An Automatic Security Testing approach of Android Applications

Stassia R. Zafimiharisoa
LIMOS - UMR CNRS 6158
Université Blaise Pascal, France
email: s.zafimiharisoa@openium.fr

Sébastien Salva
LIMOS - UMR CNRS 6158
Université d'Auvergne, France
email: sebastien.salva@udamail.fr

Patrice laurençot
LIMOS - UMR CNRS 6158
Université Blaise Pascal, France
email: laurencot@isima.fr

***Abstract*—In this paper, we propose a security testing approach which aims to check whether Android applications are not vulnerable to malicious *intents*. An intent is an IPC (Inter-Process Communication) mechanism which is used to compose Android components together to form a whole application. From Manifest files, which provide information about Android applications, and based on the vulnerabilities expressed within test patterns, we automatically generate JUnit test cases that can detect intent-based vulnerabilities. Using formal methods, executable security tests are then automatically generated from any Android applications.**

***Keywords*-security testing, Android applications, model-based testing.**

I. INTRODUCTION

As mobile usage grows, so should security: this sentence summarises well the conclusions of several recent reports [1] about mobile application security with platforms such as iOS, Android or Windows Phone. These reports also show that an alarming amount of malicious software is currently available. On the other side, end-users wish using trustworthy mobile applications; so, more and more developers have in mind that security must not be left aside. Nevertheless, eliminating security vulnerabilities in mobile applications is not so obvious since these ones depend on different concepts such as composition of software components, which have not been completely covered by security studies.

This paper proposes a work in progress about an automatic security testing method for Android applications. Since mobile application security is a tremendous research field, we focus here on the composition of Android components: most of mobile applications gather a set of components composed together statically or dynamically. With Android, these components are glued by means of the concept of *intent* which is an IPC mechanism. In reference to the Android documentation [2], an intent is an abstract description of an operation to be performed. Basically, an intent is a data structure holding an abstract description of an action to be executed by another component. This one is generally used to call or launch another component, e.g., an activity (a component which represents a single screen), or a service (component which can be executed in background). For example, an intent is used when a user wishes to search for a contact for later sending an email.

The use of intents introduces vulnerability issues (availability, integrity issues, etc.) when intents are composed of malicious data. The security testing method developed in this paper aims at detecting whether components are vulnerable to these malicious intents. The first step of this method is the generation of formal models ioSTS (input output Symbolic Transition Systems [3]) from Android applications, and more precisely from *Manifest* files. These Manifest files which can be found in Android application packages (apk), contain the list of Android components, permissions of these components and a list of intents that can be performed. More precisely, with introspection of the different compiled components, we derive an incomplete class diagram which will be used to refine and reduce the test case generation. We also parse Manifest files to derive one ioSTS for each component that describes the intent communication. Vulnerabilities (that would be tested) are also described formally with ioSTS called *test patterns*. We give an example of test pattern for availability testing in this short paper. However, our method can be also used to test other security concepts such as integrity or authorisation. Then, our method constructs test cases by combining the component models with test patterns. The resulting ioSTS test cases are finally translated into JUnit test cases to be later executed with classical development tools.

The paper is structured as follows: Section II presents some works dealing with Android security and security testing. The methodology is described in Section III. We conclude in Section IV.

II. RELATED WORK

Several works, dealing with Android security, have been proposed recently. Some works focused on the definition of a more secure Android framework. For instance, different actions were monitored to check the system integrity in [4]. These approaches offer a different point of view, in comparison to the work tackled in this paper which aims at detecting applications vulnerabilities, since they target the discovery of framework vulnerabilities. Analysis of IPC were studied by E. Chin et al. [5]. They described the permission system vulnerabilities that applications may exploit to perform unauthorised actions. We have exploited the described vulnerabilities to model test patterns which

 ISBN: 978-1-61208-230-1

can be used to generate test cases. We have also completed this vulnerability list by means of the recommendations and vulnerabilities referenced by the OWASP (Open Web Application Security Project) communities [6]. At the moment, only Web applications are considered, but this gives some indications for mobile applications too. Then, our approach do not require Android framework extension and deal with a large coverage of Android applications vulnerabilities.

Security testing, based upon formal models, has been studied in several works [7]. Mouelhi et al. propose to produce test cases from security policies described with logic based languages(OrBAC). This ones permits to decribe specific properties such as access control [8].In these two previous works, test cases are generated from specification and invariants or rules describing security policies. Starting with a similar approach, we also use formal models to describe Android applications and vulnerabilities, and we propose to push one step beyond in the automatic generation of partial models to produce tests.

