Intent Security Testing
An Approach to Testing the Intent-based Vulnerability of Android Components

Sébastien Salva, Stassia R. Zafimiharisoa and Patrice Laurençot
LIMOS - UMR CNRS 6158, PRES Clermont-Ferrand University, Clermont-Ferrand, France

Keywords: Security Testing, Android Applications, Model-based Testing.

Abstract: The intent mechanism is a powerful feature of the Android platform that helps compose existing components together to build a Mobile application. However, hackers can leverage the intent messaging to extract personal data or to call components without credentials by sending malicious intents to components. This paper tackles this issue by proposing a security testing method which aims at detecting whether the components of an Android application are vulnerable to malicious intents. Our method takes Android projects and intent-based vulnerabilities formally represented with models called vulnerability patterns. The originality of our approach resides in the generation of partial specifications from configuration files and component codes to generate test cases. A tool, called APSET, is presented and evaluated with experimentations on some Android applications.

1 INTRODUCTION

As mobile usage grows, so should security: this sentence summarises well the conclusions of several recent reports (Report, 2012) providing analyses in Mobile threats. These reports accentuate the idea that Mobile security continues to be a global issue, independently of the platform. Security testing represents the most valuable solution to detect vulnerability flaws in Mobile systems and applications. In this context, the latter are experimented with test cases usually constructed by hands from known attacks or vulnerabilities. Model-based testing is another approach that brings some advantages, e.g., the automatisation of some steps or the definition of the confidence level of the test.

Mobile security testing is a very large field that depends on several different concepts such as threat families, internal mechanisms provided by the Mobile platform or more sophisticated concepts such as composition of software. This paper focuses on the Android inter-component communication mechanism, called *intent*, and describes an original testing method to detect intent-based vulnerabilities. This vulnerability family stems from the Android application structure: these applications consist of one or more core components tied together by means of intents. The intent concept is an inter-component communication mechanism, 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). Any component can freely interact with other exposed components, for example to request data. A malicious component can also leverage the intent mechanism to send attacks. So considered, the intent mechanism becomes an attack vector (Chin et al., 2011).

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 (Zhong et al., 2012). 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 (Jing et al., 2012). 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. This approach lacks of scalability though since the set of properties is based on the intent functioning and cannot be upgraded. So, testing the presence of new vulnerabilities being discovered cannot be achieved with this method.

The security testing method, introduced in this paper, aims at detecting whether components are vulnerable to malicious intents. The notion of vulnerability of a component is modelled with specialised IOSTSs (input output Symbolic Transition Systems (Frantzen et al., 2005)) called vulnerabilities patterns. This formal model leads to define vulnerabilities and finally test verdicts without ambiguity. Then, from vulnerability patterns, our method performs both the automatic test case generation and execution. The originality of this work resides in the test case generation. First, partial class diagrams and partial IOSTS specifications are generated from component compiled classes and configuration files. These class diagrams and specifications are used to determine the nature of each component and represent the functional behaviours that should be observed from each component after the receipt of an intent (in reference to the Android documentation (Android, 2013)). These items help refine and reduce the test case generation. For instance, vulnerability patterns dedicated to Activity components shall be only applied on the Activities of an application. IOSTS test cases are derived from a combination of vulnerability patterns with partial specifications.

The paper is structured as follows: Section 2 gives IOSTS definitions and notations to be used throughout the paper. Vulnerability patterns are defined in Section 3. Finally, the testing methodology is described in Section 4 and we conclude in Section 5.

