A Framework for Static Detection of Privacy Leaks in Android Applications

Christopher Mann
Modeling and Analysis of Information Systems
TU Darmstadt, Germany
christophermann@web.de

Artem Starostin
Modeling and Analysis of Information Systems
TU Darmstadt, Germany
starostin@mais.informatik.tu-darmstadt.de

ABSTRACT

We report on applying techniques for static information flow analysis to identify privacy leaks in Android applications. We have crafted a framework which checks with the help of a security type system whether the Dalvik bytecode implementation of an Android app conforms to a given privacy policy. We have carefully analyzed the Android API for possible sources and sinks of private data and identified exemplary privacy policies based on this. We demonstrate the applicability of our framework on two case studies showing detection of privacy leaks.

Categories and Subject Descriptors

D.2.0 [**Software Engineering**]: General—*Protection mechanisms*

General Terms

Security

Keywords

Information flow security, Static analysis, Privacy, Android

1. INTRODUCTION

Nowadays, mobile devices and, particularly, smartphones are becoming universal and omnipresent. Gartner[1], the world's leading information technology research and advisory company, predicts [8] over 500 million smartphones to be sold in 2012. Among those, the smartphones' operating system market share is predicted to be headed by Symbian (37,4%), Android (18%), BlackBerry (13.9%), and iPhone (13.6%), with Android being the fastest growing.

The range of applications in the rapidly growing market of programs for mobile devices vary from clients for online banking to apps for monitoring health conditions, from sports betting apps to personal calendars. Obviously, many such applications are intended to operate on users' private information. However, recent studies [18, 16] show that smartphones' operating systems still do not provide clear visibility of how third-party applications use the owner's private data stored on the smartphone. Moreover, the experimental results [15] outline a frustrating phenomenon: a large fraction of mobile applications seem to abuse user's privacy by sending the data collected on a smartphone to advertising companies without notifying the smartphone's owner. A public opinion poll [15] accompanying this study showed that 98,5% of more than 12.000 respondents would like to be informed whether an application sends data to other companies.

In a response to this problem, Enck et al. recently presented TaintDroid [5], an information flow tracking system for real-time privacy monitoring on Android smartphones. TaintDroid has been successfully applied to identify potential misuse of user's private information across a sample of stock Android applications. However, there are two fundamental limitations of the undertaken approach of dynamic information flow analysis. First, it always produces some performance overhead during the application run time. Second, it complicates to recognize implicit leaks [13, 9].

In this paper we address the issue from a different perspective: we apply static information flow analysis to the Android apps' Dalvik bytecode. This bytecode is stored directly in the application packages and available for analysis directly on the phone. The novel research results of the paper are:

- a systematization of a complete Dalvik VM instruction set (218 instructions) into a significantly smaller abstract instruction set (61 instructions) capturing relevant information flow aspects,
- a security type system for the abstract instruction set detecting explicit (in the current version) information leaks, and
- a carefully identified set of sources and sinks of private information in the Android API (version 2.2) from which a privacy policy is defined.

Based on these three contributions we have implemented in Java a framework featuring the security type system and a privacy policy generator. The latter features a carefully engineered inference algorithm for field and method security signatures. We have demonstrated the applicability of the framework on a number of small self-developed Android applications. We believe that with this work we lay both formal and practical foundations for application of static

[1] `http://www.gartner.com`

This is a preprint. Original publication to appear in the Proceedings of 27th Symposium on Applied Computing (SAC): Computer Security Track.

.

information flow analysis techniques to detect privacy violations in stock Android applications.

Related Work.

The related work for this paper can be grouped into two categories: (i) Android security, and (ii) information flow analysis for bytecode and assembly languages. As Enck [4] provides a comprehensive overview of the state of the art in the area of Android security, we limit this paragraph to highlighting the second category. Genaim and Spoto [6] reports on the first implementation of an information flow analysis for Java bytecode. Medel et al. [11] introduces a type system for checking noninterference of simple typed assembly programs. Barthe et al. [2] presents a noninterference property and a security type system for simplified Java bytecode.

Outline.