In [9], a threat model-based security testing approach is presented. This method produces security test cases from threat trees and transforms them into executable tests. Using trees is intuitive for industry but the use of formal languages offer several other advantages such as the description of the testing coverage.

III. Security testing methodology

In this section, we present an automatic security testing approach which can be used to detect vulnerability issues based on the intent mechanism. The different steps of the method are illustrated in Figure 1. Initially, several models are extracted from an Android application: from the compiled classes, we extract a partial class diagram by introspection. This one lists the components and gives the associations between classes. We also produce one ioSTS per component. These ioSTSs will be combined with ioSTS test patterns, which describe vulnerabilities, to produce test cases. We obtain symbolic test cases which need to be concretised with values. Finally, these latter are translated into JUnit test cases to be executed with classical development tools. These steps are more detailed in the remainder of the paper.

A. Model generation

1) Android components and interactions: Android applications are usually constructed over a set of components. A component belongs to one of the four basic components types that are *activities* (user interfaces), *services* (background processing), *content providers* (database management) and *broadcast receivers* (broadcast message handling). The communication between these components is performed by intents. An intent is a kind of bundle of information which gathers: the action which has to be performed, data specifies the type of data to operate on or the MIME

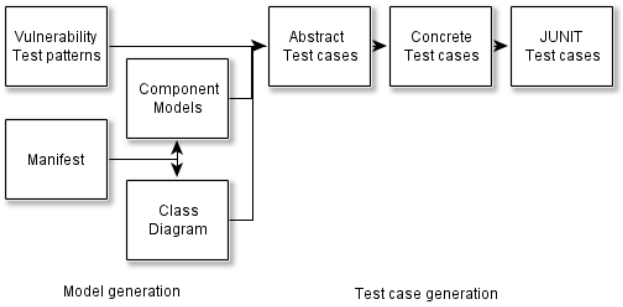

Figure 1. Test case generation.

(Multipurpose Internet Mail Extension) type, a category and eventually some extras which represent variable affectations which are required for the action [2]. Intents are divided into two groups: explicit intents where the target component is designated and must be launched and implicit intents (the most generally ones) where a component of another application is going to be used. The mapping of an intent to a component is expressed in Manifest files with *intent filters*. The latter are given for each component. Android Manifest file is mandatory in a project and it specifies system information about the application. In this paper, Manifest files are used to extract the list of Android components deployed in the application and their intent filters.

2) Component partial model extraction: A simplified class diagram, giving the Android components of the application, is initially computed. The relationships between components are established by applying reverse engineering based on introspection in Java. This step aims to later reduce the test case generation. For instance, we only produce test cases to check data integrity on components which have access to databases or to content-provider components only. Our approach combines introspection with the detection of specific Android component methods defined in [2], which give details about intents, and the access to data.

Then, we generate one ioSTS for each component to describe the intents that it can accept. An ioSTS is a kind of automaton model which is extended with a set of variables, with transition guards and assignments, giving the possibilities to model the system state and constraints on actions. Its complete definition can be found in [3]. An ioSTS is defined by the tuple $< L, l_0, V, V_0, I, \Lambda, \rightarrow >$, where: L is the finite set of locations, with l_0 the initial one, V is the finite set of internal variables, while I is the finite set of interaction ones. Λ is the finite set of symbols and $\rightarrow$ is the finite transition set.

IoSTS are generated with Algorithm 1. It constructs one partial ioSTS specification per component listed in the Manifest file. It is also based on the recommendations provided in [2] to describe the different responses that can be observed after the receipt of an intent by a component. In this paper,

 ISBN: 978-1-61208-230-1

we focus on activity and service components only. Intents are expressed with the output action *!intent* composed of specific variables for the action, the data, the category, the type and the extras. IoSTS are constructed according to the intent action type. The set denoted A_o is composed of the list of Actions that requires response message after the intent, e.g.,the action PICK. A_q is the set composed of the list of Actions that do not need response, e.g.,the action VIEW.

