Security Testing of the Communication among Android Applications

Andrea Avancini and Mariano Ceccato
Fondazione Bruno Kessler
Trento, Italy
{anavancini,ceccato}@fbk.eu

***Abstract*—An important reason behind the popularity of smartphones and tablets is the huge amount of available applications to download, to expand functionalities of the devices with brand new features. In fact, official stores provide a plethora of applications developed by third parties, for entertainment and business, most of which for free. However, confidential data (e.g., phone contacts, global GPS position, banking data and emails) could be disclosed by vulnerable applications. Sensitive applications should carefully validate exchanged data to avoid security problems.**

In this paper, we propose a novel testing approach to test communication among applications on mobile devices. We present a test case generation strategy and a testing adequacy criterion for Android applications. Our approach has been assessed on three widely used Android applications.

***Index Terms*—Testing, Security testing, Mobile applications.**

I. Introduction

As illustrated by market studies [1], tablets and smartphones meet bigger and bigger success, replacing desktops and laptops for many daily activities. Android is recognized as the world's leading smartphone platform (59% of the market share in 2012 [2]) with 400 millions of Android devices activated [3] at the beginning of the third quarter of 2012, with a rate of 1 million of new activations per day [4].

Hand-held devices are not only used for leisure, such as real time updates on social networks, reading newspapers, blogs, enjoying radio and TV programs, but also in business to read emails while traveling, to access personal archives and financial information. More and more banks, for example, are providing their own home banking applications that run on smartphones.

Given the popularity of smartphones, malicious applications started to spread, attempting to steal sensitive information and to threat the user privacy [5] (e.g. passwords, GPS locations, contacts, browser history) or trying to directly attack applications that involve paying services [6] or that contain private data [7]. The underlying operating system faces the challenge to keep the system secure and so a very restrictive isolation is imposed among applications. Communication among applications is always mediated and controlled by the operating system, so when accessing sensitive data sources policies are accurately checked and enforced.

Despite all the mechanisms deployed to enforce security, poorly designed applications may still contain defects that may expose vulnerabilities. Inter-application messages are a potential source of attacks to vulnerable code. An aggressor application, in fact, may rely on faulty validation of input data to take control of benign applications. Despite software testing represents a valuable strategy to identify program faults, testing the specificity of mobile software is still preliminary [8].

In this work, we propose a novel approach for testing mobile software which aims to extensively test routines devoted to validate input values coming from inter-application communication messages. In particular, the contributions of this paper are manifold, we define (i) an adequacy criterion for testing data validation routines, (ii) a test case generation strategy and (iii) a procedure to validate if a test case reveals a potential security problem.

The paper is structured as follows. Section II covers the background of Android inter application communication. Section III presents our threat model and our approach to testing mobile applications. Then, Section IV describes our tool implementation and Section V reports about the experimental evaluation. After comparing this work with the state of the art in Section VI, Section VII closes the paper.

II. Background

A. Android Design

Android is a popular Linux-based software stack for smartphones and tablets. It includes the system kernel, device drivers, middleware, libraries and system applications. Android applications are written in Java and compiled in the DEX bytecode format to run in the equipped Dalvik virtual machine. New applications developed by third parties are available on the official Android application store (called *Google Play*) to extend the functionalities of a mobile device.

A fundamental part of an Android application is the *manifest* XML file. This file contains all the informations the framework needs to make the application interact with the other applications installed on the device, such as the list of the application components, its special access permissions and the declaration of services exposed to the other applications. Figure 1 shows a fragment of the manifest file for a puzzle-game application, which contains the definition of an application component (i.e. an *activity*), the definition of a capability exposed by it (in the *intent filter*) and two special permissions.