The remainder of this paper is organized as follows. After giving a bird's eye overview of the framework in Sect. 2 we identify in Sect. 3 a set of private information sources and a sinks on a typical Android smartphone. In Sect. 4 we describe the abstract instruction set for the Dalvik VM and present a security type system to track explicit information flows. In Sect. 5 we show in two case studies how privacy leaks are detected with the help of our framework. We conclude and point out directions for future work in Sect. 6. We conclude and point out directions for future work in Sect. 6.

2. FRAMEWORK OVERVIEW

The core component of the framework presented in this paper (Fig. 1) is the implementation of a security type system for tracking explicit information flows, i.e., those which result from passing the information in assignments, in the Dalvik bytecode of Android apps. The implementation of the security type system takes as input a privacy policy which defines method signatures and field signatures for the API and the application. These signatures define which method parameters, return values, or fields allowed to contain private information. The tool then checks that all methods in the application respect the specified signatures.

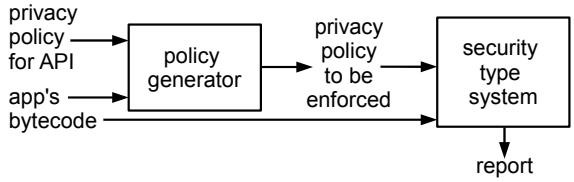

Figure 1: Sketch of the Framework

The policy enforced by the security type system comes from the policy generator. The generator itself requires security signatures for the API methods invoked by the application. It uses afterwards a signature inference algorithm to generate the signatures of the methods and fields implemented in the application. The security signatures for the API methods include those which might be used to extract and transmit users' private information. In the next section we analyze a set of such methods used in our work.

3. PRIVATE INFORMATION ON DEVICE

In this section we identify privacy relevant information which is typically stored on an Android mobile device, and how this information can be retrieved with the API. Further, we identify the ways to send information to the outside world. The identified sources and sinks are the endpoints for the information flow analysis performed by the framework.

3.1 Private Information Sources

Several different kinds of private data are available to applications on an Android smartphone. Below we present five categories we have identified for this work. In each category we reference relevant Android API functions and describe how the private data is obtained. This is not trivial since there are several different ways to retrieve data from the Android API and, therefore, types of information sources. We distinguish *return values* of functions, parameters of *callback methods*, *content resolvers* used to iterate over data sets, and *intent* messages for inter-process communication. The additional information on these peculiarities of Android's middleware could be found in the Android Developer's Guide [7].

Note that the resulted lists of private information sources is obtained by the analysis of Android API in the version 2.2 and reflect our subjective view of which user's data should be denoted as private and, thus, might be incomplete.

Location Data.

Typically, location data can be retrieved either from a GPS receiver which is available on most smartphones, or it can be approximated based on the location of the used base station for either the cellular network or WLAN. The class LocationManager gives access to location data and supports different providers, like a GPS receiver or the mobile phone network. The method getLastKnownLocation(String provider) returns a Location object representing the last location fix acquired by the given provider. Alternatively, the location data can be accessed by using the class LocationListener which can be extended by the application to monitor the phone's location. The application can overwrite the method onLocationChanged(Location location) which is called by the application framework to inform the application about a new location fix contained in the parameter location. The monitoring is activated by calling the method requestLocationUpdates(..., LocationListener listener) with an instance of the application class extending the class LocationListener as the parameter listener.

Unique Identifiers.

A cell phone stores several unique identifiers like the International Mobile Equipment Identity (IMEI) which is globally unique for each GSM based mobile phone, or the International Mobile Subscriber Identity (IMSI) which is globally unique for each subscriber of GSM services. These unique identifiers can be used to identify devices and subscribers very accurately. The class TelephonyManager gives access to such identifiers. We consider that the following methods return privacy relevant data: getDeviceId() returns the IMEI of the phone; getDeviceSoftwareVersion() returns the IMEISV which is the IMEI extended with two digits giving the version number of the phone's software; getLine1Number() returns the primary phone number associated with this phone; getNetworkCountryIso() returns country code of the network in which the phone is currently logged in; getSubscriberId() returns the IMSI from the SIM card.

Call State.

