Data Vulnerability Detection by Security Testing for Android Applications

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

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

***Abstract*—The Android intent messaging is a mechanism that ties components together to build Mobile applications. Intents are kinds of messages composed of actions and data, sent by a component to another component to perform several operations, e.g., launching a user interface. The intent mechanism eases the writing of Mobile applications, but it might also be used as an entry point for security attacks. The latter can be easily sent with intents to components, that can indirectly forward attacks to other components and so on. In this context, this paper proposes a Model-based security testing approach to attempt to detect data vulnerabilities in Android applications. In other words, this approach generates test cases to check whether components are vulnerable to attacks, sent through intents, that expose personal data. Our method takes Android applications and intent-based vulnerabilities formally expressed with models called vulnerability patterns. Then, and this is the originality of our approach, partial specifications are automatically generated from configuration files and component codes. Test cases are then automatically generated from vulnerability patterns and the previous specifications. A tool, called APSET, is presented and evaluated with experimentations on some Android applications.**

***Keywords*-Security testing, Android applications, Model-based testing, Mobile device security**

I. Introduction

Flaws and security vulnerabilities are common issues in any complex software system. Mobile device operating systems and applications are no exceptions. Many security reports and some recent papers [1], [2] show the presence of several security flaws and proposed some solutions to correct them. An important flaw concerns the intent mechanism of Android. Android applications consist of components that are joined together: user interfaces and background processing are coded with *Activities* and *Services*, instantiated by the Android operating system. Data can be stored in a device by various options, e.g., in raw files or SQLite databases. The *Content-Provider* component represents a more elegant and secure solution which makes data available to applications. The composition mechanism is performed with intents, which is an IPC (Inter Process Communication) mechanism, used to call or launch another component. The Content-Provider access can be restricted with permissions. Without permission (the default mode), data cannot be directly read by external applications. Considering this case, data can still be exposed by the components which have a full access to Content-Providers, i.e. those composed with Content-Providers inside the same application. These components can be attacked by malicious intents [1], composed of incorrect data or attacks, that are indirectly forwarded to Content-Providers. As a result, data may be exported or modified. This work tackles this issue by proposing a Model-based testing method which automatically generates test cases from intent-based vulnerabilities.

Some works, relative to security vulnerabilities associated with intents, have been recently proposed in the literature: Zhong et al. showed that pre-installed Android applications receiving oriented intents can re-delegate wrong permissions [3]. Some tools have been developed to detect this issue. However, these tools are not tailored to detect other vulnerabilities. Jing et al. proposed a model-based conformance testing framework for the Android platform as well [4]. Basic specifications (only intent descriptions) are constructed from the configuration files of a project. Test cases are generated, from these specifications, to check whether intent-based properties hold. The set of properties defined in this paper is only based on the intent functioning and cannot be upgraded hence this approach lacks of scalability. Our work takes as input a larger set of vulnerability.

Based upon these remarks, we propose the following contributions: our method takes intent-based vulnerabilities formally expressed with *vulnerability patterns*. The latter are specialised ioSTSs (input output Symbolic Transition Systems [5]) which formally exhibit intent-based vulnerabilities and help define test verdicts without ambiguity. From vulnerability patterns, our method performs both the automatic test case generation and execution. The test case generation is achieved from vulnerability patterns, class diagrams and specifications automatically generated from the information provided in the Android documentation [6], the component compiled classes and the configuration files of an Android project. These class diagrams and specifications are used to determine the nature of each component (type, links to other components) and describe the functional behaviours that should be observed from components after receiving intents. These items help refine and reduce the test case generation. In particular, since the paper is dealing with data vulnerabilities, test cases shall be constructed from the components that are composed with Content-Providers only. Afterwards, we introduce the evaluation of the tool *APSET* ((Android aPplications SEcurity Testing)) which implements this method. The experimentation results show that this tool is effective in detecting vulnerability flaws.

The paper is structured as follows: Section II recalls some ioSTS definitions and notations. Vulnerability patterns are defined in Section III. The testing methodology is described in Section IV. We give some experimentation results in Section V and we conclude in Section VI.

II. Model Definition and notations.

We shall consider the input/output Symbolic Transition Systems (ioSTS) model [5] to generate partial specifications of Android components and to express vulnerabilities. An ioSTS is a kind of automata model which is extended with two sets of variables, internal variable to store data, and parameters to enrich the actions. Transitions carry actions, guards and assignments over variables. The action set is separated with inputs beginning by ? to express actions expected by the system, and with outputs beginning by ! to express actions produced by the system. An ioSTS does not have states but locations.

Below, we give the definition of an ioSTS extension, called ioSTS suspension, which also expresses quiescence i.e., the authorised deadlocks observed from a location. For an ioSTS $\mathcal{S}$, quiescence is modelled by a new action $!\delta$ and an augmented ioSTS denoted $\mathcal{S}^{\delta}$, obtained by adding a self-loop labelled by $!\delta$ for each location where no output action may be observed.

Definition 1 (ioSTS suspension) *A deterministic ioSTS suspension $\mathcal{S}^{\delta}$ is a tuple $< L, l0, V, V0, I, \Lambda, \to>$, where:*

- *L is the finite set of locations, $l0$ the initial location,*
- *V is the finite set of internal variables, I is the finite set of parameters. We denote D_v the domain in which a variable v takes values. The internal variables are initialised with the assignment $V0$ on V, which is assumed to be unique,*
- *Λ is the finite set of symbolic actions $a(p)$, with $p = (p_1, ..., p_k)$ a finite list of parameters in $I^k (k \in \mathbb{N})$. p is assumed unique. $\Lambda = \Lambda^I \cup \Lambda^O \cup \{!\delta\}$: Λ^I represents the set of input actions, (Λ^O) the set of output actions,*
- *$\to$ is the finite transition set. A transition $(l_i, l_j, a(p), G, A)$, from the location $l_i \in L$ to $l_j \in L$, denoted $l_i \xrightarrow{a(p),G,A} l_j$ is labelled by an action $a(p) \in \Lambda$. G is a guard over $(p \cup V \cup T(p \cup V))$ which restricts the firing of the transition. $T(p \cup V)$ is a set of functions that return boolean values only (a.k.a. predicates) over $p \cup V$. Internal variables are updated with the assignment function A of the form $(x := A_x)_{x \in V}$, A_x is an expression over $V \cup p \cup T(p \cup V)$*
- *for any location $l \in L$ and for all pair of transitions $(l, l_1, a(p), G_1, A_1)$, $(l, l_2, a(p), G_2, A_2)$ labelled by the same action, $G_1 \wedge G_2$ is unsatisfiable.*

