difference between AppIntent and MORPHDROID is that AppIntent—by providing the analyst with the sequence of GUI manipulations corresponding to the events that lead to transmission of data—attempts to determine if the release is user intended or not, whereas MORPHDROID provides an expressive specification language.

McCamant and Ernst [12] propose an analysis for determining *how much* information about a program's secret inputs is revealed by its public outputs. In this sense, their approach can be defined as a quantitative information-flow-security analysis. In contrast, MORPHDROID performs structural analysis of the data. Rather than focusing on a quantitative measure of the data that is released, MORPHDROID allows for specific units of data to be tracked.

Policy languages. Over the years, researchers have proposed a number of security-oriented specification languages [4, 5, 9, 17]. These cover different security measures, including authentication, authorization, auditing, delegation, obligation, access control and provenance. Only some of these languages (e.g., DLSS [4] and Fable [17]) enable the expression of integrity and confidentiality policies. Contrary to the MORPHDROID policy language, however, these languages allow for neither distinguishing individual units of privacy inside the data manipulated by a program nor taking into account data transformations that structurally modify the data.

7. CONCLUSION AND FUTURE WORK

In this paper, we presented MORPHDROID, a static-analysis solution for Android that verifies mobile applications against fine-grained privacy policies. MORPHDROID tracks flows of fine-grained privacy units while addressing important challenges, including (i) detection of correlations between different units and (ii) modeling of semantic transformations of private data. The experimental results underline both the precision and scalability of MORPHDROID.

In the future, we plan to expand on this work in multiple directions. Today's mobile market is dominated by a combination of Android vendors and by Apple's iOS platform, so much so that virtually any mobile application is released for both the Android and the iOS ecosystems. In order for MORPHDROID to apply to any application regardless of the target platform, we are working towards extending MORPHDROID to iOS.

8. REFERENCES

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[2] 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*, 2014.

[3] A. Cortesi, P. Ferrara, M. Pistoia, and O. Tripp. Datacentric Semantics for Verification of Privacy Policy Compliance by Mobile Applications. In *VMCAI*, 2015.

[4] F. Cuppens and R. Demolombe. A Deontic Logic for Reasoning about Confidentiality. In *DEON*, Jan. 1996.

[5] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. In *POLICY*, 2001.

[6] 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.

[7] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. SCanDroid: Automated Security Certification of Android Applications. Technical report, CS-TR-4991, Department of Computer Science, University of Maryland, 2009.

[8] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the world wide web from vulnerable javascript. In *ISSTA*, 2011.

[9] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A Logical Language for Expressing Authorizations. In *S&P*, 1997.

[10] J. Jung, S. Han, and D. Wetherall. Short Paper: Enhancing Mobile Application Permissions with Run-time Feedback and Constraints. In *SPSM*, 2012.

[11] J. Lin, B. Liu, N. M. Sadeh, and J. I. Hong. Modeling Users' Mobile App Privacy Preferences: Restoring Usability in a Sea of Permission Settings. In *SOUPS*, 2014.

[12] S. McCamant and M. D. Ernst. Quantitative Information Flow as Network Flow Capacity. In *PLDI*, 2008.

[13] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.

[14] B. G. Ryder. Dimensions of Precision in Reference Analysis of Object-Oriented Programming Languages. In *CC*, 2003.

[15] D. Schreckling, J. Posegga, J. Köstler, and M. Schaff. Kynoid: Real-Time Enforcement of Fine-grained, User-defined, and Data-centric Security Policies for Android. In *WISTP*, 2012.

[16] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

[17] N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A Language for Enforcing User-defined Security Policies. In *S&P*, 2008.

[18] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri. Andromeda: Accurate and Scalable Security Analysis of Web Applications. In *FASE*, 2013.

[19] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: Effective Taint Analysis of Web Applications. In *PLDI*, 2009.

[20] O. Tripp and J. Rubin. A Bayesian Approach to Privacy Enforcement in Smartphones. In *USENIX Security*, 2014.

[21] 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*, 2013.