Android applications can watch the current state of the cell phone. The applications are then informed about the starting and ending of incoming and outgoing calls. The class PhoneStateListener can be extended by the application to monitor the incoming phone calls. When extending the class, the application can overwrite the method onCallStateChanged(int state, String incomingNumber) which is called by the application framework if the call state changes. The parameter incomingNumber contains the phone number of the incoming call. The monitoring is activated by calling the method TelephonyManager.listen(PhoneStateListener listener, int events) with an instance of the class which extends the class PhoneStateListener as the parameter listener.

Authentication Data.

Android provides a single sign-on system where applications can request authentication tokens from the Android OS to access accounts at Internet services without having the user to enter the credentials. The class AccountManager is used to acquire an authentication token for a user account stored on the Android phone. If the user stored the password for the account on the phone or an authentication token is already cached, the user will not be informed about the request and the token will be provided in the background. An authentication token can be requested with the method getAuthToken(...) which returns an AccountManagerFuture.

Contact and Calendar Data.

Android smart phones are almost always used for personal information management and therefore store addresses and phone numbers as well as appointments. Such data is typically provided by a content provider and can be accessed with query operations which use SQL-like syntax to specify the retrieved data. A query can be executed, for instance with the methods Activity.managedQuery(Uri uri, ...) or ContentResolver.query(Uri uri, ...). The parameter uri specifies with an URI the content provider and data set to be accessed. Normally, these URIs are retrieved from class constants. For example, the constant ContactsContract.Contacts.CONTENT_URI contains the URI referencing the content provider to access the contacts stored in the phone. All these methods return a Cursor object which is used to iterate over the retrieved result set.

3.2 Information Sinks

As long as private information is stored in some variable of an Android application, several possibilities arise for it to flow to the outer world. Below we describe a set of information sinks we considered in our work.

SMS Communication.

The class SMSManager can be used to send SMS to other phones. The method sendTextMessage(String destinationAddress, ... , String text, ...) sends an SMS with the content text to the phone number specified by destinationAddress.

File Output.

Android applications can write to files which are globally readable. This allows other applications to access the written data. The Android application framework contains the classes from the java.io package which provide an unified interface for writing data to different sinks, like files or network sockets. A sink is represented by a OutputStream object which provides methods to write binary data to the sink. In most cases, the subclasses of Writer are used to wrap the output streams as they provide a more convenient character based interface. For example, the method write(String str) writes the string provided by the parameter str to the sink. An output stream for writing a file can be acquired by creating a new FileOutputStream with the constructor FileOutputStream(String path). The parameter path expects a string giving the file to write to.

Network Communication.

Android applications can access the Internet via several different APIs including a socket-like API and a high level HTTP client. The class Socket models the endpoint of a network connection. A new network connection can be created by calling the constructor Socket(String dstName, int dstPort). The parameter dstName expects a string giving the IP-address or domain name of the destination and the parameter dstPort expects a port number on which the destination is to be contacted. With the method getOutputStream (), an output stream is retrieved from an open socket, which can be used to write to the open network connection. The written data is then transmitted to the destination.

Intents.

Android applications can send Intent objects containing data to other components. The class Context contains the methods startActivity, startService, and sendBroadcast which can be used to activate other components of the same or different application. Each of these methods expects an Intent object as the single parameter. It can contain arbitrary data which is transmitted to the activated component.

Content Resolver.

The content provider/resolver API can also be used to modify the stored data with SQL-like statements. Besides retrieving data, the class ContentResolver can also be used to store data at a content provider. The method insert(Uri uri, ContentValues values) can be used to store an additional entry in the data set managed by the content provider. The method update(Uri uri, ContentValues values, ...) can be used to modify a subset of the already existing entries in the data set. The parameter uri specifies the content provider and data set, whereas the parameter values expects a ContentValues object containing the values to store in the content provider's data set.

4. ENFORCING PRIVACY POLICIES

In order to statically analyze whether a Dalvik bytecode program respects a provided privacy policy we adopt the concept of a security type system [17]. Similarly to data type systems [12] which ensure that operations are applied only to correct instances of data types, security type systems ensure that information represented by the language constructs does not flow such that confidential information becomes public. This includes checking of *explicit* flows when information is passed in assignments as well as *implicit* flows when information is passed via control flow structures [3].

Security type systems are defined with the typing rules which have to cover all language constructs, i.e., all DVM bytecode instructions in our case. In order to reduce the

size of the type system we reorganized the complete DVM instruction set [14] of 218 instructions into an abstract one of 61 instructions by grouping several DVM instructions under one abstract instruction.