2 MODEL DEFINITION AND NOTATIONS

We shall consider the input/output Symbolic Transition Systems (IOSTS) model (Frantzen et al., 2005) to generate partial specifications of Android components and to express vulnerabilities. Below, we recall 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 $< L, l0, V, V0, I, \Lambda, \rightarrow>$, 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,*
- *$\rightarrow$ 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, the ioLTS semantics corresponds to a valued automaton without symbolic variable, which is often infinite: the ioLTS states are labelled by internal variable valuations while transitions are labelled by actions and parameter valuations. The semantics of an IOSTS $\mathcal{S} =< L, l0, V, V0, I, \Lambda, \rightarrow>$ is the ioLTS $[\![\mathcal{S}]\!] =< Q, q_0, \sum, \rightarrow>$ composed of valued states in $Q = L \times D_V$, $q_0 = (l0, V0)$ is the initial one, $\sum$ is the set of valued symbols and $\rightarrow$ is the transition relation. The complete definition of ioLTS semantics can be found in (Frantzen et al., 2005). 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 value 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 $\mathcal{S} =< L, l0, V, V0, I, \Lambda, \rightarrow>$, interpreted by its ioLTS semantics $[\![\mathcal{S}]\!] =< Q, q_0, \sum, \rightarrow>$, 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 the parallel composition which is a classical state-machine operation used to represent the parallel execution of two systems. We give this definition for IOSTSs. However, the same operation can be also applied between two underlying ioLTS semantics. For IOSTSs, this parallel execution illustrates shared behaviours of the two original IOSTSs that are compatible:

Definition 3 (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 4 (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 following inference rule:*

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

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* $F_1 \subseteq L_{\mathcal{S}_1}$, $F_2 \subseteq L_{\mathcal{S}_2}$.

3 VULNERABILITY MODELLING

3.1 Android Applications and Notations

In this Section, we define some notations to model intent-based behaviours of Android components with IOSTSs. Android applications are usually constructed over a set of components. A component belongs to one of the four basic types: *Activities* (user interfaces) , *Services* (background processing), *Content providers* (SQLite database management) and *Broadcast receivers* (broadcast message handling). For readability, we essentially focus on Activities in the paper.

The inter-component communication is performed with intents. An intent, denoted $intent(a,d,c,t,ed)$ gathers: an action a which has to be performed, a data d expressed as a URI, and eventually a component category c, a type t which specifies the MIME type of the intent data and extra data ed representing additional data. Intents are divided into two groups: explicit intents, which explicitly target a component, and implicit intents (the most generally ones) which let the Android system choose the most appropriate component. Both can be exploited by a malicious application to send attacks to components since any component may determine the list of available components at runtime. As a consequence, we consider both implicit and explicit intents in this work. The mapping of an implicit intent to a component is expressed with items called *intent filters* stored in *Manifest* files. A Manifest file, is a part of any Android project and specifies configuration information about the whole application.

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 (Android, 2013), the action set, denoted ACT, is divided into categories to ease the vulnerability and component modelling: the action set ACT_r gathers the actions requiring the receipt of a response, ACT_{nr} gathers the other actions. We denote C, the set of predefined Android categories, T the set of types.

Android components may raise 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.

Finally, Android components, called by intents, produce different behaviours in reference to their types. For instance, the Activity role has to display screens (denoted *!Display(Activity a)*) with a response message or not, while a service usually aims to return a response only. To ease the writing of vulnerability patterns, we denote $AuthAct_{type}$ the action set that can be used with a type of component in accordance with the Android documentation. For instance, for Activities $AuthAct_{Activity} = \{?intent(a,d,c,t,ed), !\delta, !Display(A), !SystemExp, !ComponentExp\}$.

3.2 Vulnerability Patterns

We chose the IOSTS formalism to model vulnerability patterns because it powerful enough and still user-friendly to express intent vulnerabilities that do not require obligation, permission, and related concepts. Instead of defining the vulnerabilities of a specification, which have to be written for each specification, we prefer defining vulnerability patterns for describing intent-based vulnerabilities in general terms. A vulnerability pattern is a specialised IOSTS suspension composed of two distinct final locations Vul, $NVul$. The latter aim to recognise the vulnerability status over component executions: runs of a vulnerability pattern starting from the initial location and ended by

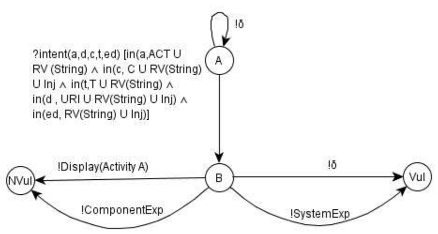
!δ
?intent(a,d,c,t,ed) [in(a,ACT U RV (String) ∧ in(c, C U RV(String) U Inj ∧ in(t,T U RV(String) ∧ in(d , URI U RV(String) U Inj) ∧ in(ed, RV(String) U Inj)]
A
B
NVul
Vul
!Display(Activity A)
!δ
!ComponentExp
!SystemExp

Figure 1: Vulnerability pattern for testing component availability.