An ioSTS is also associated to an ioLTS (Input/Output Labelled Transition System) to formulate its semantics. Intuitively, ioLTS semantics correspond to valued automata without symbolic variable, which are often infinite: ioLTS states are labelled by internal variable valuations while transitions are labelled by actions and parameter valuations. The semantics of an ioSTS $\mathcal{S} =< L, l_0, V, V_0, I, \Lambda, \to>$ is the ioLTS $[\![\mathcal{S}]\!] =< Q, q_0, \sum, \to>$ composed of valued states in $Q = L \times D_V$, $q_0 = (l_0, V_0)$ is the initial one, $\sum$ is the set of valued symbols and $\to$ is the transition relation. The ioLTS semantics definition of can be found in [5].

Intuitively, for an ioSTS transition $l_1 \xrightarrow{a(p),G,A} l_2$, we obtain an ioLTS transition $(l_1, v) \xrightarrow{a(p),\theta} (l_2, v')$ with v a set of valuations over the internal variable set, if there exists a parameter valuation set θ such that the guard G evaluates to true with $v \cup \theta$. Once the transition is executed, the internal variables are assigned with v' derived from the assignment $A(v \cup \theta)$. Runs and traces of an ioSTS can now be defined from its semantics:

Definition 2 (Runs and traces) *For an ioSTS $\mathcal{S} = < L, l0, V, V0, I, \Lambda, \to>$, interpreted by its ioLTS semantics $[\![\mathcal{S}]\!] =< Q, q_0, \sum, \to>$, a run $q_0\alpha_0 ... \alpha_{n-1}q_n$ is an alternate sequence of states and valued actions. $Run(\mathcal{S}) = Run([\![\mathcal{S}]\!])$ is the set of runs found in $[\![\mathcal{S}]\!]$. $Run_F(\mathcal{S})$ is the set of runs of $\mathcal{S}$ finished by a state in $F \times D_V \subseteq Q$, with F a location set in L.*

It follows that a trace of a run r is defined as the projection $proj_{\sum}(r)$ on actions. $Traces_F(\mathcal{S}) = Traces_F([\![\mathcal{S}]\!])$ is the set of traces of all runs finished by states in $F \times D_V$.

Below, we recall the definition of some classic operations on ioSTSs. The same operations can be also applied on underlying ioLTS semantics.

An ioSTS can be completed on its output set to express incorrect behaviours that are modelled with new transitions to the sink location Fail, guarded by the negation of the union of guards of the same output action on outgoing transitions:

Definition 3 (Output completion) *The output completion of a deterministic ioSTS $\mathcal{S} =< L, l0, V, V0, I, \Lambda, \to>$ gives the ioSTS $\mathcal{S}^! =< L \cup \{Fail\}, l0, V, V0, I, \Lambda, \to \cup \{(l, Fail, a(p), \bigwedge_{(l,l',a(p),G,A) \in \to} \neg G, (x := x)_{(x \in V)}) \mid l \in L, a \in \Lambda^O\} >$*

Definition 4 (ioSTS product $\times$) *The product of the ioSTS $\mathcal{S}_1 =< L_1, l0_1, V_1, V0_1, I_1, \Lambda_1, \to_1>$ with the ioSTS $\mathcal{S}_2 = < L_2, l0_2, V_2, V0_2, I_2, \Lambda_2, \to_2>$, denoted $\mathcal{S}_1 \times \mathcal{S}_2$, is the ioSTS $\mathcal{P} =< L_\mathcal{P}, l0_\mathcal{P}, V_\mathcal{P}, V0_\mathcal{P}, I_\mathcal{P}, \Lambda_\mathcal{P}, \to_\mathcal{P}>$ such that $V_\mathcal{P} = V_1 \cup V_2$, $V0_\mathcal{P} = V0_1 \wedge V0_2$, $I_\mathcal{P} = I_1 \cup I_2$, $L_\mathcal{P} = L_1 \times L_2$, $l0_\mathcal{P} = (l0_1, l0_2)$, $\Lambda_\mathcal{P} = \Lambda_1 \cup \Lambda_2$. The transition set $\to_\mathcal{P}$ is the smallest set satisfying the following inference rules:*

(1) $l_1 \xrightarrow{a(p),G_1,A_1}_{\mathcal{S}_1} l_2, l'_1 \xrightarrow{a(p),G_2,A_2}_{\mathcal{S}_2} l2' \vdash (l_1, l'_1) \xrightarrow{a(p),G_1 \wedge G_2,A_1 \cup A_2}_{\mathcal{P}} (l_2, l'_2)$

(2) $l_1 \xrightarrow{a(p),G_1,A_1}_{\mathcal{S}_1} l_2, a(p) \notin \Lambda_2, l'_1 \in L_2 \vdash (l_1, l'_1) \xrightarrow{a(p),G_1,A_1 \cup \{x:=x\}_{x \in V_2}}_{\mathcal{P}} (l_2, l'_1)$ (and symmetrically for $a(p) \notin \Lambda_1, l_1 \in L_1$)

The parallel composition of two ioSTSs is a specialised product which illustrates shared behaviours only of the two original ioSTSs that are compatible:

Definition 5 (Compatible ioSTSs) *An ioSTS* $\mathcal{S}_1 = < L_1, l0_1, V_1, V0_1, I_1, \Lambda_1, \rightarrow_1 >$ *is compatible with* $\mathcal{S}_2 = < L_2, l0_2, V_2, V0_2, I_2, \Lambda_2, \rightarrow_2 >$ *iff* $V_1 \cap V_2 = \emptyset$, $\Lambda_1^I = \Lambda_2^I$, $\Lambda_1^O = \Lambda_2^O$ *and* $I_1 = I_2$.