4.1 Abstract Instruction Set

The abstract instruction set [10, Ch. 3] captures all information flow aspects of DVM bytecode while leaving out irrelevant details. The nature of the abstractions we have made is as follows. Since Dalvik VM has been optimized for small bytecode size and execution time, many DVM instructions exist in several different versions which only differ in the maximal size of their parameters, but not in their semantics. For the analysis, these instructions are grouped together, which greatly reduces the number of instructions to handle. Besides that, all instructions with similar semantics, e.g., different binary arithmetic operations, are grouped together, if the semantical differences are irrelevant for the information flow analysis. Further, we have abstracted the varying instruction lengths. Finally, several other implementation details are also abstracted, like the complex way to build type, method, and field identifiers out of several references, or the way in which tables for switch instructions are stored. Finally, several other implementation details are also abstracted, like the complex way to build type, method, and field identifiers out of several references, or the way in which tables for switch instructions are stored.

4.2 Program Model

Let the code of a method to be analyzed consists of N_{PP} of program points, each corresponding to a single DVM instruction. The set $PP = \{i \in \mathrm{N} \mid 1 \leq i \leq N_{PP}\}$ contains all these program points. A method is modeled by a function P which maps each program point to its instruction and and a set E, which contains the exception handlers of the method (defined below).

Since DVM is a register-transfer machine, most of its instructions take registers as parameters and store their result to a register. Let N_{reg} be the number of registers, that are available to the method. The set $ID = \{i \in \mathrm{N} \mid 1 \leq i \leq N_{reg}\}$ contains the register identifiers, that are valid inside a method. To model semantics of instructions, we extend the set ID with three invisible registers: $ID' = ID \cup \{res_L, res_H, excp\}$. Besides the identifiers for registers, we introduce the sets ID_C, ID_F, and ID_M which contains identifiers for classes, fields, and methods, respectively. Method identifier consists of the method name and the method's descriptor which gives the type of each parameter and the return type. This design decision allows us to properly handle Java's method overloading.

Now, the set E is defined as $E \subseteq PP \times ID_C \times PP$. The entry $(i, c, j) \in E$ denotes, that an exception assignment compatible to the class c occurring at program point i is handled by the exception handler starting at the program point j. The set is expected to be consistent in the way that for every program point and every exception the exception handler is unique.

4.3 Security Type System

The type system given in this thesis is inspired by the type system in [2] for a JVM-like language. The lattice $SL = \{L, H\}$ with $L \sqsubseteq_{SL} L, L \sqsubseteq_{SL} H, H \sqsubseteq_{SL} H$ defines the available security levels. Any privacy relevant information is given the security level H, all other information the security level L. The function $\sqcup_{SL} \colon SL \times SL \to SL$ returns the least upper bound of the two parameters. The set $ST = SL \cup \{rt :: ct \mid rt \in SL, ct \in SL\}$ contains the possible security types. This definition follows [1] and allows more precise handling of arrays. For non-array types the security type is just the security level of the stored data. For arrays the security type is $rt :: ct$, where rt gives the security level of the array's reference and length, and ct gives the security type of the array's content. The type system is restricted to one dimensional arrays to simplify the implementation. The relation $\sqsubseteq_{ST}$ and the least upper bound function $\sqcup_{ST}$ are defined similarly. The set $PPST = ID' \to ST$ defines the possible types for a program point. A type for a program point is a function $ppst \in PPST$, that maps each register to the register security type it has after the execution of the program point's instruction. The relation $\sqsubseteq_{PPST}$ and the function $\sqcup_{PPST}$ are just the pointwise extensions of the relation $\sqsubseteq_{ST}$ and the function $\sqcup_{ST}$. In the following, the relation $\sqsubseteq$ and the function $\sqcup$ are used, as it should be clear from the context, which relation or function is meant.

Further, the security types for fields and result values of method invocations are required. These signatures will be provided by the sets $FS \subseteq ID_C \times ID_F \times ST$ and $MS \subseteq ID_C \times ID_M \times (ST^*) \times ST$. By $(c.f, s) \in FS$ we denote that the field f in the class c has the security type s. We perform the analysis classwise and do not distinguish particular instances of a class. By $(c.m, (p_1, \ldots, p_n), r) \in MS$ we denote that if the method m in the class c is invoked with n parameters with the security types $p_1, \ldots, p_n$, the return value has the security type r.