Vul exhibit the presence of the vulnerability. By deduction, runs ended by *NVul* express functional behaviours which show the absence of the vulnerability.

Such patterns have to be composed with actions which match one component type. For instance, if a vulnerability pattern is dedicated to Activities, then its action set must be equal to $AuthAct_{Activity}$. Guards can also be composed of specific predicates to ease their writing. In the paper, we consider some predicates such as *in ()* which represents a Boolean function returning true if the parameter 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 5 (Vulnerability Pattern)**.** *A Vulnerability pattern is a deterministic IOSTS suspension* $\mathcal{VP} =< L_{\mathcal{VP}}, l0_{\mathcal{VP}}, V_{\mathcal{VP}}, V0_{\mathcal{VP}}, I_{\mathcal{VP}}, \Lambda_{\mathcal{VP}}, \rightarrow>_{\mathcal{VP}}>$ *such that the final locations of* $L_{\mathcal{VP}}$ *belong to* $\{Vul, NVul\}$. $\Lambda_{\mathcal{VP}} = AuthAct_{type}$ *with type the component type targeted by* $\mathcal{VP}$.

Figure 1 illustrates a straightforward example of vulnerability pattern to test the Availability of Android Activities. It describes that an Activity is unavailable and consequently vulnerable when quiescence is observed or when the Activity crashes, which is observed when an exception is raised by the Android system. The intent action belongs either to the Android action set or in the RV(String) set which stands for String values known for relieving bugs, e.g., "$" or ";" and random values. The data d takes a value either in URI or in $RV(String)$ or in Inj.

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

Definition 6 (Vulnerability Status of an IOSTS)**.** *Let* $\mathcal{S}$ *be an IOSTS,* $\mathcal{VP}$ *be a vulnerability pattern such that* $\mathcal{S}^{\delta}$ *is compatible with* $\mathcal{VP}$. *We define the vulnerability status of* $\mathcal{S}$ *(and of its underlying ioLTS semantics* $[\![\mathcal{S}]\!]$*) over* $\mathcal{VP}$ *with:*

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

4 SECURITY TESTING METHODOLOGY

The steps of our approach can be summarised as follows: we assume having a set of vulnerability patterns modelled with IOSTS suspensions. From an Android project (compiled classes and configuration files), we extract a partial class diagram by introspection. This one lists the components, gives their types and the associations between classes. Furthermore, IOSTS suspensions expressing partial specifications of one component, are extracted from the Manifest file. Intermediate IOSTSs, called vulnerability properties, are then derived from the combination of vulnerability patterns with partial specifications. These properties still express vulnerabilities but are refined with the implicit and explicit intents that a component may accept. Test cases are obtained by concretising vulnerability properties i.e., parameter values are added to obtain executable test cases only. Finally, the latter are translated into JUNIT test cases to be executed with classical development tools. All these steps are detailed below.

4.1 Model Generation

Android applications gather a lot of information that can be used to produce partial models:

1. a simplified class diagram, depicting Android components of the application and their types, is initially computed. The component method and attribute names are established by applying reverse engineering based on Java reflection. This class diagram also gives some informations about the relationships among components. This step aims to later reduce the test case generation. For instance, the verification of data vulnerabilities has to be done on components tied with Content providers (specialized components managing database). This relationship is established when a component has a *ContentResolver* attribute,

2. one partial specification $\mathcal{S}_{ct} = (S1_{ct}, S2_{ct})$ is generated for each component found in the Android application. $S1_{ct}$ is an IOSTS suspension composed of the implicit intents given in the Manifest file. In contrast, $S2_{ct}$ models any (explicit) intent except the implicit ones. This separation shall be particularly useful to distribute the test case set between implicit and explicit intents when the number of test cases is limited.