Definition 6 (Parallel composition $||$) *The parallel composition of two compatible ioSTSs* $\mathcal{S}_1, \mathcal{S}_2$*, denoted* $\mathcal{S}_1 || \mathcal{S}_2$*, is the ioSTS* $\mathcal{P} =< L_\mathcal{P}, l0_\mathcal{P}, V_\mathcal{P}, V0_\mathcal{P}, I_\mathcal{P}, \Lambda_\mathcal{P}, \rightarrow_\mathcal{P} >$ *such that* $V_\mathcal{P} = V_1 \cup V_2$, $V0_\mathcal{P} = V0_1 \wedge V0_2$, $I_\mathcal{P} = I_1 \cup I_2$, $L_\mathcal{P} = L_1 \times L_2$, $l0_\mathcal{P} = (l0_1, l0_2)$, $\Lambda_\mathcal{P} = \Lambda_1 \cup \Lambda_2$*. The transition set* $\rightarrow_\mathcal{P}$ *is the smallest set satisfying the first rule of the product definition only.*

Lemma 1 (Parallel composition traces) *If* $\mathcal{S}_2$ *and* $\mathcal{S}_1$ *are compatible then* $Traces_{F_1 \times F_2}(\mathcal{S}_1 || \mathcal{S}_2) = Traces_{F_1}(\mathcal{S}_1) \cap Traces_{F_2}(\mathcal{S}_2)$*, with* $F_1 \subseteq L_{\mathcal{S}_1}$*,* $F_2 \subseteq L_{\mathcal{S}_2}$*.*

III. Vulnerability modelling

Several formalisms have been proposed in the literature to describe security vulnerabilities or attacks, e.g., regular expressions, temporal and deontic logics or state machines. We chose the latter because it sounds more user-friendly to express intent-based vulnerabilities that do not require obligation, permission, and related concepts.

Rather than defining the vulnerabilities of a specification, (which have to be written for each specification), we chose defining vulnerability patterns for describing intent-based vulnerabilities of an Android component type. A vulnerability pattern, denoted $\mathcal{V}$, is a specialised ioSTS suspension composed of two distinct final locations Vul, $NVul$ which aim to recognise the vulnerability status over component executions. Intuitively, runs of a vulnerability pattern, starting from the initial location and ended by Vul, express the presence of the vulnerability. By deduction, runs ended by $NVul$ express functional behaviours which show the absence of the vulnerability. $\mathcal{V}$ is also output-complete to recognise a status whatever the actions observed while testing.

Naturally, a vulnerability pattern $\mathcal{V}$ has to be equipped of actions also used for describing behaviours of Android components. We denote $AuthAct_{type}$ the action set that can model a type of component in accordance with the Android documentation. Transition guards can also be composed of specific predicates to ease their writing. In the paper, we consider some predicates such as *in*, which stands for a Boolean function returning true if a parameter list belongs to a given value set. In the same way, we consider several value sets to categorise malicious values and attacks: RV is a set of values known for relieving bugs enriched with random values. Inj is a set gathering XML and SQL injections constructed from database table URIs found in the tested Android application. URI is a set of randomly constructed URIs completed with the URIs found in the tested Android application. New sets can be also added upon condition that real value sets with the same name would be added to the testing tool.

Definition 7 (Vulnerability pattern) *A Vulnerability pattern is a deterministic and output-complete ioSTS suspension* $\mathcal{V}$ *such that the sink locations of* $L_\mathcal{V}$ *belong to* $\{Vul, NVul\}$*.* $type(\mathcal{V})$ *is the component (or component composition) type targeted by* $\mathcal{V}$*. The action set* $\Lambda_\mathcal{V} = AuthAct_{type}$ *where* $type$ *is equal to* $type(\mathcal{V})$*.*

In this paper, we focus on intent-based vulnerabilities of components exposing personal data managed by Content-Providers. Activities, and Services are the two Android components that can interact with Content-Providers. A Vulnerability pattern $\mathcal{V}$ is then composed of actions of a component $Comp$ (Activity or Service), composed with a Content-Provider Cp. Consequently, $\Lambda_\mathcal{V} = AuthAct_{Comp} \cup AuthAct_{Cp}$. For readability, we consider Activities only in the paper.

Activities are the most common Android components which display screens to let users interact with programs. We denote $!Display(A)$ the action modelling the display of a screen for the Activity A. Activities may also throw exceptions that we group into two categories: those raised by the Android system on account of the crash of a component and the other ones. This difference can be observed while testing with our framework. This is modelled with the actions *!SystemExp* and *!ComponentExp* respectively.

Components are tied together with intents, denoted $intent(Cp, a, d, c, t, ed)$ with Cp the called component, a an action which has to be performed, d a data expressed as a URI, c a component category, t a type which specifies the MIME type of the intent data and finally ed which represent additional (extra) data [6]. Intent actions have different purposes, e.g., the action VIEW is called to display something, the action PICK is called to choose an item and to return its URI to the calling component. Hence, in reference to the Android documentation [6], the action set, denoted ACT, is divided into two categories: the action set ACT_r gathers the actions requiring the receipt of a response, ACT_{nr} gathers the other actions. We also denote C, the set of predefined Android categories, T the set of types. Finally, one deduce that $AuthAct_{Activity}$ is the set $\{?intent(Cp, a, d, c, t, ed), !Display(A), !SystemExp, !ComponentExp, !\delta\}$.