The judgment for instructions $P, MS, FS, r \vdash i : ppst \to ppst'$ with $r \in ST$ and $ppst, ppst' \in PPST$ denotes that for given sets MS, FS the type of the program point must be $ppst'$ after the execution of the instruction $P(i)$ at program point i, if it was $ppst$ before the instruction's execution and the instruction's execution does not violate any of the security types given by the sets MS and FS. Additionally, if the instruction $P(i)$ is a return instruction, the security type of the returned value is less than or equal to r.

The purpose of the type system we design is to ensure that a method respects a given method signature for given sets MS and FS. The judgment for methods $MS, FS \vdash c.m, (p_1, \ldots, p_n), r$ denotes that the method $c.m$ respects the method signature $(c.m, (p_1, \ldots, p_n), r)$ for given sets MS and FS. Formally, this judgment is defined as

$$\exists mst \in PP \to PPST \,.\, \forall i, j \in PP \,.\, \exists ppst' \in PPST \,.$$
$$i \to j \Longrightarrow P, MS, FS, r \vdash j : mst(i) \to ppst' \wedge ppst' \sqsubseteq mst(j)$$

Figure 2 lists selected typing rules of our security type system which are used to deduce the above introduced judgments for instructions. In the figure, $\oplus$ is the notation for function update and $at : PP \rightharpoonup ST$ is a partial function which gives for each program point where an array is created the security type for the content of the new array. Altogether we have defined 61 typing rules corresponding to the number of instructions in the abstract instruction set.

5. CASE STUDIES

In this section we demonstrate how our framework detects privacy violating information flow on example of two small applications with undesired behavior. Although the analysis

BINOP

$$\frac{P(i) = \texttt{binop}\ \ v_a, v_b, v_c}{P, MS, FS, r \vdash i : ppst \to ppst \oplus \{v_a \mapsto ppst(v_b) \sqcup ppst(v_c)\}}$$

IGET

$$\frac{P(i) = \texttt{iget}\ \ v_a, v_b, c.f \qquad (c.f, st) \in FS}{P, MS, FS, r \vdash i : ppst \to ppst \oplus \{v_a \mapsto ppst(v_b) \sqcup st\}}$$

IPUT

$$\frac{P(i) = \texttt{iput}\ \ v_a, v_b, c.f \qquad (c.f, st) \in FS \qquad ppst(v_a) \sqcup ppst(v_b) \sqsubseteq st}{P, MS, FS, r \vdash i : ppst \to ppst}$$

INVOKE

$$\frac{P(i) = \texttt{invoke}\ \ v_a, v_b, v_c, v_d, v_e, n, c.m \qquad x = (v_a, v_b, v_c, v_d, v_e) \qquad (c.m, (ppst(x_1), \ldots, ppst(x_{n-1})), st) \in MS}{P, MS, FS, r \vdash i : ppst \to ppst \oplus \{res_L \mapsto st, res_H \mapsto st\}}$$

THROW

$$\frac{P(i) = \texttt{throw}\ \ v_a \qquad ppst(v_a) = L}{P, MS, FS, r \vdash i : ppst \to ppst}$$

NEW-ARRAY

$$\frac{P(i) = \texttt{new-array}\ \ v_a, v_b, t}{P, MS, FS, r \vdash i : ppst \to ppst \oplus \{v_a \mapsto ppst(v_b)::at(i)\}}$$

ARRAY-LENGTH

$$\frac{P(i) = \texttt{array-length}\ \ v_a, v_b \qquad ppst(v_b) = rt::ct}{P, MS, FS, r \vdash i : ppst \to ppst \oplus \{v_a \mapsto rt\}}$$

Figure 2: Selected Typing Rules

is performed on the level of Dalvik bytecode we present the source code of applications to increase readability of the examples. Each example is accompanied with a privacy policy assigning privacy levels to each parameter and return value of the occurring methods. However, due to space limitations we do not present the complete policies, but rather describe textually the assigned security levels.