An *activity* is a single window that interacts with the user via the touchscreen. An application is composed by multiple activities, one for each distinct page the application needs

```
<application android:icon="@drawable/game_logo" android:label="@string/app_name">
<activity android:name=".newGameImportActivity">
    <intent-filter>
            <action android:name="android.intent.action.VIEW"></action>
            <category android:name="android.intent.category.BROWSABLE"></category>
            <data android:scheme="http" android:host="*" android:pathPattern=".*\\.game" />
    </intent-filter>
</activity>
</application>
<uses-permission android:name="android.permission.READ_CONTACTS" />
<uses-permission android:name="android.permission.INTERNET" />
```

Fig. 1. Example of Android manifest file.

to display. The example application in Figure 1 defines the activity *newGameImportActivity*, devoted to import a new puzzle definition file from the Internet. This activity can be started by other applications by using Android inter process communication mechanism.

B. Communication among Applications

Applications are provided by different developers with different levels of trust. The security model adopted by Android is to sand-box and firewall applications. Sand-boxing applications means isolate them from system resources. In order to access system resources, applications have to explicitly request proper permissions, that the user evaluates and authorizes at installation time. The application of Figure 1, for example, intends to access the network and the phone contacts, so *INTERNET* and *READ_CONTACTS* permissions are requested in the manifest. At install-time, the user is asked to review and to decide whether to grant the requested permissions to install the application.

Firewalling applications means separating them from each other. Applications receive a distinct Unix user-id, so they run in their own private space and private memory. Communication among applications is possible only through the mediation of the operating system by the so called Inter-Process Communication mechanism (IPC). The framework is designed such that applications can collaborate and provide services each other. An application, in fact, can delegate specific actions to other applications by exchanging messages called *intents*. Intents contain the description of the operation to be performed by the receiver application. An intent contains:

- **Component name**: The name of the component that should receive the message. The field is optional.
- **Action**: A string containing the action to be performed by the receiver, such as initiating a call, displaying data for the user to edit or opening a file.
- **Category**: A string describing the kind of component that should perform the requested action. For example, an application component is in the category *"browsable"* if it should be invoked by a browser to display the content of a link (e.g. an image).
- **Data**: Information required to complete the action, for example the URI of an image to display, a phone number to call or an address of an email to compose.
- **Extras**: Key-value pairs of additional data.
- **Flags**: Various flags.

Intents can be sent in an *explicit* or an *implicit* way. In *explicit* intents, the sender application specifies the receiver name as part of the intent. Different Android users, however, may have a wide diversity of installed applications, so a specific application may not be available. To cope with the heterogeneity of device configurations, application developers rarely use explicit intents.

Implicit intents just specify the action to be performed, without naming any target component. The system inspects the intent content (limited to action, category and data) to decide the most appropriate destination application(s). To achieve this objective, the system compares the intent content with *intent filters*, data structures defined by those components that can potentially receive intents. The browser, for example, sends implicit intents to ask the platform to render those files that is not able to render. By inspecting intent filters, the framework delivers each request to the most appropriate application, i.e. to the application that declares the corresponding file type in its intent filter.

C. Intent Filters

The manifest file defines what requests (intents) each application component is able to handle, by specifying proper *intent filters*. The framework relies on intent filters to dispatch implicit intents. The intent filter of the activity *newGameImportActivity* (Figure 1), for example, specifies that the activity can handle intents coming from a web browser (*BROWSABLE* category) to display (*VIEW* action) for the user the information contained in a remote file accessible via the *HTTP* protocol, whose name ends with ".game".

In particular, an implicit intent is delivered to a component (e.g. an activity) if the intent message satisfies the following conditions of the intent filter: (1) The action in the intent matches one of the actions in the filter; (2) Every category in the intent matches one of the category in the intent filter; (3) The data in the intent match the data element in the intent filter. A data element is specified in terms of its MIME type (not shown in the example) and in terms of the *scheme*, *host*, *port* and *pattern* of the data URL.

In case an implicit intent satisfies multiple intent filters, the user is prompted the list of compatible applications in order to specify which one to activate.