Content-Providers are components which receive SQL-oriented requests (no intents) denoted $!call(Cp, request, tableURI)$ and eventually return responses denoted $!callResp(Cp, resp)$. Consequently, $AuthAct_{Content-Provider} = \{!call(Cp, request, tableURI), !callResp(Cp, resp), !\delta, !ComponentExp, !SystemExp\}$.

Figure 1 illustrates a straightforward example of vulnerability pattern, related to data integrity. It aims to check whether an Activity, called with an intent composed of malicious data, cannot alter the content of a database table managed

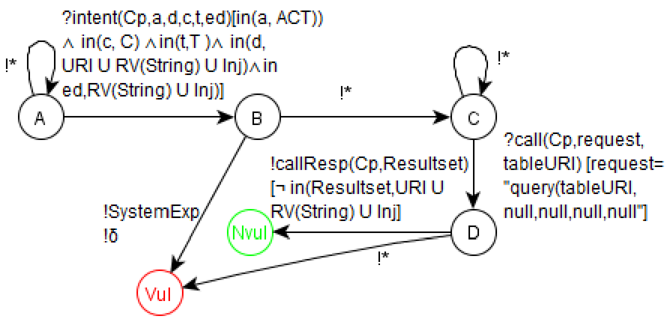

Fig. 1: Vulnerability pattern example

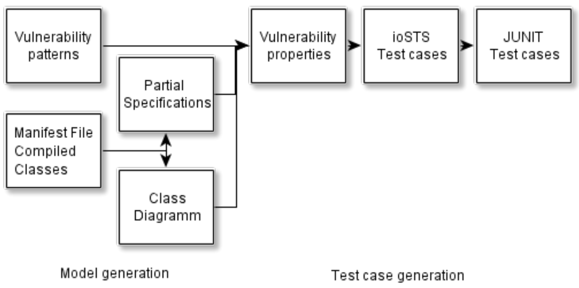

Fig. 2: Test case generation

by a Content-Provider. Intents are constructed with data and extra data composed of malformed URIs or String values known for relieving bugs or XML/SQL injections. If the called component crashes, it is considered as vulnerable. Once the intent is performed, the Content-Provider is called with the *query* function of the Android SDK to retrieve all the data stored in a table whose URI is given by the variable *tableURI*. If the result set is not composed of incorrect data given in the intent, then the component is not vulnerable. Otherwise, it is vulnerable. The label $!*$ is a shortcut notation for all valued output actions that are not explicitly labelled by other transitions.

Considering an ioSTS $\mathcal{S}$ compatible with a vulnerability pattern $\mathcal{V}$, the vulnerability status of $\mathcal{S}$ is given when its suspension traces are recognised by the locations Vul and $NVul$:

Definition 8 (Vulnerability status of an ioSTS) *Let $\mathcal{S}$ be an ioSTS, $\mathcal{V}$ be a vulnerability pattern such that $\mathcal{S}^\delta$ is compatible with $\mathcal{V}$. We define the vulnerability status of $\mathcal{S}$ (and of its underlying ioLTS semantics $[\![\mathcal{S}]\!]$) over $\mathcal{V}$ with:*

- *$\mathcal{S}$ is not vulnerable to $\mathcal{V}$, denoted $\mathcal{S} \models \mathcal{V}$ if $Traces(\mathcal{S}^\delta) \subseteq Traces_{NVul}(\mathcal{V})$,*
- *$\mathcal{S}$ is vulnerable to $\mathcal{V}$, denoted $\mathcal{S} \not\models \mathcal{V}$ if $Traces(\mathcal{S}^\delta) \cap Traces_{Vul}(\mathcal{V}) \neq \emptyset$.*

IV. Security Testing Methodology

A component under test (CUT) is regarded as a black box whose interfaces are known only. However, one usually assumes the following test hypotheses to carry out the test case execution:

- the functional behaviours of the component under test, observed while testing, can be modelled by an ioLTS CUT. CUT is unknown (and potentially nondeterministic). CUT is assumed input-enabled (it accepts any of its input actions from any of its states),
- to dialog with CUT, one assumes that CUT is a composition of an ACtivity or Service with a Content-Provider, whose type is equal to $type(\mathcal{V})$ and that it is compatible with $\mathcal{V}$.

The test case generation involves the following major steps, illustrated in Figure 2: we assume having a set of vulnerability patterns modelled with ioSTS suspensions. From a decompiled Android application, a partial class diagram is extracted which lists the components and the associations between them. We keep only the components composed with Content-Providers here. From the Android configuration file called $Manifest$, an ioSTS suspension is generated for each component. It describes the component behaviours after the receipt of intents combined with Content-Providers requests. Models, called vulnerability properties, are then derived from the composition of vulnerability patterns with specifications. Test cases are obtained by concretising vulnerability properties to obtain executable test cases only. Finally, the latter are translated into JUNIT test cases. These steps are detailed below.

A. Model generation

Android applications gather a lot of information that can be used to produce partial specifications. For this method, we generate the following structures and models:

1) a simplified class diagram is generated by means of Java reflection. This class diagram depicts Android components with their types and gives some information about the relationships between components. In particular, this step gives the Activities or Services composed with Content-Providers: $L_C = \{ct_i \times cp_j\}$ is the set gathering the combinations of a component ct_i with a Content-Provider cp_j.

