The Transitivity-of-Trust Problem in Android Application Interaction

Steffen Bartsch
TU Darmstadt, Germany
steffen.bartsch@cased.de

Bernhard Berger, Michaela Bunke, Karsten Sohr
Technologie-Zentrum Informatik, Universität Bremen, Germany
{berber|mbunke|sohr}@tzi.de

***Abstract*—Mobile phones have developed into complex platforms with large numbers of installed applications and a wide range of sensitive data. Application security policies limit the permissions of each installed application. As applications may interact, restricting single applications may create a false sense of security for end users, while data may still leave the mobile phone through other applications. Instead, the information flow needs to be policed for the composite system of applications in a transparent manner. In this paper, we propose to employ static analysis, based on the software architecture and focused on data-flow analysis, to detect information flows between components. Specifically, we aim to reveal transitivity-of-trust problems in multi-component mobile platforms. We demonstrate the feasibility of our approach with two Android applications.**

I. Introduction

Powerful and well-connected smartphones are becoming increasingly common. Their features are provided by focused applications that users can easily install from application market places. With hundreds of thousands of applications available, however, there is only limited control over the quality and intent of those applications. Mobile code and extensibility is one of the key issues that increase the complexity of software security. To counter this threat, mobile operating systems impose security restrictions for each application. The Android mobile operating system is one of the major smartphone platforms. The Android security model enforces the least-privilege principle through application-level permissions that can be requested by the applications. End users need to grant the permissions at install time and decide on the adequacy of the required permissions and the trustworthiness of the individual application. Often it cannot be assumed that a user fully understands the risks related to granting permissions to applications [1]. This situation becomes more complex if two or more applications collude, each single application only requiring innocuous permissions. In particular, the user has little knowledge about the consequences regarding the transitivity of permission granting. As depicted in Figure 1, an application (1) with only local permissions (2) may proxy sensitive data (3) through third party applications and services (4) to external destinations (5). The inter-component cooperation is an important concept on the Android platform, but the user needs transparency concerning information flows between applications.

The above-described issue of missing transparency poses a risk with the spreading of smartphones, the large numbers of available applications and the prevalent custom of installing applications from untrustworthy sources. Attacks will become more likely in the future due to the different kinds of sensitive data stored on the phones, ranging from online banking and business application credentials to communication data and location information. The threat is further increased by the number of data channels, such as the short message service, E-mail or Web access that allow the flow of information out of the device context. As a consequence, tools are needed that reveal such information flows on mobile phones. Note that these information flows not necessarily need to be malicious—in fact, a higher transparency is of importance here.

In this paper, we describe an approach to making information flows explicit between different applications and out of the platform. This way, problems can be revealed that are induced by interacting applications and permission transitivity. Our contributions are in particular:

1) Static analysis of Android applications w.r.t. interactions and transitivity of permissions.
2) Employing a two-layer approach for static analysis, combining code-level analyses (on the bytecode) with analyses at the more abstract level of the software architecture.
3) Demonstrating the feasibility of our approach with the help of two applications from Google Play.

Our analysis method can be considered complementary to other static code analysis approaches that aim to detect implementation bugs [2]. We aim to focus on the aspect of program comprehension for security assessments in making transparent interactions between different applications.

The rest of this paper is structured as follows. In Section II, we briefly describe the background of Android, before discussing the possible data sources and sinks of Android applications. In Section IV, we present our approach to the security analysis of Android applications, followed by a case study. After listing the related work, we conclude in Section VII.

II. Android Concepts

Applications on the Android platform are developed with Java. They are, however, not executed on traditional Java virtual machines, but are converted into the custom bytecode and interpreted with the Dalvik virtual machine. The Android SDK supports most of the Java Platform and contains some specific extensions, including telephony functionality.

Android applications consist of four basic component types, activities, services, broadcast receivers, and content providers.