III. Testing

A. Threat Model

While an implicit intent is delivered to a component only when it satisfies the conditions of an intent filter, explicit intents already contain the name of the destination component, so the framework delivers them immediately without verifying any condition. Thus, on explicit intents, the validation of the content of an intent is completely in charge of the receiving application. There could be a difference, however, between the validation defined in the filter (enforced by the framework on implicit intents) and the validation implemented by the target application (enforced by the application itself on explicit intents). Mismatches may cause messages to be incorrectly handled and may cause errors or application crashes.

Consequences might even be more serious when the target application is granted special permissions to access sensitive phone information. A malicious application may exploit the inadequate validation of a vulnerable application to misuse special permissions. In fact, if not adequately validated, malicious data may reach protected API calls, and the confidentiality of sensitive user data may be threatened.

Data validation must be carefully tested to avoid mismatches between the intended validation (specified in the filter) and the implemented one (specified in the code). A mismatch is found when the application under test receives an intent I such that:

1) I violates the intent filter;
2) The application fails to validate the intent and does not reject it. Thus, I is processed as a valid intent; and
3) While performing the action requested by the intent, the application executes a protected API call that requires a special permission (defined in the manifest).

The objective of IPC testing is to generate test cases that satisfy testing conditions (1), (2) and (3).

B. Filter Violation

Condition (1) requires input data (intents) to violate the intent filter. So, test case generation consists in systematizing input data that do not satisfy the filter as specified in the manifest file. To violate a filter, an intent just needs to violate one among all the conditions defined in it. To maximize the intent coverage, we adopt a criterion based on condition coverage, i.e. we require that each condition in the intent filter is violated by at least one test case.

Before starting the generation of test cases, the manifest file is parsed to extract data about the intent filters defined in it. Then, each intent filter is subject to test case generation. The first test case to be generated is the *prototype*, a test case that does satisfy all the conditions in the filter. To assess the validity of the prototype, we deliver it as an implicit intent and we check if the component under test is selected as one of the potential destination by the Android framework.

New test cases are generated by cloning the prototype and by mutating each clone such that it negates one of the conditions in the filter. The mutation operators are:

- *Change action*: The action of the intent is changed into a new action that is not present in the intent filter;
- *Change category*: The category of the intent is changed into a new category that is not present in the intent filter;
- *Change data*: The data in the intent are changed such that just one filter condition on data is not satisfied. Since data are expressed in the format *scheme://host:port/path*, this means to change any of the composing parts (i.e., MIME type, scheme, host name, port or path). The newly created URL, of course, must be valid and it must refer to an existing resource. To respect this constraint, we generate only URLs for a domain we control, where the new resources can be easily deployed.

After generating all the mutated intents, we verify whether testing condition (1) holds for all of them, i.e. test intents do not satisfy the intent filter. To assess this, we send test intents as implicit intents, without specifying any destination. The Android framework will rely on intent filters to decide the proper destination. We keep only those messages that are not delivered to the application under test, because they violate the intent filter.

C. Dynamic Analysis

Intent messages generated so far need to be sent to the application under test, to verify if they also satisfy testing conditions (2) and (3), i.e. the application fails to reject invalid intent messages and the execution triggers a privileged protected API call. A way to achieve this is by comparing the behavior of the application under test with dynamic analysis. Dynamic analysis is able to detect whenever an invalid intent is not only accepted as valid but also, as consequence of its execution, whenever a privileged API call is triggered.

Dynamic analysis is performed by cloning the application under test P into application P'. Then, the privileges of the clone are reduced by removing all the permissions from its manifest. P and P' are instrumented to record their execution in terms of method executions, exceptions and errors. We log the execution of methods, exception handling blocks (the Java *catch* statement) and exceptions that are not handled (application crashes). Logs record an execution trace which is direct manifestation of an incoming intent.