Leaking Device ID through a Network Connection.

The code in Fig. 3 shows a small activity which retrieves a unique device identifier and sends it to the Internet by writing to an output socket stream. As identified in Sect. 3, device ids are considered private, and, hence, the security policy assigns the security level H to return value of the method getDeviceId() (line 10). To protect private data from sinking over the socket the security policy allows the method write(String str) only to be called with a parameter of security type L (line 15). With the provided security policy, the analysis terminates with the report that a signature is missing which allows the invocation of write(String str) with a parameter of type H. This means, that the implementation tracked the information flow through the method and the malicious leakage to the Internet was discovered.

Leaking Phone's Location by Sending an Intent.

The code in Fig. 4 shows another small activity which subscribes itself as a location listener on creation. The callback

```
1  public class MainScreen extends Activity {
2    public static final String HOST = "10.0.2.2";
3    public static final int PORT = 2000;
4
5    @Override
6    public void onCreate(Bundle savedInstanceState) {
7      super.onCreate(savedInstanceState);
8      TelephonyManager telephonyManager = (TelephonyManager)
9        getSystemService(Context.TELEPHONY_SERVICE);
10     String imei = telephonyManager.getDeviceId();
11     try {
12       Socket socket = new Socket(HOST, PORT);
13       BufferedWriter writer = new BufferedWriter
14        (new OutputStreamWriter(socket.getOutputStream()));
15       writer.write(imei);
16       writer.newLine();
17       writer.close();
18     } catch (FileNotFoundException e) { e.printStackTrace(); }
19       catch (IOException e) { e.printStackTrace(); }
20   }
21 }
```

Figure 3: Source Code of Case Study 1

method onLocationChanged(Location location) is called by the application framework if a new location fix is available. Inside the callback method a new Intent object is created which contains an URL with the coordinates supplied by the Location object from the callback method's parameters. The Intent object is then used to start a new activity which will receive this intent. The activity to be started is determined by the OS, and as the intent contains a web URL, the web browser is started to display the given URL. Since Location objects always contain a location fix and thereby privacy relevant information, the security policy assigns the security level H to the return values of the methods getLongitude() and getLatitude() (lines 15 and 17). An Intent object is a potential information sink as it is used as transport object for the communication with other components. Therefore, the security policy must not allow to create new Intent objects containing data with the security type H or to store data with the security type H into an existing Intent object. Consequently, the security policy allows invocation of the method parseUri(String uri, int flags) only with parameters of the security type L, as this method creates a new Intent object containing the URI as a parameter (line 14).

The return values of the methods (Uri.encode(String s)) and String.valueOf(double value) only depend on the given parameter and the methods do not have any side effects. Therefore, it is secure to allow their invocation with the security types L and H (lines 15 and 17). The security type of the return value must be equal to the security level of the parameter. As for the method concat(String string), it is secure to allow the invocation of the method with parameters of any security type (lines 15–17). The security type of the return value must then be the least upper bound of the parameter security types. This is reflected by the corresponding entries in the security policy.

With the described security policy, the analysis of the method onLocationChanged(Location location) terminates with the report that a signature is missing which allows the invocation of the method Intent.parseUri(String uri, int flags) with a first parameter with the security type H. This means, that the implementation tracked the information flow through the method and discovered the malicious information flow to the newly created Intent object.

```
public class MainScreen extends Activity implements LocationListener {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    LocationManager locManager = (LocationManager)
      getSystemService(Context.LOCATION_SERVICE);
    locManager.requestLocationUpdates(LocationManager.GPS_PROVIDER,
      0, 0, this);
  }

  @Override
  public void onLocationChanged(Location location) {
    try {
      Intent i = Intent.parseUri("http://10.0.2.2:2000/gps?lat="
        .concat(Uri.encode(String.valueOf(location.getLatitude())))
        .concat("&long=")
        .concat(Uri.encode(String.valueOf(location.getLongitude()))), 0);
      startActivity(i);
    } catch (URISyntaxException e) {}
  }
  ...
}
```

Figure 4: Source Code of Case Study 2

6. SUMMARY AND FUTURE WORK

In this paper we have presented a framework for detection of privacy-violating information flow inside Android applications by means of static analysis of the application's bytecode. The framework is based on a carefully crafted security type system and supports the complete instruction set of the Dalvik virtual machine. We have analyzed the Android API and identified a meaningful set of private information sources and sinks which we have used to specify privacy policies to be enforced by the framework. A couple of case studies have shown the applicability of the framework. We conclude by a number of directions of our future work comprising scientific as well as engineering efforts.