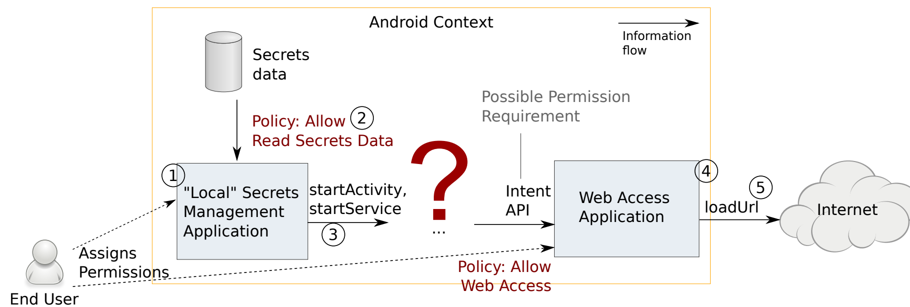

Fig. 1. Example of non-transparent information flow

Activities constitute the presentation layer of an application and allow users to directly interact with the application. Services represent background processes without a user interface. Broadcast receivers subscribe to broadcast messages from applications or the system. Content providers allow an application to share data with other applications. All components follow a component-specific lifecycle that is managed by the OS. For communication between the individual components of applications, inter-process communication (IPC) provides a means to pass messages between different components [3]. One way of IPC are messages that contain meta information and arbitrary data, called intents. Two basic methods of security enforcement are used in Android. Firstly, components run as Linux processes with individual Unix users and thus are separated from each other. This way, a security hole in one application does not affect other applications. A second enforcement mechanism in the Android middleware implements a reference monitor to mediate the access to application components based on permission labels. Similarly, access to security-critical resources, such as Internet, is restricted by permissions.

Description	Exemplary API calls
Invoke an `Activity` by `Intent` (in the foreground)	`Intent intent = new Intent(this, Receiver.class); startActivity(intent);`
Broadcast messages to registered listeners (one-to-many)	`sendBroadcast(intent);` `sendStickyBroadcast(intent);` `sendOrderedBroadcast(intent);`
Communication with `Service` (in the background)	`startService();` `stopService();` `bindService();`

TABLE I
INTER-COMPONENT COMMUNICATION MECHANISMS

Data source	Accessible data
Content Provider	contains passwords, contact list
SMS/MMS	sensitive information
User input	passwords
Files	business documents
Network (HTTP)	protected files
Bluetooth	contacts, files, images
Camera	observation of image data
C2DM	sensitive URI
Location Manager	observation of location data
Device identifiers	personal identification

TABLE II
DATA SOURCES (INCOMING CHANNELS)

III. SOURCES AND SINKS IN ANDROID

Our approach to information-flow analysis is to analyze inter-component flows from information sources, such as contact lists, to channels through which information leaves the device context. Thus, we must identify communication mechanisms between components as well as critical incoming and outgoing channels on the Android platform. The incoming channels are referred to as *data sources*, outgoing channels as *data sinks*. We identified a list of inter-component communication mechanisms, sources and sinks by exploring of the Android application framework and the provided samples. Table I lists the primary communication types on the Android platform. For the sake of brevity, only individual examples of the API calls are given. The origin of the data in an information flow needs to be known to effectively analyze the flows' criticality. Table II provides a list of data sources that allow the flow of information into the device and application context. Enck et al. similarly identified data sources for the placement of security hooks in their dynamic analysis, categorizing sources into sensors, such as location sensors and camera, information databases and device identifiers [1]. In Table III, we list data sinks of Android applications with possible attack scenarios.

IV. INFORMATION FLOW ANALYSIS

We propose to analyze the information flows between the applications to improve the transparency w.r.t. the permission-transitivity problem on the Android platform. Our analysis approach aims to identify information flows between different Android applications/components. To analyze a larger set of applications (as it usually exists on an end user's phone), we did not only utilize the abstract syntax tree (AST) for the entire analysis, but also higher-level information of the software architecture. Through this two-layer approach, we plan to reduce the information input for resource-consuming analyses based on the AST. Our prototype uses two distinct tools to implement the analysis. We employ the Bauhaus tool-suite [4] at the architectural level and the Soot tool [5] for AST-based analyses. The Soot tool provides different types of analyses like context-sensitive pointer analyses implemented in the Paddle extension [6]. However, some of these analyses cannot be used for our approach because of Android's specialized method-starting mechanism for applications. The basic steps our analysis approach are as follows: Using the Resource Flow

Data sink	Attack scenario/attack requirements (exemplary)
Network (WebView)	manipulation of URI / access to URI
SMS/MMS	manipulation of data or number
Bluetooth	influencing the transferred file / proximity, control of receiver, completed pairing
Content Provider	malicious application misleads into writing into content provider / need to specify content provider URI
Files	malicious application misleads into writing into files/ need to specify file name
Google Translate API	manipulation of address resolution / access to OS services
MapView	manipulation of address resolution / access to OS services