Each test intent is sent (as explicit intent) directly to P and to P' and execution traces are compared. There could be three cases:

1) *No error*: In case of equal traces with no errors, it means that the execution of the given intent with and without permissions does not pose any difference. So the intent is either discarded by both P and P', or it is not discarded but no protected API is called. Thus, according to our threat model, no defect is found in the intent validation routine;
2) *Same errors*: In case the two traces present the same error, with and without permissions, it means that the

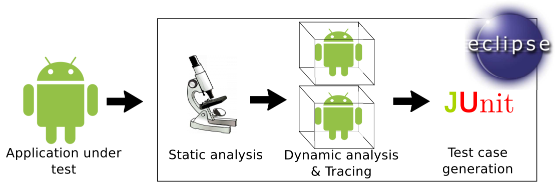

Fig. 2. Overview of the tool support.

malformed intent makes them fail in the same way, but without triggering any privileged API call (otherwise there would have been a difference in the traced errors). So, even if the intent is not rejected, no sensitive actions have been performed.

3) *Different errors*: Conversely, if the execution traces are different, it means that in one case there has been an error not occurred in the other case. As the only difference between P and P' is in the permissions, a diversity in the traces means that the intent made the application P' trigger a privileged API call that caused an error due to insufficient permissions. In this case, the test could possibly expose a potential security vulnerability, because the invalid intent is not rejected and forces the application to perform a privileged action.

The latter case needs to be fixed by corrective maintenance.

IV. Tool Support

The approach presented in the previous sections has been implemented in a tool prototype. The prototype consists of an Eclipse plug-in, integrated with the Android Development Tools (ADT). It relies on static and dynamic analysis to generate JUnit test cases in an automatic way. The overview of the testing framework is shown in Figure 2.

A. Static Analysis

The manifest file of the application under test is copied into the *input* folder of a *test runner* application by the Eclipse plug-in. The test runner loads the manifest file, parses it and populate a database with data to generate invalid intents that satisfy the adequacy criterion.

B. Dynamic Analysis

The target application P is cloned into application P' with reduced permissions and both the applications are instrumented to log their execution during testing.

The instrumentation records the execution of the application under test, logging method calls, method executions and exception handling *catch* blocks. The instrumentation also logs all the errors that are not handled and that make the application crash. Instrumentation is implemented by an aspect written in AspectJ, the aspect-oriented extension of Java, and applied at source code level. Traces are written in log files that are extracted and inspected by the Eclipse plug-in.

C. Intent Generation

Intent generation is performed automatically by the test runner application, which is deployed onto the Android emulator together with P and P'. The manifest database is accessed and, for each intent filter, a valid intent is created, i.e. the prototype. The test runner verifies that prototypes are valid intents by using a query API in the Android libraries. Only if P and P' are among the candidate destinations, the prototype is classified as valid.

Each prototype is then mutated by the test runner (as described in Section III) to generate a test suite of invalid intents. To satisfy the adequacy criterion, an intent is constructed by mutating one attribute of the prototype and this procedure is repeated until each attribute is negated at least once. Mutation operators described earlier reuse values from a list of possible actions, categories, protocols, hosts and MIME types, while file names are built from scratch by using the Xeger[1] library, which generates strings starting from a regular expression (i.e. the file pattern specified in the intent filter). If needed, new files are created on the secure digital (sd) card of the Android emulator with a random content, or uploaded to a local web server. At the end of the testing, all these files are deleted.

D. Test Case Generation

When the execution traces of applications P and P' are available, they are compared by the Eclipse plug-in. Any difference in terms of different errors or same error in a different position is the manifestation of a mismatch in the behavior of the application with and without permission. In this case, a candidate defect that could lead to a security issue is found and a new JUnit test case is automatically generated to document this problem. This JUnit test case consists of a preparation phase (*setUp* method) that fills the intent content and any file is required to be accessed, the dispatching of the intent and the monitoring of the application execution.

As testing condition (1) is satisfied by construction, the new JUnit test case asserts conditions (2) and (3). The (explicit) intent is sent to the application under test and its execution is monitored. An AspectJ aspect traces the executed methods and compares them with the failing trace that has been observed during dynamic analysis. The test fails if the execution flow triggered by the current test still reaches a permission-privileged instruction.