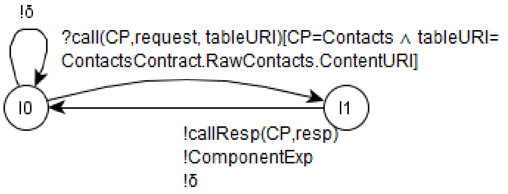
!δ
?call(CP,request, tableURI)[CP=Contacts ∧ tableURI=
ContactsContract.RawContacts.ContentURI]
l0
l1
!callResp(CP,resp)
!ComponentExp
!δ

Fig. 3: Content-Provider specification

2) an ioSTS suspension $\mathcal{S}_{ct_i \times cp_j} = (S_{ct_i} \times S_{cp_j})^!$ is generated for each item of L_C such that $type(ct_i \times cp_j)$, e.g., Activity $\times$ Content-Provider is also the type of the vulnerability pattern $\mathcal{V}$. S_{cp_j} is an ioSTS suspension modelling the call of the Content-Provider cp_j, derived from a generic ioSTS where only the Content-Provider name and the variable tableURI are updated

from the information found in the projet configuration file called *Manifest*. An example is depicted in Figure 3 for the Contact-Provider with the table "ContactsContract.RawContacts". The Contact-Provider is a specialised Content-Provider managing contact information. Naturally, this specification is written in accordance with the set $AuthAct_{Content-Provider}$.
S_{ct_i} is the ioSTS suspension of the component ct_i constructed by means of the intent filters listed in the Manifest file of the Android project. An intent filter *IntentFilter(act,cat,data,type)* declares a type of intent accepted by the component. For readability, we present the algorithm dedicated to Activities only in Algorithm 1. Initially, the action set of $\Lambda_{S_{ct_i}}$ is set to $AuthAct_{Activity}$. Then, Algorithm 1 produces the ioSTS suspension S_{ct_i} from intent filters and with respect to the intent functioning, described in the Android documentation. It covers each intent filter and adds one transition carrying an intent followed by two transitions labelled by output actions. (lines 7-15). Depending on the action type read in the intent filter, the guard of a transition equipped by an output action is completed to reflect the fact that a response may be received or not. For instance, an action in ACT_r (line 9), implies both the display of a screen and the receipt of a response. If the action of the intent filter is unknown (lines 13,14), no guard is formulated on the output action (a response may be received or not).
Finally, the product $(S_{ct_i} \times S_{cp_j})$ is completed on the output action set to express incorrect behaviours, modelled with new transitions to a $Fail$ location. The $Fail$ location shall be particularly useful to refine the test verdict by helping recognise correct and incorrect behaviours of an Android component w.r.t. its specification. For the Service components, the specification generation algorithm is similar.

Figure 4 illustrates a specification example which stems from the product of one Activity with the Contact-Provider. This composition accepts intents composed of the action *PICK* and data whose URI corresponds to the contact list stored in the device. It returns responses (probably a chosen contact). This composition also accepts requests to the Contact-Provider. Incorrect behaviours are expressed with transitions to Fail.

Algorithm 1: Component specification Generation

```
input  : Manifest file $MF$
output: Partial specifications $S_{ct_i}$
1  foreach component $ct_i$ in $MF$ do
2      $it := 0$;
3      $S_{ct_i}$ is the ioSTS specification of $ct_i$ ;
4      $\Lambda_{S_{ct_i}} = AuthAct_{type(ct_i)}$;
5      Add $l0_{S_{ct_i}} \xrightarrow{!\delta}_{S_{ct_i}} l0_{S_{ct_i}}$ to $\rightarrow_{S_{ct_i}}$ ;
6      if type of $ct_i$ == Activity then
7          foreach IntentFilter(act,cat,data,type) of $ct_i$ in $MF$ do
8              $it := it + 1$ ;
9              if $act \in ACT_r$ then
10                 Add $l0_{S_{ct_i}} \xrightarrow{?evt_1^{(1)}}_{S_{ct_i}} (l_{it,1}) \xrightarrow{!di_1^{(2)},[ct_i.resp \neq Null]} l0_{S_{ct_i}}$ to $\rightarrow_{S_{ct_i}}$
11             else if $act \in ACT_{nr}$ then
12                 Add $l0_{S_{ct_i}} \xrightarrow{?evt_1^{(1)}}_{S_{ct_i}} (l_{it,1}) \xrightarrow{!di_1^{(2)},[ct_i.resp = Null]} l0_{S_{ct_i}}$ to $\rightarrow_{S_{ct_i}}$
13             else
14                 Add $l0_{S_{ct_i}} \xrightarrow{?evt_1^{(1)}}_{S_{ct_i}} (l_{it,1}) \xrightarrow{!di_1^{(2)}} l0_{S_{ct_i}}$ to $\rightarrow_{S_{ct_i}}$
15             Add $(l_{it,1}) \xrightarrow{!ComponentExp}_{S_{ct_i}} l0_{S_{ct_i}}$ to $\rightarrow_{S_{ct_i}}$ ;
16 (1) $?intent(Cp, a, d, c, t, ed)[Cp = ct_i \wedge a = act \wedge d = data \wedge c = cat \wedge t = type], A = (x := x)_{x \in V_{S_{ct_i}}}$
17 (2) $!Display(Activity\ a)[Cp = ct_i], A = (x := x)_{x \in V_{S_{ct_i}}}$
```

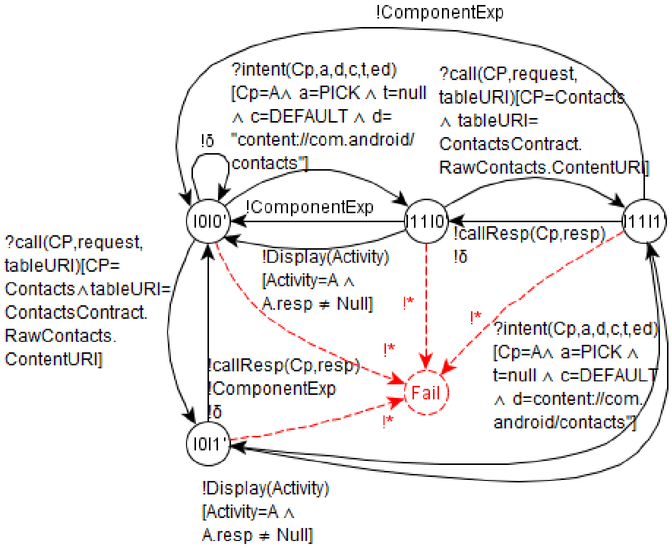

Fig. 4: A specification example

B. Test case selection

Test cases are extracted after composing vulnerability patterns with specifications. Given a vulnerability pattern $\mathcal{V}$ compatible with a specification $\mathcal{S}_{ct_i \times cp_j}$, the composition $\mathcal{VP}_{ct_i \times cp_j} = (\mathcal{V} || \mathcal{S}_{ct_i \times cp_j})$ is called a vulnerability property of $\mathcal{S}_{ct_i \times cp_j}$. It represents the vulnerable and non-vulnerable behaviours that can be observed from $ct_i \times cp_j$. Besides, the parallel composition $\mathcal{V} || \mathcal{S}_{ct_i \times cp_j}$ produces new locations and, in particular, new final verdict locations:

Definition 9 (Verdict location sets) *Let $\mathcal{V}$ be a vulnerability pattern and $\mathcal{S}_{ct_i \times cp_j}$ a specification compatible with $\mathcal{V}$. The vulnerability property $\mathcal{VP}_{ct_i \times cp_j} = \mathcal{V}||\mathcal{S}_{ct_i \times cp_j}$ is composed of new locations recognising vulnerability status:*

1) ***NVUL*** $= NVul \times L_{\mathcal{S}_{ct_i \times cp_j}}$. ***NVUL/FAIL*** $= (NVul, Fail) \in NVUL$ *aims to recognise incorrect behaviours w.r.t. the specification $\mathcal{S}_{ct_i \times cp_j}$ and not vulnerable behaviours w.r.t. $\mathcal{V}$,*
2) ***VUL*** $= Vul \times L_{\mathcal{S}_{ct_i \times cp_j}}$. ***VUL/FAIL*** $= (Vul, Fail)$ *aims to recognise incorrect behaviours w.r.t. $\mathcal{S}_{ct_i \times cp_j}$ and vulnerable behaviours w.r.t. $\mathcal{V}$.*

Test cases are achieved with Algorithm 2 which performs two main steps. Firstly, it splits a vulnerability property $\mathcal{VP}_{ct_i \times cp_j}$ into several ioSTSs. Intuitively, from a location l having k transitions carrying an input action, e.g., an intent, k new test cases are constructed to experiment CUT with the k input actions and so on for each location l having transitions labelled by input actions (lines 1-4). Then, a set tuple of valuations is computed for the list of undefined parameters of each input action (line 5). For instance, intents are composed of several variables whose domains are given in guards. These ones have to be concretised before testing. Instead of using a cartesian product to construct a tuple of valuations, we adopt a Pairwise technique [7]. This technique strongly reduces the coverage of a variable domain by constructing discrete combinations for pair of parameters only. The set of valuation tuples is constructed with the *Pairwise* procedure which takes the list of undefined parameters and the transition guard to find the domain of each parameter. In the second step (line 6-13), input actions are concretised, i.e. each undefined parameter of an input action is assigned to a value. Given a transition t and its set of valuation tuples $P(t)$, this step constructs a new test case for each tuple $pv = (p_1 = v_1, ..., p_n = v_n)$ by replacing the guard G with $G \wedge pv$ if $G \wedge pv$ is satisfiable. Finally, if the resulting ioSTS suspension tc has verdict locations, then tc is added in the test case set TC. Steps 1. and 2. are iteratively applied until until each combination of transitions carrying input actions and each combination of valuation tuples are covered. Since the algorithm may produce a large set of test cases, depending on the number of tuples of valuations given by the Pairwise function, the algorithm also ends when the test case set TC reaches a cardinality of $tcnb$ (lines 17,18).

A test case example, derived from the specification of Figure 4 and the vulnerability pattern of Figure 1 is depicted in Figure 5. It expresses the sending of an intent with the extra data parameter composed of an SQL injection. Then, the data managed by the Contact-Provider must not have been modified. Otherwise, the component is vulnerable. If it crashes, it is vulnerable as well.

A test case constructed with Algorithm 2, from a vulnerability property $\mathcal{VP}_{ct_i \times cp_j}$, produces traces that belong to the trace set of $\mathcal{VP}_{ct_i \times cp_j}$. In other words, the test selection algorithm does not add new traces leading to verdict locations. Indeed, a test case is composed of paths of a vulnerability property, starting from its initial location. Each guard G' of a test

Algorithm 2: Test case generation

```
input  : A vulnerability property $\mathcal{VP}_{ct_i \times cp_j}$, $tcnb$ the maximal
         number of test cases
output: Test case set $TC$
1  begin 1. input action choice
2      foreach location $l$ having outgoing transitions carrying input actions do
3          Choose a transition $t = l \xrightarrow{?a(p),G,A}_{\mathcal{VP}_{ct_i \times cp_j}} l_2$;
4          remove the other transitions labelled by input actions;
5          $P(t) = Pairwise(p_1, ..., p_n, G)$ with $(p_1, ..., p_n) \subseteq p$ the
           list of undefined parameters;
6  begin 2. input concretisation
7      foreach $t = l \xrightarrow{?a(p),G,A}_{\mathcal{VP}_{ct_i \times cp_j}} l_2$ do
8          Choose a valuation tuple $pv = (p_1 = v_1, ..., p_n = v_n)$ in
           $P(t)$;
9          if $G \wedge pv$ is satisfiable then
10             Replace $G$ by $G \wedge pv$ in $t$;
11         else
12             Choose another valuation tuple in $P(t)$;
13     $tc$ is the resulting ioSTS suspension;
14 begin 3.
15     if $tc$ has reachable verdict locations then
16         $TC := TC \cup \{tc\}$ ;
17     if $Card(TC) \geq tcnb$ then
18         STOP;
19     Repeat 1. and 2. until each combination of transitions carrying
       input actions and each combination of valuation tuples are
       covered;
```