An minimalist and straightforward example of specification is given below. It illustrates one intent composed of the action *VIEW* that is called to display information about the first person in the contact list of the mobile phone.

$$l_0 \xrightarrow[action=ACTION\_VIEW \wedge data=../contact/people/1]{?intent(action,data)}$$
$$l_1 \xrightarrow{!display(ActivityA),[A.resp=isNull]} l_0.$$

Algorithm 1: Partial Specification set Generation

input : Manifest file MF
output: An ioSTS set $\{S_{ct} \mid ct \text{ is a component}\}$
foreach *component* ct *in* MF **do**
  foreach *IntentFilter* $it(a,d,c,v)$ *of* ct *in* MF **do**
    if $ct.type = activity$ **then**
      if $a \in A_o$ **then**
        $l_0 \xrightarrow{?intent(a,d,c,v),[a.in(A_o)]}_{S_{ct}} (l_{it,1})$
        $\xrightarrow{!display(ActivityA),[A.resp \neq Null]}_{S_{ct}} l_0$
      else if $a \in A_q$ **then**
        $l_0 \xrightarrow{?intent(a,d,c,v),[a.in(A_q)]}_{S_{ct}} (l_{it,1})$
        $\xrightarrow{!display(ActivityA),[A.resp=Null]}_{S_{ct}} l_0$
      else
        $l_0 \xrightarrow{?intent(a,d,c,v),[a.in(A_c)]}_{S_{ct}} (l_{it,1})$
        $\xrightarrow{!display(ActivityA)}_{S_{ct}} l_0$
    if $ct.type = service$ **then**
      if $a \in A_o$ **then**
        $l_0 \xrightarrow{?intent(a,d,c,v),[a.in(A_o)]}_{S_{ct}} (l_{it,1})$
        $\xrightarrow{!running(ServiceS),[S.resp \neq Null]}_{S_{ct}} l_0$
      else
        $l_0 \xrightarrow{?intent(a,d,c,v)}_{S_{ct}} (l_{it,1})$
        $\xrightarrow{!running(ServiceS)}_{S_{ct}} l_0$

B. Vulnerability modelling with ioSTS test patterns

We propose to define security vulnerabilities of Android components with ioSTSs in order to combine them later with ioSTS specifications. Our method can take different security concepts, e.g., availability, integrity or authorization in condition that they could be modelled with ioSTSs. One vulnerability is expressed by a tree whose final locations are labelled by a verdict in $\{pass, fail, inconclusive\}$: branches beginning from the initial location and ended by $fail$ express functional behaviours composed of malicious intents and responses which show the presence of the vulnerability. Branches ended by $inconclusive$ express functional behaviours which do not help to conclude on the presence or the absence of the vulnerability. Branches ended by $pass$ express functional behaviours which show the absence of the vulnerability.

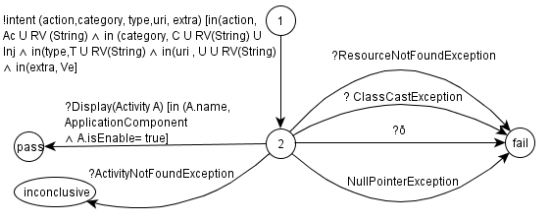

Figure 2. Test pattern for availability

We only present a test pattern example to check availability in this paper. Availability of Android components means that they must respond despite the receipt of malicious intents. An availability test pattern for Activity components is shown in Figure 2. The other security properties are composed with more states. Availability issues are detected when exceptions such as *ClassCastException, NullPointerException, RessourceNotFoundException, etc.* are received [2]. So, the test pattern expresses that when one of these exceptions is observed, the fail location is reached. For instance, these exceptions can be received when there is no input validation of the URI path. System exceptions e.g., *ActivityNotFoundException, SecurityException, etc.* mean that the intent has been blocked directly by the Android system. This is expressed by transitions leading to the inconclusive location. Variables of the intent action take values in specific sets with the term *in*. These sets are required to target the test with specific values and above all to reduce the number of test cases. The set A_c is composed of the list of Actions found in the Android documentation [2]. In the same way, C is a specific set of categories, T a set of types, U a set of URI. Ve is a set of extras i.e., tuples (key, values) where the keys are extracted from the application package (apk) and values are chosen randomly. RV stands for a set gathering values known for relieving bugs and random values. For instance, for the type String, $RV(String)$ holds values such as "", "$", "&", and random values. Inj gathers String values equal to XML and SQL injections.