Test cases generated by the tool prototype are intended to help developers in fixing security bugs. A test case reports all the relevant information to understand and replicate the suspect fault:

- Values to be used to construct the invalid intent. The intent fields are the *Action* requested to the receiver, the *Category* of the intent and the *Data* to be loaded;
- An annotation of which of these fields violates the intent filter;
- If required, the file to be deployed either in the Android file system or in the local web server; and

[1]http://code.google.com/p/xeger/

TABLE I
SUBJECT ANDROID APPLICATIONS

Subject	Version	Popularity	Classes	LOC
AnkiDroid	2.0beta16	100.000+	203	25884
Jamendo	1.0.3beta	100.000+	132	4384
OpenSudoku	1.1.5	1.000.000+	63	3749

- The list of methods in the execution trace that would make the application execute a privilege-protected statement when the invalid intent is received.

The developer can use the test case to debug the application, setting break-points and understanding how and why this invalid intent is not rejected by the application. When the problem in the data validation routine has been located and fixed, a new run of the test case would confirm that the maintenance task is completed.

V. EMPIRICAL EVALUATION

The goal of our empirical evaluation is to assess the effectiveness of our testing approach in terms of discovering inadequate validation of invalid intent messages that cause calls to access-restricted APIs. The evaluation has been performed by using the prototype tool described in Section IV on three subject applications. Then, manual inspection has been required to confirm the reported defects.

A. Test Subjects

We selected three real-world, open source Android applications, also available in the official Android application store, called Google Play. The first, AnkiDroid[2] 2.0beta16 is the mobile version of Anki, a spaced repetition flashcard program which intends to help the user in remembering things easily. The second, Jamendo[3] 1.0.3beta, is the Android porting of the popular portal Jamendo, a music player, music catalog and a website supporting a community of music authors for free and legal music downloads under Creative Commons licenses. The third application, OpenSudoku[4] 1.1.5, is an open source Sudoku game that also offers the possibility to download extra games and to create new games from scratch.

Table I lists, for each subject application, its version, the popularity (expressed in the number of total downloads from Google Play), the number of Java classes and the total lines of code (LOC).

B. Testing Results

Experimental results are reported in Table V-B. For each application under test (first column), the second column reports the number of activities that are declared in the manifest file, but only those with a specified intent filter (third column) are subject to testing. The central part of Table V-B (columns 4-6) deals with test case generation. Column *Prototypes* reports the number of valid intents exposed to mutation to generate those listed in column *Test Intents*. Among them, *Invalid Intents* are those that violate intent filters and that have been concretely used in the testing procedure. The last part of the Table (columns 7-8) presents the results of testing. Column *Failing Tests* shows how many test cases are classified as non-pass by the tool so they should reveal a problem in message validation. The same fault could be revealed by different tests, so a unique sink, the point in the code where the fault manifests, can be identified. Column *Unique Sinks* reports how many distinct problems could be observed after manual inspection.

AnkiDroid declares 16 activities in its manifest file, but only 2 define filters and can receive intents. The tool generated 13 prototypes and, after mutation, 61 new distinct test intents. After filtering, 50 invalid intents were still in the test suite and have been used for the experiment. The tool generated 5 test cases out of 50 invalid intents, where receiving an invalid intent caused an exception in P' due to restricted permissions that was not observed in P, so the plug-in generated 5 new JUnit test cases. Manual inspection revealed that all these violations were due to the *CardEditor* activity that raised the *SQLiteCantOpenDatabaseException*. An invalid intent was processed as valid, so the application attempted to update a card collection by accessing a database.

By parsing the manifest file of the second application, Jamendo, we found 14 activities and all of them defined intent filters. The test runner produced 46 prototypes and, after mutation and filtering, 150 violating intents. The majority of the invalid intents has been generated for the activity *IntentDistributorActivity*, because it defines a quite complex intent filter, as it acts as a sort of internal proxy for other activities. We logged 51 differences among the execution traces of the two versions of Jamendo caused by different permissions. Whenever *IntentDistributorActivity* receives an invalid intent that contains a media database entry that satisfies the Jamendo format, the validation of the intent content results incomplete and the application, as requested, tries to connect the Jamendo server causing an *UnknownHostException* due to the missing grant of *INTERNET* permission. Also in this case, all the failing test cases reveal the same sink located in class *Caller*, called on connections to remote hosts.