   Algorithm 1 constructs a partial specification $\mathcal{S}_{ct} = (S1_{ct}, S2_{ct})$ from the intent filters *IntentFilter(act,cat,data)* found in a Manifest file. The action sets of $\Lambda_{Si_{ct}}(i = 1,2)$ are set to $AuthAct_{type(ct)}$ with $type(ct)$ the type of the component, e.g., Activity. For readability, we present the algorithm dedicated to Activities only. It produces two IOSTSs w.r.t. the intent functioning described in the Android documentation. Firstly, Algorithm 1 constructs $S1_{ct}$ with the implicit intents found in the intent filters (lines 6-16). Depending on the action type read, the guard of the output action is completed to reflect the fact that a response may be received or not. If the action of the intent filter is unknown (lines 12,13), no guard is formulated on the output action (a response may be received or not). While the generation of $S1_{ct}$, the algorithm also produces a guard G equals to the negation of the union of guards added with the ?intent action (line 15). Then, $S2_{ct}$ is constructed by means of this guard: it models the sending of an intent with the guard G (intuitively, any intent except the intents of $S1_{ct}$) followed by a transition carrying the action !Display without guard and a transition labelled by !ComponentException.

   Finally, both $S1_{ct}$ and $S2_{ct}$ are completed on the output set to express incorrect behaviours modelled with new transitions to the Fail location, guarded by the negation of the union of guards of the same output action on outgoing transitions (lines 17-20). The new *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 other Android component types, the algorithms are very similar.

Figure 2 illustrates a partial specification example composed of implicit intents ($S1_{ct}$). Two intents are accepted by the component, one composed of the action *VIEW* that is called to display information about the first person in the contact list of the Mobile phone and another action *PICK* which aims to ask the user to choose a contact that is returned to the calling component. Transitions to Fail represent undesired behaviours. For instance, after a PICK action, the response must not be null.

Algorithm 1: Partial Specification Generation.

```
   input  : Manifest file $MF$
   output: Partial specifications $\mathcal{S}_{ct} = (S1_{ct}, S2_{ct})$
1  foreach component ct in MF do
2      $it := 0; G := \emptyset$;
3      $\mathcal{S}_{ct} = (S1_{ct}, S2_{ct})$ is a partial specification of ct /
       $\Lambda_{Si_{ct}}(i = 1,2) = AuthAct_{type(ct)}$;
4      Add $l0_{Si_{ct}} \xrightarrow{!\delta}_{Si_{ct}} l0_{Si_{ct}}$ to $\rightarrow_{Si_{ct}}; (i = 1,2)$
5      if type of ct == Activity then
6          foreach IntentFilter(act,cat,data) of ct in MF do
7              $it := it + 1$ ;
8              if $act \in ACT_r$ then
9                  Add $l0_{S1_{ct}} \xrightarrow{?evt_1^{(1)}}_{S1_{ct}} (l_{it,1}) \xrightarrow{!di_1^{(2)},[ct.resp \neq Null]} l0_{S1_{ct}}$ to $\rightarrow_{S1_{ct}}$
10             else if $act \in ACT_{nr}$ then
11                 Add $l0_{S1_{ct}} \xrightarrow{?evt_1^{(1)}}_{S1_{ct}} (l_{it,1}) \xrightarrow{!di_1^{(2)},[ct.resp = Null]} l0_{S1_{ct}}$ to $\rightarrow_{S1_{ct}}$
12             else
13                 Add $l0_{S1_{ct}} \xrightarrow{?evt_1^{(1)}}_{S1_{ct}} (l_{it,1}) \xrightarrow{!di_1^{(2)}} l0_{S1_{ct}}$ to $\rightarrow_{S1_{ct}}$
14             Add $(l_{it,1}) \xrightarrow{!ComponentExp}_{S1_{ct}} l0_{S1_{ct}}$ to $\rightarrow_{S1_{ct}}$;
15             $G := G \wedge \neg G1$;
16         Add
           $l0_{S2_{ct}} \xrightarrow{?intent(a,d,c,v),G,A=(x:=x)_{x \in V_{S2_{ct}}}}_{S2_{ct}} l_1 \xrightarrow{!di_2^{(2)}} l0_{S2_{ct}}, l_1 \xrightarrow{!ComponentExp}_{S2_{ct}} l0_{S2_{ct}}$ to
           $\rightarrow_{S2_{ct}}$;
17     foreach $l_1 \in L_{Si_{ct}}(1 \leq i \leq 2)$ such that
       $l_1 \xrightarrow{!a,G,A}_{Si_{ct}} l_2$ do
18         foreach $a \in \Lambda^O_{Si_{ct}}$ do
19             $G_a = \bigwedge_{l_1 \xrightarrow{a(p),G,A}_{Si_{ct}} l} \neg G$;
20         Add $l_1 \xrightarrow{?a(p),G_a,A_a=(x:=x)_{x \in V}}_{Si_{ct}} Fail$ to $\rightarrow_{Si_{ct}}$
21 (1) $?intent(a,d,c,v), G1 = [a = act \wedge d = data \wedge c = cat], A = (x := x)_{x \in V_{S1_{ct}}}$
   (2)$!Display(Activity\ ct), A = (x := x)_{x \in V_{Si_{ct}}}$
```