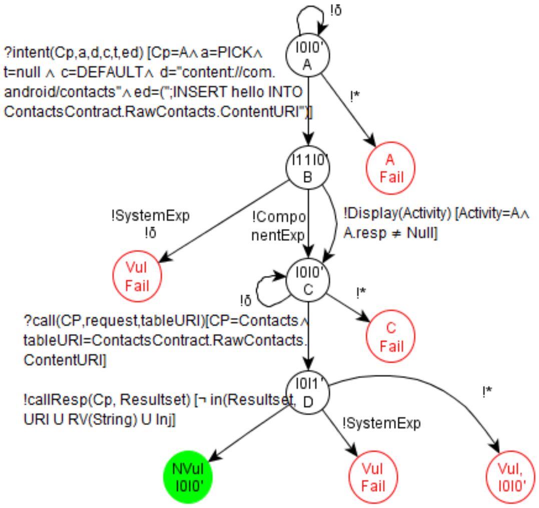
!δ
I0I0'
A
?intent(Cp,a,d,c,t,ed) [Cp=A∧ a=PICK∧
t=null ∧ c=DEFAULT∧ d="content://com.
android/contacts"∧ed=(";INSERT hello INTO
ContactsContract.RawContacts.ContentURI")]
!*
A
Fail
I1 1I0'
B
!SystemExp
!δ
!Compo
nentExp
!Display(Activity) [Activity=A∧
A.resp ≠ Null]
Vul
Fail
I0I0'
C
!δ
!*
C
Fail
?call(CP,request,tableURI)[CP=Contacts∧
tableURI=ContactsContract.RawContacts.
ContentURI]
I0I1'
D
!callResp(Cp, Resultset) [¬ in(Resultset,
URI U RV(String) U Inj]
!SystemExp
!*
NVul
I0I0'
Vul
Fail
Vul,
I0I0'

Fig. 5: A test case example

case transition carrying a input action stems from a guard G completed with a tuple of valuations such that if G' is satisfied then G is also satisfied. This is captured by the following Proposition:

Proposition 10 *Let $\mathcal{VP}_{ct_i \times cp_j} = \mathcal{V}||\mathcal{S}_{ct_i \times cp_j}$ be a vulnerability property. TC is the test case set generated from $\mathcal{VP}_{ct_i \times cp_j}$ with Algorithm 2. We have $\forall tc \in TC$, $Traces(tc) \subseteq Traces(\mathcal{V}||\mathcal{S}_{ct_i \times cp_j})$.*

C. Test case execution definition

The test case execution is defined by the parallel composition of the test cases with the implementation under test CUT:

Proposition 11 (Test case execution) *Let TC be a test case set obtained from the vulnerability pattern $\mathcal{V}$ and the specification $\mathcal{S}_{ct_i \times cp_j}$. CUT is the ioLTS of the component under test, assumed compatible with $\mathcal{V}$. For all test case $tc \in TC$, the execution of tc on CUT is defined by the parallel composition $tc||CUT^{\delta}$.*

Remark 12 *A test case tc obtained from a vulnerability pattern $\mathcal{V}$, can be experimented on CUT since tc and CUT are compatible. Indeed, tc is produced from a vulnerability property $\mathcal{VP}_{ct_i \times cp_j} = (\mathcal{V}||\mathcal{S}_{ct_i \times cp_j})$ such that the action set $\Lambda_{\mathcal{S}_{ct_i \times cp_j}} = \Lambda_{\mathcal{V}}$. From this equality and Algorithm 2, one can deduce that tc has the same action set as $\mathcal{VP}_{ct_i \times cp_j}$ and as $\mathcal{V}$. Since we assume that CUT is compatible with $\mathcal{V}$, tc can be experimented on CUT.*

The above proposition leads to the test verdict of a component under test against a vulnerability pattern $\mathcal{V}$. Intuitively, the test verdict refers to the Vulnerability status definition, completed by the detection of incorrect behaviours described in the specification with the verdict locations *VUL/FAIL* and *NVUL/FAIL*. An inconclusive verdict is also defined when a verdict location has not been reached after a test case execution:

Definition 13 (Test verdict) *We take back the notations of Proposition 11. The execution of the test case set TC on CUT yields one of the following verdicts:*

- *CUT is vulnerable to $\mathcal{V}$ iff $\exists tc \in TC$, $tc||CUT$ produces a trace σ such that σ is also a trace of $Traces_{VUL}(tc)$. If σ is a trace of $Traces_{VUL/FAIL}(tc)$ then CUT does not also respect the component normal functioning modelled by $\mathcal{S}_{ct_i \times cp_j}$,*
- *CUT is not vulnerable to $\mathcal{V}$ iff $\forall tc \in TC$, $tc||CUT$ produces a trace σ such that σ is also a trace of $Traces_{NVUL}(tc)$. However, if σ is a trace of $Traces_{NVUL/FAIL}(tc)$ then CUT does not respect the component normal functioning,*
- *CUT has an unknown status iff $\exists tc \in TC, tc||CUT$ produces a trace σ such that $\sigma \notin Traces_{VUL}(tc) \cup Traces_{NVUL}(tc)$.*

Proof:

Sketch of proof of 1: $\exists tc \in TC$ such that $[\![tc]\!]||CUT^{\delta}$ produces a trace $\sigma \in Traces_{VUL}(tc)$.
$Traces_{VUL}(tc) \cap Traces(CUT^{\delta}) \neq \emptyset$ (Lemma 1)
$Traces_{VUL}(\mathcal{V}||\mathcal{S}_{ct_i \times cp_j}) \cap Traces(CUT^{\delta}) \neq \emptyset$ (Proposition 10)
$Traces_{VUL}(\mathcal{V}||\mathcal{S}_{ct_i \times cp_j}) = Traces_{Vul}(\mathcal{V}) \cap Traces_{L_{\mathcal{S}_{ct_i \times cp_j}}}(\mathcal{S}_{ct_i \times cp_j})$ since $\mathcal{S}_{ct_i \times cp_j}$ is compatible with $\mathcal{V}$ (Algorithm 1 and Lemma 1)
We have $(Traces_{Vul}(\mathcal{V}) \cap Traces_{L_{\mathcal{S}_{ct_i \times cp_j}}}(\mathcal{S}_{ct_i \times cp_j}))) \cap Traces(CUT^{\delta}) \neq \emptyset$. Hence, $Traces_{Vul}(\mathcal{V}) \cap Traces(CUT^{\delta}) \neq \emptyset$ (a) and $Traces_{L_{\mathcal{S}_{ct_i \times cp_j}}}(\mathcal{S}_{ct_i \times cp_j}) \cap Traces(CUT^{\delta}) \neq \emptyset$ (b). From (a), we obtain $CUT \not\models \mathcal{V}$ (Definition 7). Consequently, CUT is vulnerable to $\mathcal{V}$.
If $\sigma \in Traces_{VUL/FAIL}(tc)$ then, from (b) we have $Traces_{Fail}(\mathcal{S}_{ct_i \times cp_j}) \cap Traces(CUT^{\delta}) \neq \emptyset$. σ represents an incorrect behaviour of the partial specification $\mathcal{S}_{ct_i \times cp_j}$.