The third application under test, OpenSudoku, defines 10 activities, while 5 of them implement an intent filter. 48 intents have been generated starting from 15 prototypes, but just 44 were actually invalid intents. The tool identified 11 cases of mismatch between executions traces, that lead to two different sinks. The recorded exceptions are thrown after calling activity *FileImportActivity*, when the received intent requests to import a remote game-definition file. Also in this case, the validation of the invalid intent is missing and the execution is carried forward instead of being blocked. Since the permission of accessing the network is not granted to the cloned application, the import task fails, triggering a *Socket* exception. By manually inspecting the code, we discovered that OpenSudoku supports two game definition formats on import, corresponding to the file suffixes *.sdm* and *.opensudoku*. Import is then delegated to two different

[2]play.google.com/store/apps/details\?id=com.ichi2.anki

[3]play.google.com/store/apps/details\?id=com.teleca.jamendo

[4]play.google.com/store/apps/details\?id=cz.romario.opensudoku

TABLE II
EMPIRICAL DATA ON THE CASE STUDIES.

Subject	Total Activities	Activities with Intent Filter	Prototypes	Test Intents	Invalid Intents	Failing Tests	Distinct Sinks
AnkiDroid	16	2	13	61	50	5	1
Jamendo	14	14	46	188	150	51	1
OpenSudoku	10	5	15	48	44	11	2

classes according to the file type. When receiving an invalid intent from the tool, activity *FileImportActivity* forwards it to another activity, without implementing any validation policy. The receiving activity delegates import of the remote file to the more appropriate class, according to the type of the file. At this point, either *SdmImportTask* or *OpenSudokuImportTask* class is executed and, in the attempt to access the network to download the game-definition file, they raise an exception due to insufficient privileges.

testing condition for the automatic generation of test cases.

VI. Related Works

Potential threats to the security of mobile applications due to inter-process communication have been studied with respect to information flow [9], [10], [11]. Information flow in mobile applications is analyzed either statically [9] or dynamically [10], to detect disclosure of sensible information. Tainted sources are system calls that access private data (e.g., global position, contacts and calendar entries), while sinks are all the possible ways that make data leave the system (e.g., network transmissions). An issue is detected when privileged information could potentially leave the application through one of the sinks.

Another relevant threat for information flow is represented by permission re-delegation [11], i.e. an application with special permissions that exposes a service that does not require the same permissions to be consumed. As a consequence, the service could be used by a second application without permissions to ask the former to act on its behalf and take privileged actions. A vulnerability is detected whenever there exists a path from a public entry point to a restricted API call.

ComDroid [12] is a tool intended to identify when inter-application messaging is implemented in insecure way, either because internal services are exposed to external calls or because of messages with missing or weak permission requirements.

Differently to us, the common objective of all these approaches is to list candidate vulnerable points, but none of them provide test cases to support developers during corrective maintenance. Given the fast time-to-market model of mobile applications, it is important to provide few runnable examples of problem manifestations, instead of large list of potential problematic cases that can not be executed, to support with some level of automation the corrective maintenance to fix vulnerabilities.

The only work which analyzes inter-application messaging with the objective of generating test cases is by Maji et al. [13], where the communication is tested to spot application (or system) crashes. The JarJarBinks tool has been implemented to study when invalid intents (i.e. that violate intent filters) make applications crash. Test cases, however, reveal generic application crashes rather than focusing on security.

Automatic testing of peculiarities of Android applications has been addressed from the point of view of the graphical user interface [14], [15], [16], to detect events and event sequences that make the application crash. Hu et al. [14] presented an approach for testing Android applications' GUI. Random graphical events are generated and patterns are used to detect bugs in the system log, such as application crashes, type exceptions and violations in the activity state machine.