4.2 Test Case Selection

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 perform the test case execution:

- the functional behaviours of the component under test, observed while testing, can be modelled

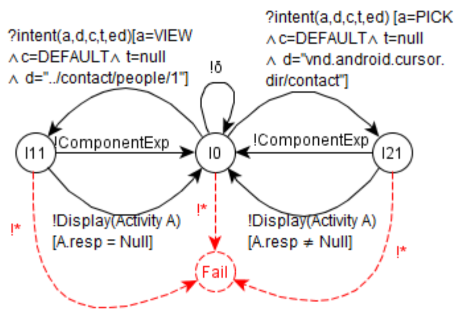

Figure 2: An Activity specification.

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). CUT^{δ} denotes its ioLTS suspension,

- to be able to dialog with CUT, one assumes that CUT is a component whose type is the same as the component type targeted by the vulnerability pattern $\mathcal{VP}$ and that it is compatible with $\mathcal{VP}$.

Test cases stem from the composition of vulnerability patterns with compatible partial specifications. Given a vulnerability pattern $\mathcal{VP}$ and a partial specification $\mathcal{S}_{ct} = (S1_{ct}, S2_{ct})$, the composition $V(\mathcal{S}_{ct}) = (\mathcal{VP}||S1_{ct}, \mathcal{VP}||S2_{ct})$ is called a vulnerability property of $\mathcal{S}_{ct}$. It represents the vulnerable and non-vulnerable behaviours which may be observed from the component with implicit or explicit intents. The parallel compositions $(\mathcal{VP}||Si_{ct})(i = 1,2)$ produce new locations and in particular new final verdict locations:

Definition 7 (Verdict location sets). *Let $\mathcal{VP}$ be a vulnerability pattern and $\mathcal{S}_{ct} = (S1_{ct}, S2_{ct})$ a partial specification with $Si_{ct}(i = 1,2)$ compatible with $\mathcal{VP}$. $(\mathcal{VP}||Si_{ct})(i = 1,2)$ are composed of new locations recognising vulnerability status:*

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

Test cases are achieved with Algorithm 2 which performs the two following main steps on each IOSTS of the vulnerability property $V(\mathcal{S}_{ct}) = (\mathcal{VP}||S1_{ct}, \mathcal{VP}||S2_{ct})$. Firstly, it splits $(\mathcal{VP}||Si_{ct})$ 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 2-5).

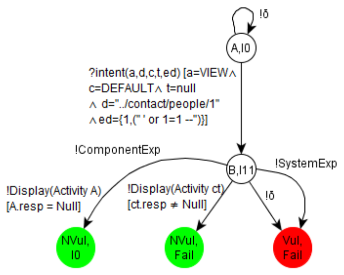

Figure 3: A test case example.

Then, a set tuple of valuations is computed for the list of undefined parameters of the input action (line 6). Instead of using a cartesian product to construct a tuple of valuations, we adopted a Pairwise technique (Cohen et al., 2003). Assuming that errors can be revealed by modifying pairs of variables, 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. If no domain is found, the RV set is used instead. In the second step (line 7-14), 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 into the test case set $TC_{(\mathcal{VP}||Si_{ct})}$. Steps 1. and 2. are iteratively applied until each combination of transitions labelled by input actions and each combination of valuation tuples are covered. Since the algorithm may produce a large set of test cases, the algorithm also ends when the test case set $TC_{(\mathcal{VP}||Si_{ct})}$ reaches a cardinality of $tcnb$ (lines 18,19). This condition limits the test case number but also allows balancing the generation of test cases build with implicit intents (those obtained from $S1_{ct}$) with the test cases executing explicit intents (obtained from $S2_{ct}$).