TABLE III
DATA SINKS (OUTGOING CHANNELS)

Graph (RFG), which is provided by the Bauhaus tool and represents architectural information of the software, we search for all security-relevant Android API calls. We understand all data sources and sinks as well as Android's IPC mechanisms as security relevant. Having extracted those calls with the help of Bauhaus, we obtain the software architecture of the Android application under investigation. This information is then communicated to the Soot tool, which performs the actual data-flow analyses (backward slicing) on the code level. In the last step, we annotate the software architecture in the RFG with the data flows identified with backward slicing. This is done via "summary edges", a concept used in slicing algorithms [7]. A high-level overview of our analysis technique is given in Figure 2. In the following, we describe the single steps of our analysis method in more detail. Input of our analysis is a set of Android applications in Java bytecode format[1].

Identify components and IPC points at the architectural level: In the first phase of the analysis, we identify the components as well as the associated entry and exit points at the architectural level. The architecture-level analyses are based on the the RFG, which the Bauhaus tool generates from the Java bytecode of the analyzed applications. The RFG is a more abstract representation than a common AST and represents architecturally relevant information of the software (e.g., call graph relations, member access) [4]. It is a hierarchical graph, which consists of typed nodes and edges representing elements like routines, types and components as well as their relations, such as call relations. From the global RFG, we create a reduced information-flow subgraph, containing only the relevant parts of the studied Android components, also called **information-flow view**. The relevant parts are identified through the search for Android framework patterns that are described in Section III. We identify relevant parts through pattern matching and mark the related methods, classes, and calls by adding the nodes and edges to an information-flow view.

[1]An analysis of applications in Android's code format (called "DEX") should also be possible with the help of the ded/dare tool [8]. However, we have not pursued this further and postpone this to future work.

Identify component-level flows by backward slicing: An intra-component data flow analysis is carried out at the AST level to find information flows between entry and exit points in components. The entry and exit points identified in the previous step were used to focus the data-flow analysis and reduce the analysis effort at this level. We developed analysis algorithms for the Soot tool that utilize the known Android framework semantics. For each class of entry and exit points supported by the prototype a corresponding analysis building block has been implemented. The behavior of the Android platform prevents the Soot framework from generating a sufficient call graph for our analyses. One reason is that there is no single entry point to the Android applications, such as `main()`. More importantly, there are several, partly dynamic framework semantics that need to be part of the call graph, such as UI event handlers, but are difficult to be statically analyzed.

To identify the intra-component information flows, we search for all program points that affect a given exit point in a component. Therefore, we chose a static backward slicing technique [7]. If the backward slicing reaches an entry point of the component under investigation, we consider this an information flow for the specific entry and exit points. This information is stored in so-called "summary edges" [7]. Summary edges usually store information about dataflows through called methods (flowing from method entry to exit). We leverage this concept by considering information which flows through a whole Android component.

In addition, we identify the destination (component) of each exit point. If a component's exit point, for instance, calls another activity explicitly, we must extract the parameter of the `startActivity()` call from the AST. This information is needed to calculate the information flows *between* components and applications, respectively (see also the following step).

Consolidate inter-component information flows in the RFG: The component-level flows from the AST-based analysis are now employed to enrich the existing RFG-based information-flow graph as follows. For each information flow that has been identified in the previous analysis step, an intra-component flow edge is added between the entry and exit point and the corresponding nodes are added to the view. If the origin of the current flow is tagged as "data source", an information-flow edge is inserted from the origin to the point of entry inside the current information-flow view. Additionally, for all types of destinations, an edge is inserted between the point of exit and the destination's point of entry.

In a last step, we derive the application-level information flows, i.e., the flows through an entire application, by combining the RFGs of multiple applications. To identify the detected flows between sources and sinks in the application ecosystem we verify them by starting from entry points that enter the application to the exit points that leave the application. This ensures that we have detected a potential critical flow from a sink to source (see Section III). The primary purpose of the information-flow graph is to allow developers and security

High-level algorithm: Information flow analysis

```
Input:   A set of Android applications in Java bytecode.

Step 1:  Search for data sources, sinks as well as IPC calls
         on the software architecture via Bauhaus (RFG).

Step 2a: Do context-sensitive backward slicing with IPC exit points
         as the slicing criteria and store this information as summary edges.

Step 2b: Calculate the target components of IPC calls.

Step 3:  Insert the information determined in 2a) and b) into the RFG and
         calculate information flows on the architectural level.

Result:  Information flows of a set of Android applications.
```

Fig. 2. Approach for analyzing the Android applications

experts to quickly identify flows and determine the risks related to the flows without the need for digging into code details.