Amalfitano et al. developed *AndroidRipper* [15] and *A2T2* [16] (Android Automatic Testing Tool) to test Android GUI. These tools dynamically analyze the applications to get a list of fireable events in the GUI widgets, they then generate sequences of graphical events and sensor events. Code is instrumented to record crashes and to eventually translate event sequences into JUnit test cases.

To the best of our knowledge, the only work on security testing of Android applications is by Mahmood et al. [8]. The authors resort on random graphical events and random input values to generate test cases. Tests, however, are not directly revealing vulnerabilities that could be potentially exploited by attackers, but their objective is just to spot crashes due to communication errors and violations of access permissions. Our approach, conversely, generates test cases that expose issues due to inadequate input validation, that could threat the confidentiality of user data.

VII. Conclusion

Applications for mobile devices handle sensitive data, so they need careful validation to avoid security problems. In this paper, we present a novel approach to test Android applications. Automatic test case generation is proposed to spot mismatches between the intended behavior declared by an application and the observed functionalities implemented in its code. Our approach managed to detect mismatches in three real world and largely used Android applications and to automatically generate JUnit test cases to reproduce potential bugs.

Currently, our approach focuses on inter-application communication of those messages that are directed to activities. As future work, we plan to broaden the approach to services and broadcast receivers, other application's components that can receive intents. We also aim to extend the threat model and to test new types of faults. Moreover, we intend to conduct a more extensive assessment on all the major applications available in the official application store.

References

[1] E. Kahn, "Forrester research ereader forecast, 2010 to 2015 (us)," Forrester research, Tech. Rep., 2010.

[2] "Android device database: Activations by vendor, handset model and region q1 2012," Signals and Systems Telecom, Tech. Rep., 2012.

[3] H. Barra, "Keynote speech," in *Google I/O Conference*, 2012.

[4] M. Burns, "There are now 1.3 million android device activations per day," in *http://techcrunch.com/2012/09/05/eric-schmidt-there-are-now-1-3-million-android-device-activations-per-day/*, 2012.

[5] Juniper Network, "2011 mobile threats report," http://www.juniper.net/us/en/security/, Tech. Rep., 2012.

[6] D. Shetty, "Demystifying the android malware," Exploit database. http://www.exploit-db.com/, Tech. Rep., 2011.

[7] G. Eisenhaur, M. N. Gagnon, T. Demir, and N. Daswani, "Mobile malware madness, and how to cap the mad hatters: A preliminary look at mitigating mobile malware," in *Black Hat 2011 conference*, 2011.

[8] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou, "A whitebox approach for automated security testing of android applications on the cloud," in *Proceedings of the 7th International Workshop on Automation of Software Test (AST)*, 2012.

[9] C. Mann and A. Starostin, "A framework for static detection of privacy leaks in android applications," in *27th Symposium on Applied Computing (SAC): Computer Security Track*, 2012.

[10] 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.

[11] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *20th Usenix Security Symposium*, 2011.

[12] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*, ser. MobiSys '11. New York, NY, USA: ACM, 2011, pp. 239–252. [Online]. Available: http://doi.acm.org/10.1145/1999995.2000018

[13] A. Maji, F. Arshad, S. Bagchi, and J. Rellermeyer, "An empirical study of the robustness of inter-component communication in android," in *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, june 2012, pp. 1 –12.

[14] C. Hu and I. Neamtiu, "Automating gui testing for android applications," in *Proceedings of the 6th International Workshop on Automation of Software Test*, ser. AST '11. New York, NY, USA: ACM, 2011, pp. 77–83. [Online]. Available: http://doi.acm.org/10.1145/1982595.1982612

[15] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using GUI ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 258–261. [Online]. Available: http://doi.acm.org/10.1145/2351676.2351717

[16] D. Amalfitano, A. Fasolino, and P. Tramontana, "A gui crawling-based technique for android mobile application testing," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, march 2011, pp. 252 –261.