A test case example is depicted in Figure 3. It originates from the IOSTS suspension $S1_{ct}$ of Figure 2 and expresses the sending of an intent with the extra data part composed of an SQL injection. In other terms, it illustrates the call of the component under test with a malicious intent composed of the classical SQL injection *"'or 1=1–"*. Other test cases are also generated from $S2_{ct}$ to send intents composed of malicious actions, categories, etc.

The test cases, constructed with Algorithm 2, are composed of paths of a vulnerability property, starting from its initial locations and whose transitions are

Algorithm 2: Test case generation.

```
input  : A vulnerability property $V(\mathcal{S}_{ct})$, $tcnb$ the
         maximal number of test cases
output: Test case set $TC$
foreach $(\mathcal{VP}||Si_{ct})(i=1,2) \in V(\mathcal{S}_{ct})$ do
    begin 1. input action choice
        foreach location $l$ having outgoing
        transitions carrying input actions do
            Choose a transition
            $t = l \xrightarrow{?a(p),G,A}_{(\mathcal{VP}||Si_{ct})} l_2$;
            remove the other transitions labelled by
            input actions;
            $P(t) = Pairwise(p_1,...,p_n,G)$ with
            $(p_1,...,p_n) \subseteq p$ the list of undefined
            parameters;
    begin 2. input concretisation
        foreach $t = l \xrightarrow{?a(p),G,A}_{(\mathcal{VP}||Si_{ct})} l_2$ do
            Choose a valuation tuple
            $pv = (p_1 = v_1,...,p_n = v_n)$ in $P(t)$;
            if $G \wedge pv$ is satisfiable then
                Replace $G$ by $G \wedge pv$ in $t$;
            else
                Choose another valuation tuple in
                $P(t)$;
        $tc$ is the resulting IOSTS suspension;
    begin 3.
        if $tc$ has reachable verdict locations then
            $TC_{(\mathcal{VP}||Si_{ct})} := TC_{(\mathcal{VP}||Si_{ct})} \cup \{tc\}$ ;
        if $Card(TC_{(\mathcal{VP}||Si_{ct})}) \geq tcnb$ then
            STOP;
        Repeat 1. and 2. until each combination of
        transitions carrying input actions and each
        combination of valuation tuples are covered;
$TC = \bigcup_{i=1,2} TC_{(\mathcal{VP}||Si_{ct})}$;
```

concretised with values that meet the original guards. In other words, the test selection algorithm does not add new traces leading to verdict locations. Hence, one can deduce that the test case traces belong to the trace set of the vulnerability property:

Proposition 8. *Let $V(\mathcal{S}_{ct})$ be a vulnerability property derived from the composition of a vulnerability pattern $\mathcal{VP}$ and a partial specification $\mathcal{S}_{ct} = (S1_{ct}, S2_{ct})$. TC is the test case set generated by Algorithm 2. We have $\forall tc \in TC, Traces(tc) \subseteq (Traces(\mathcal{VP}||S1_{ct}) \cup Traces(\mathcal{VP}||S2_{ct}))$.*

4.3 Test Case Execution Definition

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

Proposition 9 (Test case execution)**.** *Let TC be a test case set obtained from the vulnerability pattern $\mathcal{VP}$. CUT is the ioLTS of the component under test, assumed compatible with $\mathcal{VP}$. For all test case $tc \in TC$, the execution of tc on CUT is defined by the parallel composition $tc||CUT^{\delta}$.*

The above proposition leads to the test verdict of a component under test against a vulnerability pattern $\mathcal{VP}$. Intuitively, this one refers to the Vulnerability status definition, completed by the detection of incorrect behaviours described in the partial specification of the component 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 10 (Test verdict)**.** *We take back the notations of Proposition 9. The execution of the test case set TC on CUT yields one of the following verdicts:*

- *CUT is vulnerable to $\mathcal{VP}$ 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,*
- *CUT is not vulnerable to $\mathcal{VP}$ 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 $\sigma \notin Traces_{VUL}(tc) \cup Traces_{NVUL}(tc)$.*

The above security testing method has been implemented in a tool called *APSET* (Android aPplications SEcurity Testing), publicly available in a Github repository [1]. It takes an Android application project (uncompressed APK) 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. The guard solving, used in Algorithm 2 and during the test case execution, is performed by the SMT (Satisfiability Modulo Theories) solver Z3 [2], whose language is augmented with the predicates given in Section 3.