V. Case Study

We evaluate our approach by means of a case study of a public transport-related application and thereafter show how the analysis results can be displayed.

Case Study Setup: We chose to analyze two real-world Android applications that are available on Google Play with installations up to 50,000 devices. The first application is called *EfaFahrplan* [9], an interface to a public transport routing Web application, primarily improving the input form to take advantage of the context (current location and time as well as previous searches). *EfaFahrplan* takes parameters such as origin, destination and desired arrival time and sends a query to the routing Web application. Thus, the *EfaFahrplan* application requires unrestricted Web access privileges.

When entering the public transport routing parameters, the user may choose to take the current location as the origin. Since using detailed location data is not strictly necessary for the application's main goal, location queries have been factored out into a separate Android component that is installed as a separate application. As shown in Figure 3, the *EfaQueryLocation* application [10] thus requires location data permissions. With two separate applications, a user may choose whether she would like to grant location access. Still, as shown in the figure, there is an information flow between both applications. This information flow is required to fulfill the intended goals, but it was not explicitly granted at installation time. Although not harmful, this information flow is an example of the missing transparency w.r.t. permission transitivity on the Android platform.

There are several reasons for selecting these two applications. One pragmatic aspect is that one of the authors developed the applications; thus, we had access to the applications' Java bytecode as the basis for our analysis. A more important criterion for choosing this application was that it encompassed different frequently-used Android concepts such as starting activities or binding to services and a multi-component design (see Section II). The two applications also demonstrate that the permission transitivity is a necessary concept, although the missing transparency may cause the concept to be misused.

Applying the Analysis Prototype to EfaFahrplan: We now describe our analysis approach in more detail with the help of this case study. In Figure 3, we can see that our system consists of two applications with three Android components. We can identify the entry and exit points of the components by means of the architecture-level analysis (step 1 in Section IV). Taking a closer look at the `ResultWebView` activity, we obtain `onCreate()` as an entry point, and as an exit point the call to the Android `WebView` UI element, which is at the same time a possible data sink as described in Section III. Thereafter, we interface with the Soot tool to perform the detailed analysis on the AST.

At the AST level, we carry out the Soot analysis with the backward slicing algorithm (step 2 in Section IV), based on the component and IPC point information. For the `ResultWebView` component, we need to verify, for example, whether an intra-component information flow exists between the IPC points `onCreate()` and `WebView`. The backward slice starting point for each component is the exit point of the component, for example, calling `loadUrl()` on the `WebView` component. Beginning here, we look up all variables passed on with the method call and move backward along the statements inside the method `loadResults()` to identify each point that affects the exit point. When the beginning of this method is reached, we have a set of variables that affect the exit point and we evaluate whether any of these are method parameters to do further analysis on affected points, maybe, in other methods of the `ResultWebView` class. In this case, the variable `extras`, is a parameter, so we trace where the current method was called. The method was called by `onCreate()` that is described as a starting point for activities in Section II. Inside `onCreate()`, `loadResults()` is called with the returned value of the inherited method `getIntent()`. This inherited method is another artifact of Android activities and returns the intent with a set of parameters that started the activity. Thus, we identified an intra-component information flow between the entry point `onCreate()` and the exit point `WebView`.