C. Test case Generation

Test cases are generated by composing test patterns, modelling vulnerabilities with component specifications. As defined, a test pattern expresses one vulnerability in general

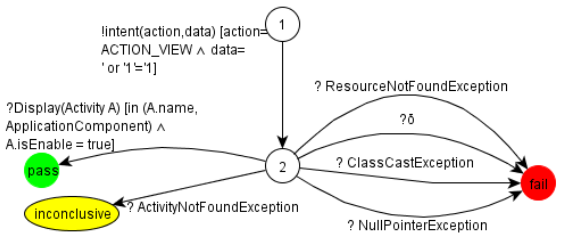

Figure 3. A test case example.

terms. This combination yields symbolic test cases which are specialised to test one vulnerability by means of the functional behaviours given in the specification. The combination of one test pattern TP with one specification S is performed by two steps: synchronous product of the specification with the test pattern to extract from the latter, the functional behaviours expressed in the specification, completion of this product with incorrect and inconclusive behaviours given in the test pattern by transition leading to fail or inconclusive respectively.

1) Test case concretisation: The resulting test cases are still composed of variables. To execute them, variables are replaced with values by using a pairwise technique [10] on the value sets provided by the test patterns (RV, Ac, etc.) instead of using a cartesian product. This technique helps to reduce the coverage of the variable domain by constructing discrete combinations for pair of parameters only. A final test case is shown in Figure 3. It illustrates a malicious intent transition composed of the classical SQL injection *' or '1'='1*.

2) Automatic unit test generation: IoSTS test cases are translated into JUnit test cases to be executed with Java development tools. This is done with an algorithm that can be summarised by: each ioSTS transition starting from the initial location corresponds to the launch of an intent on a component with its parameters defined in the ioSTS test case. Then, the following transitions that correspond to both observations and verdicts are translated into JUnit assertions. Since inconclusive verdict are not allowed with JUnit, it will be assigned to pass verdict and will be identified by specific annotations. The following example illustrates a JUnit test case.

```
public void testAvailability(){ ... setIntent(
    ACTION_VIEW, ' or
'1'='1}; try{ mActivity=getActivity();
assertTrue(currentView,isNotNull());
assertTrue(activityResult,isNull());}
catch (ClassCastException c) {fail(c.message);}
catch (ActivityNotFoundException a)
{assertTrue(a.message,true);} ... }
```

IV. Conclusion and Future Work

We have introduced a work in progress about Android security testing based on the notion of intent which can be considered as a glue of the components participating in applications. The intent concept offers some advantages and flexibility but is also a weak spot in security since attacks can be send to components. Our method generates models from Android Manifest files and constructs test cases from these models and vulnerabilities models. The originality of the method is to produce formal models from Android documentations and files extracted from Android applications automatically. The use of these formal models will be useful to express without ambiguity the coverage of the tests. The next step is to list all the possible intent-based vulnerabilities (we have collected five vulnerabilities at the moment) and to develop the corresponding testing tool to perform experiments on real applications.

References

[1] *IT Business: Android Security*, (June , 2012). [Online]. Available: http://www.itbusinessedge.com/cm/blogs/weinschenk/google-must-deal-with-android-security-problems-quickly/?cs=49291

[2] "Android developer page," (May 1, 2012). [Online]. Available: http://developer.android.com/index.html

[3] L. Frantzen, J. Tretmans, and T. Willemse, "Test Generation Based on Symbolic Specifications," in *FATES 2004*, ser. Lecture Notes in Computer Science, J. Grabowski and B. Nielsen, Eds., no. 3395. Springer, 2005, pp. 1–15.

[4] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically rich application-centric security in android," in *Proceedings of the 2009 Annual Computer Security Applications Conference*, ser. ACSAC '09, 2009, pp. 340–349.

[5] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proc. of the 9th International Conference on Mobile Systems, Applications, and Services*, 2011, pp. 239–252.

[6] OWASP, "Owasp testing guide v3.0 project," 2003, (accessed May 1, 2012). [Online]. Available: https://www.owasp.org/index.php/Category:OWASP_Testing_Project#OWASP_Testing_Guide_v3

[7] H. Marchand, J. Dubreil, and T. Jéron, "Automatic Testing of Access Control for Security Properties," in *TESTCOM/FATES 2009*, Nov. 2009.

[8] T. Mouelhi, F. Fleurey, B. Baudry, and Y. Traon, "A model-based framework for security policy specification, deployment and testing," in *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, 2008, pp. 537–552.

[9] K. H. S. K. Aaron Marback, Hyunsook Do and D. Xu, "A threat model-based approach to security testing," in *Softw. Pract. Exper*, 2012.

[10] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn, "Constructing test suites for interaction testing," in *Proc. of the 25th ICSE*, 2003, pp. 38–48.

 ISBN: 978-1-61208-230-1