■

V. Experimentation

The above security testing method has been implemented in a tool called *APSET* (Android aPplications SEcurity Testing), publicly available in a Github repository \footnote{https://github.com/statops/apset.git}. It takes an Android application (uncompressed .apk file) and vulnerability patterns, builds ioSTS specifications and generates JUNIT test cases. Then it executes them on Android emulators or devices and displays the final verdicts. For any action defined in Section III, a corresponding function has to coded in the tool. For instance, the action $!Display(A)$ is represented by the function Display() which returns true if a screen is displayed. The guard solving, used in Algorithm 2 and during the test case execution, is performed by the SMT (Satisfiability Modulo Theories) solver Z3 [1], whose language is augmented with the predicates given in Section III.

We experimented several real Android applications provided by the Openium company[2]and popular applications from Google play store [3](eg. app8=Youtube) with three vulnerability patterns: $\mathcal{V}1$, corresponds to the vulnerability pattern taken as example in the paper. $\mathcal{V}2$ checks whether an Activity called with intents composed of malicious data, cannot alter the structure of a database managed by a Content-Provider (modification of attribute names, removal of tables, etc.). $\mathcal{V}3$ checks that incorrect data, initially stored into a database, are not displayed by an Activity after having called it with an intent. A part of the experimentation results are depicted in Table I which illustrates respectively the number of tested components, the number of issues detected with each vulnerability pattern and the total number of test cases providing a vulnerable verdict. All the applications in Table I have vulnerability issues. For instance, with *app 3*, 102 test cases were generated and 17 showed vulnerability issues. 10 test cases showed that *app 3* is vulnerable to $\mathcal{V}1$. More precisely,

[1] http://z3.codeplex.com/

[2] www.openium.fr

[3] https://play.google.com/store

Applications		Test results				
Name	# component	$\mathcal{V}1$	$\mathcal{V}2$	$\mathcal{V}3$	#vul/ #testcases	Time
app 1	7	8	-	3	11/54	0,85
app 2	16	1	-	3	4/164	0,18
app 3	15	10	2	5	17/102	2,3
app 4	9	2	-	9	11/97	1,56
app 5	7	19	-	2	21/73	0,77
app 6	8	-	-	4	4/71	1,05
Maps	38	28	-	11	39/370	1,67
Youtube	12	3	-	-	3/131	3,21

TABLE I: Experimentation results

1 test case showed that personal data can be modified by using malicious intents. *app 3* crashed with the other test cases probably because of the bad handling of malicious intents by the components. 2 test cases also revealed that the structure of the database is modified ($\mathcal{V}2$). These modification or deletion issues are closely related to the implementation of the Content-Provider methods which are probably not protected against malicious requests. Finally, 5 test cases revealed the display of incorrect data stored in database ($\mathcal{V}3$). This means that the database content is directly displayed to the user interface, without any validation.

Table I also gives the average test case execution time, measured with Mid 2011 computer with a CPU 2.1Ghz Core i5. The execution time is included between some milliseconds up to 2.3 seconds, depending of the code of the tested components. Benli et al. showed in [8] that the execution time per test case may be up to 17s. All these results tend to show that our approach is effective and can be used in practice.

Acknowledgment

We would like to thank the Openium [4] company for providing advice and comments on this paper and Android applications.

VI. Conclusion

In this paper, we have presented a security testing method of Android applications which aims at detecting data vulnerabilities based on the intent mechanism. The originality of this work resides in the automatic generation of partial specifications, used to generate test cases. These specifications also enrich the test verdict with the verdicts NVUL/FAIL and VUL/FAIL, pointing out that the component under test does not meet the recommendations provided in the Android documentation. In future works, we intend to extend the generation of partial specifications. An immediate solution would be to compose the partial specifications of each component together to test a composite component. The latter could be tested by injecting malicious intents headed to sub-components.

References

[1] 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*, 2011, pp. 239–252.

[2] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *In Proceedings of the 18th ACM conference on Computer and communications security*, Oct. 2011, pp. 627–638.

[3] J. Zhong, J. Huang, and B. Liang, "Android permission re-delegation detection and test case generation," in *Computer Science Service System (CSSS), 2012 International Conference on*, aug. 2012, pp. 871 –874.

[4] Y. Jing, G.-J. Ahn, and H. Hu, "Model-based conformance testing for android." in *Proceedings of the 7th International Workshop on Security (IWSEC)*, ser. Lecture Notes in Computer Science, G. Hanaoka and T. Yamauchi, Eds., vol. 7631. Springer, 2012, pp. 1–18.

[5] 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.

[6] AD, "Android developer page," in *http:// developer.android.com/index.html, last accessed feb 2013*, 2013.

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

[8] S. Benli, A. Habash, A. Herrmann, T. Loftis, and D. Simmonds, "A comparative evaluation of unit testing techniques on a mobile platform," in *Proceedings of the 9th International Conference on Information Technology - New Generations*, ser. ITNG '12. IEEE Computer Society, 2012, pp. 263–268.

[4] www.openium.fr