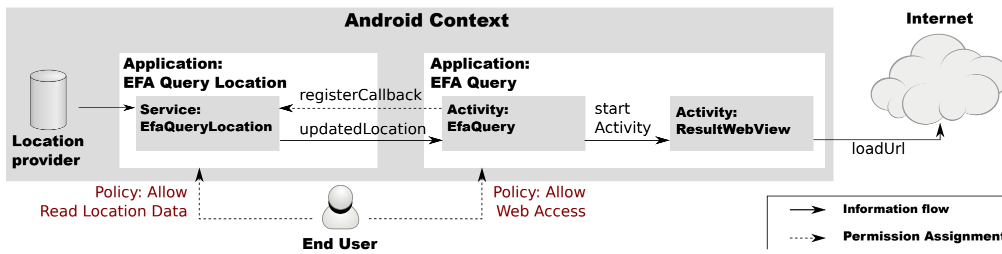

Fig. 3. Case study setup for public transport application

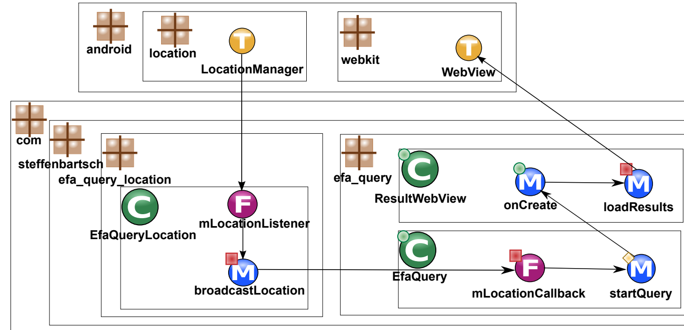

Fig. 4. Generated and simplified information-flow view

The results from the AST-level analysis are passed back to the architectural analysis through an exchange file. In the last step, the information flow data are used to draw appropriate edges in the information-flow view at the architectural level (step 3 in Section IV). The information-flow view of the resulting RFG is shown in Figure 4 as displayed in the Bauhaus Gravis visualization. When comparing the information-flow graph to the case study set up in Figure 3, one can follow the information flow from the location provider source through the three components to the Internet. Thus, the developed method successfully identified the non-transparent information flow. This detailed visualization is helpful for security analysts and developers, who need to find out which architectural elements are related to potentially unexpected information flows.

Visualization for Information Flow Transparency: The developer-oriented visualization in Figure 4 is too detailed to be of use for end users. To provide an adequate level of abstraction, we generate a more abstract visualization from the analysis results, depicted in Figure 5. The goal is to provide insights into the potentially malicious information flows between the applications and critical sources and sinks. We display those information flows that take advantage of the transitivity of trust. First, we show all information flows that start out at a critical source and lead to a critical sink. Additionally, to prevent false negatives, we also display flows to the sink from applications on the path. One option is for end users to employ this visualization on-demand to gain an overview of hidden information flows on their devices. Arguably even more effectively, the visualization might serve as an addition to the existing installation process. In this case, additional information flows that are facilitated by the new application are shown after the user has accepted the permissions that the application requires but before the actual installation. The latter case is what Figure 5 depicts.

Fig. 5. End-user information flow visualization

While currently implemented as a separate application, the information flow transparency view could be integrated into the installer at a later time. Also, the flow transparency application currently reads the information flow data from a file that has been previously generated on a PC. We may port the analysis to the Android platform as part of our future work. Alternatively, the analysis may be conducted by the Android Market owner when the application is uploaded to the market and provided at installation time in addition to the application package. The generated information flow data may additionally be used by security engineers to assess the security of a set of applications. For example, the market supervision could use an appropriate visualization to identify potentially harmful applications. Information security staff at companies might also be interested in analyzing the security of applications on their employees' smartphones.

VI. Related Work

Several works for the static security analysis of software exist, which are discussed by Chess and West [2]. Some of the prototypes for static security analysis have developed into commercial tools such as the Fortify Source Code Analyzer (SCA) [11]. Our approach is complementary to the aforementioned tools as they are designed to detect common low-level security bugs such as SQL injection and Cross-site scripting vulnerabilities. We, however, focus on interactions between applications and specifically consider Android's framework semantics for our analyses. In the latest versions, Fortify SCA supports Android analyses, but these are quite simple. Fortify SCA scans the manifest/configuration files and notes if security-critical permissions are requested and if components are not appropriately secured by permissions. In addition, Fortify rules exist that can track information flow *within* Android components. The interaction between components is not supported at the time of this writing.