We experimented several real Android applications provided by the Openium company [3]. Table **??** summarises the results obtained on 10 applications with two vulnerability patterns: $\mathcal{VP}1$ is the one of Figure 1, $\mathcal{VP}2$ is a vulnerability pattern dedicated to

[1]https://github.com/statops/apset.git
[2]http://z3.codeplex.com/
[3]http://www.openium.fr/

Table 1: Experimentation Results.

Applications		$\mathcal{VP}$1 test results		$\mathcal{VP}$2 test results	
App	# component	Time/ test	*#vul*/ *#testcases*	Time/ test	*#vul*/# *testcases*
app 1	35	8s	861/969	0.7s	0/175
app 2	6	12s	95/147	0.25s	7/60
app 3	5	4s	0/117	-	-
app 4	24	0.15s	52/545	-	-
app 5	11	2s	3/33	0.175s	7/77
app 6	11	3s	11/120	-	-
app 7	11	3s	20/110	-	-
app 8	11	3s	20/110	-	-
app 9	13	0.90s	19/80	-	-
app 10	15	2.1s	15/105	1.6s	31/105

integrity testing. Intuitively, this one aims at checking whether stored data can be modified with malicious intents: initially, a set of structured data, managed by a Content Provider, are stored. Then, all the components (Service or Activity) composed with this Content provider, are called with malicious intents composed of SQL and XML injections. Finally, the Content Provider state is requested to check if it has been modified without having any user or administrator credentials.

For each application and each vulnerability pattern, we provide, the number of tested components, the average test case execution time delay, and the number of vulnerability issues detected over the test case number. With $\mathcal{VP}$1, all the tested applications revealed vulnerability issues. For instance, 969 test cases were generated by our tool for *app 1* and 861 revealed issues. Obviously, several vulnerable verdicts were obtained on account of the same vulnerability in the component code. All these issues were essentially observed by component crashes when receiving malicious intents (receipt of exceptions such as *NullPointerException*). With $\mathcal{VP}$2, tests were only applied on the applications whose generated class diagrams reveal at least one Content Provider component (applications 1, 2, 5 and 10). We detected some data integrity issues with *app 5* and *app 10*. In particular, the test reports showed that the modifications of data were detected with *app 5* and the deletion of data with *app 10* without providing login credentials with the intents.

5 CONCLUSIONS

We have presented a security testing method of Android applications for testing whether components are vulnerable to malicious intents. The originality of this work resides in the intent mechanism security testing first, but also in the automatic generation of partial specifications from Android Manifest files. These specifications are used to generate test cases composed of either implicit or explicit intents. They also contribute to complete the test verdict with the specific 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 perform other experimentations with further vulnerability patterns based on the Authorisation concept.

REFERENCES

Android, D. (2013). Android developer's guide. In *http://developer.android.com/index.html, last accessed feb 2013*.

Chin, E., Felt, A. P., Greenwood, K., and Wagner, D. (2011). Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 239–252.

Cohen, M. B., Gibbons, P. B., Mugridge, W. B., and Colbourn, C. J. (2003). Constructing test suites for interaction testing. In *Proc. of the 25th International Conference on Software Engineering*, pages 38–48.

Frantzen, L., Tretmans, J., and Willemse, T. (2005). Test Generation Based on Symbolic Specifications. In Grabowski, J. and Nielsen, B., editors, *FATES 2004*, number 3395 in Lecture Notes in Computer Science, pages 1–15. Springer.

Jing, Y., Ahn, G.-J., and Hu, H. (2012). Model-based conformance testing for android. In Hanaoka, G. and Yamauchi, T., editors, *Proceedings of the 7th International Workshop on Security (IWSEC)*, volume 7631 of *Lecture Notes in Computer Science*, pages 1–18. Springer.

Report (2012). It business: Android security. In *http://www.itbusinessedge.com/cm/blogs/weinschenk/google-must-deal-with-android-security-problems-quickly/?cs=49291, , last accessed feb 2013*.

Zhong, J., Huang, J., and Liang, B. (2012). Android permission re-delegation detection and test case generation. In *Computer Science Service System (CSSS), 2012 International Conference on*, pages 871 –874.