Implicit Leaks. The type system presented in this paper is designed to track only explicit information flow. However a situation when an application sinks user's private data via an implicit leak is easily imaginable. One of our current efforts is concentrated on a type system for Dalvik bytecode which will be able to detect implicit leaks due to the program control flow.

Semantical Foundations. We work on a formal specification of our execution model which will allow us to provide provable privacy and security guarantees for the programs accepted by our type system(s).

Inference of Signatures for API. It would be worthwhile to have a tool for generation of method and field signatures for the application framework automatically. Two different ways seem to be possible. The first way would be to analyze the Java source code of the application framework. The second way would be to extract the DVM bytecode of the application framework from a phone and analyze it with a modified version of the tool presented in the paper. Independently of the selected way, the difficulty of handling calls to functions written only in the native code will occur.

Infrastructure. A custom application installer for Android could be developed which will use the results of this paper to check an application for privacy-violating information flows prior to installing it.

Acknowledgements.

This work was supported by the German Research Foundation (DFG) under the project RSCP within the Priority Programme 1496 RS^3.

7. REFERENCES

[1] A. Askarov and A. Sabelfeld. Security-Typed Languages for Implementation of Cryptographic Protocols: A Case Study. In *Proc. of 10th ESORICS*, pages 197–221. Springer, 2005.

[2] G. Barthe, D. Pichardie, and T. Rezk. A Certified Lightweight Non-interference Java Bytecode Verifier. In *Proc. of 16th ESOP*, pages 125–140. Springer, 2007.

[3] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Commun. of the ACM*, 20(7):504–513, 1977.

[4] W. Enck. Defending Users Against Smartphone Apps: Techniques and Future Directions. In *Proc. of 7th ICISS*. Springer, 2011.

[5] W. Enck et al. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. of 9th OSDI*, pages 393–407. USENIX, 2010.

[6] S. Genaim and F. Spoto. Information Flow Analysis for Java Bytecode. In *Proc. of 6th VMCAI*, pages 346–362. Springer, 2005.

[7] Google Inc. The Android Developer's Guide. `http://developer.android.com/guide/`, 2011.

[8] M. Hamblen. Symbian, Android Will Be Top Smartphone OSes in '12, Gartner Reiterates. `http://www.computerworld.com/s/article/9139301/Symbian_Android_will_be_top_smartphone_OSes_in_12_Gartner_reiterates`, 2009.

[9] J. Ligatti, L. Bauer, and D. Walker. Edit Automata: Enforcement Mechanisms for Run-time Security Policies. *Int. J. of Inf. Sec.*, 4(1):2–16, 2005.

[10] C. Mann. A Static Framework for Privacy Analysis of Android Applications. Bachelor's thesis, TU Darmstadt, 2011.

[11] R. Medel, A. B. Compagnoni, and E. Bonelli. A Typed Assembly Language for Non-interference. In *Proc. of 9th ICTCS*, pages 360–374. Springer, 2005.

[12] J. C. Mitchell. *Handbook of Theoretical Computer Science*, chapter Type Systems for Programming Languages, pages 365–458. MIT Press, 1990.

[13] F. B. Schneider. Enforceable Security Policies. *ACM Trans. on Inf. Sys. Sec.*, 3(1):30–50, 2000.

[14] The Android Open Source Project. Bytecode for the Dalvik VM. `http://source.android.com/tech/dalvik/dalvik-bytecode.html`, 2007.

[15] The Wall Street Journal Blogs. What They Know - Mobile. `http://blogs.wsj.com/wtk-mobile/`, 2010.

[16] S. Thurm and Y. I. Kane. Your Apps Are Watching You. Available at `http://online.wsj.com/`, 2010.

[17] D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *J. of Comp. Sec.*, 4(3):1–21, 1996.

[18] A. Yeager. Researchers Find Phone Apps Sending Data without Notification. `http://www.physorg.com/news204978481.html`, 2010.