The TaintDroid tool implements dynamic monitoring of privacy-relevant flows by modifying the Dalvik VM and the Android kernel [1]. Instead of static analysis before installation, TaintDroid complements our approach by offering analyses at runtime. While TaintDroid aims to minimize the performance overhead, static analyses can also afford to carry out more detailed analyses. Enck et al. reported on their results of statically analyzing widely-used Android applications [8]. They implemented the "ded" tool which translates DEX code to Java class files, and used Soot as a decompiler to finally recover Java source code. Then, they use Fortify SCA to check security rules against the Android apps, for example, to detect phone misuse or hidden eavesdropping functionality. Their approach, however, does not support IPC, which is one of the main concepts of the Android platform. As a consequence, this analysis technique cannot follow information flows between Android components. Chin et al. presented ComDroid, a tool, which can detect vulnerabilities in apps which stem from the faulty usage of IPC, e.g., broadcast messages which are not protected by permissions [12]. Their approach works on the dex code level in contrast to our approach. Although quite effective, they only use simple data- and control flow algorithms rather than more sophisticated analyses as we have done (context-sensitive backward slicing, summary edges as well as architectural analyses). Consequently, ComDroid cannot reveal situations as shown in Fig. 4. Building upon ComDroid and a permission map for the Android framework, Felt et al. provided a tool that can detect overprivileged Android apps. These are apps which request permissions that they do not need [13]. Grace et al. present their Woodpecker tool which can detect capability leaks in Android system apps, e.g., provided by smartphone manufacturers [14]. They also employ a static analysis approach on the dex code (or to be more precise, on the format of the disassembler `baksmali`). However, they currently do not consider the collusion of applications nor an architectural representation of apps and their interactions. To improve security in Google Play, Google has introduced the Bouncer, a dynamic test tool for Android apps. Apps run within Google's Cloud and are checked for suspicious behavior. This mechanism, however, has been successfully tricked, by delaying the actual attack [15].

VII. Conclusion and Outlook

We described an approach to making transparent interactions between applications in dynamic multi-component systems. Focusing on the Android platform, we presented a two-layer approach to the static security analysis of information flows for composite Android applications. On the upper layer, we identified the applications' architecture including entry and exit points as well as Android components. Thereafter, the actual data flow analysis is carried out at the AST level for single components. The results are integrated into the architecture to derive information flows at the architectural layer. There are several directions for further research. First, we aim to support a more complete set of data sources and sinks as well as other concepts of the Android framework such as pending intents and service hooks. We will also analyze larger sets of applications. For example, it would be interesting to investigate (at least) parts of Google Play and develop information-flow policies that the applications should adhere to.

References

[1] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[2] B. Chess and J. West, *Secure Programming with Static Analysis*. Addison-Wesley, 2007.

[3] W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security," *IEEE Security and Privacy*, vol. 7, pp. 50–57, 2009.

[4] K. Sohr and B. Berger, "Idea: Towards architecture-centric security analysis of software," in *Engineering Secure Software and Systems*. Springer-Verlag, 2010.

[5] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java bytecode optimization framework," in *Proceedings of Conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1999, p. 13.

[6] O. Lhoták, "Program analysis using binary decision diagrams," Ph.D. dissertation, School of Comp. Sci., McGill University, Montreal, 2006.

[7] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems, TOPLAS*, vol. 12, no. 1, pp. 26–60, Jan. 1990.

[8] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A Study of Android Application Security," in *Proceedings of the 14th USENIX Security Symposium*, Aug. 2011.

[9] Google Play: EFA-Fahrplan, 2011. [Online]. Available: https://market.android.com/details?id=com.steffenbartsch.efa_query_location

[10] Google Play: EFA-Query-Location, 2013. [Online]. Available: https://market.android.com/details?id=com.steffenbartsch.efa_query_location

[11] Fortify Software, "Fortify Source Code Analyser," 2009, http://www.fortify.com/products.

[12] E. Chin, A. Porter Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proc. of the 9th International Conf. on Mobile Systems, Applications, and Services (MobiSys 2011), Bethesda, USA*. ACM, 2011, pp. 239–252.

[13] A. Porter Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proc. of the 18th ACM Conf. on Computer and Comm. Security, CCS 2011, Chicago, USA*, 2011, pp. 627–638.

[14] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic Detection of Capability Leaks in Stock Android Smartphones," in *Proc. of the 19th Network and Distributed System Security Symposium (NDSS)*. USENIX Assoc., Feb. 2012.

[15] J. Oberheide, "Dissecting the Android Bouncer," 2012. [Online]. Available: http://jon.oberheide.org/files/summercon12-bouncer.pdf