INTENT SECURITY TESTING: AN APPROACH TO TESTING THE INTENT-BASED VULNERABILITY OF ANDROID COMPONENTS.

Sébastien Salva[1], Stassia R. Zafimiharisoa [1] and Patrice Laurençot[1]

[1] *LIMOS - UMR CNRS 6158, PRES Clermont-Ferrand University, FRANCE, France*

sebastien.salva@udamail.fr, s.zafimiharisoa@openium.fr, laurencot@isima.fr

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. We also enrich the Android test platform to refine the flaw detection. 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 analysis in Mobile threats. These reports accentuate the idea that Mobile security continues to be a global issue, independently of the platform. As the perfect mobile security product does not (and will probably never) exist, security testing represents the most valuable solution to detect vulnerability flaws in Mobile systems and applications. The latter are experimented with test cases, during the testing process, usually constructed by hands from known attacks or vulnerabilities. Model-based testing, which is the topic of the paper, is another approach which 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 threats 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, originally discovered by (Chin et al., 2011a), stems from the Android application structure: these applications consist of one or more core components glued together by means of the intent concept, which is an IPC (Inter Process 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 Availability or Integrity-based attacks. So considered, the intent mechanism becomes an attack vector

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 vulnerability patterns and test verdict without ambiguity. Then, from vulnerability patterns, our method performs both the automatic test case generation and execution.

Contribution: our first contribution 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 (type, link to other components) and represent the functional behaviours that should be observed from each compo-

nent after the receipt of an intent (in reference to the Android documentation (AD, 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. These test cases are finally translated into JUNIT test cases to ease their executions.

Then, we introduce a test execution framework that offers a better flaw detection than the Android tools. This one supports the distinction of two class of exceptions: those raised by the Android system when applications crash only and the other ones. This distinction is also taken into account in the vulnerability pattern modelling to augment the vulnerability expressiveness.

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. The testing methodology is described in Section 4. We describe the implementation of the method in a tool called *APSET* and give experimentation results in Section 5. Finally, Section 6 compares our approach with some related works and we conclude in Section 7.

2 MODEL DEFINITION AND NOTATIONS

We shall consider the input/output Symbolic Transition Systems (ioSTS) model to generate partial specifications of Android components and to express vulnerabilities. The use of this formal model offers the advantage to define without ambiguity test verdicts and to define ioSTS operations in an algebraic manner instead of providing complex algorithms.

An ioSTS is a kind of automata model which is extended with two set 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. Inputs of a system can only interact with outputs provided by the system environment and vice-versa. 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 $\mathcal{S}^{\delta}$, which is obtained by adding a self-loop labelled by $!\delta$ for each location where no output action may be observed. More details about ioSTSs can be found in (Frantzen et al., 2005).

Definition 1 (ioSTS suspension)**.** *A deterministic ioSTS suspension is a tuple* $< L, l0, V, V0, I, \Lambda, \rightarrow >$, *where:*

- *L is the finite set of locations, with $l0$ the initial one,*
- *V is the finite set of internal variables, while 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 set of parameter variables 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 all 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, an ioLTS semantics corresponds to a valued automaton: the ioLTS states are labelled by internal variable valuations while transitions are labelled by actions and parameter valuations.

Definition 2 (ioLTS semantics)**.** *The semantics of an ioSTS* $\mathcal{S} = < L, l0, V, V0, I, \Lambda, \rightarrow >$ *is the ioLTS* $[\![\mathcal{S}]\!] = < Q, q0, \sum, \rightarrow >$ *where:*

- *$Q = L \times D_V$ is the set of states,*
- *$q0 = (l0, V0)$ is the initial state,*
- *$\sum = \{(a(p), \theta) \mid a(p) \in \Lambda, \theta \in D_p\}$ is the set of valued actions. $\sum^I$ is the set of input actions and $\sum^O$ is the set of output ones,*
- *$\rightarrow$ is the transition relation $Q \times \Sigma \times Q$ deduced by the following rule:*

$$\frac{l_1 \xrightarrow{a(p),G,A} l_2, \theta \in D_p, v \in D_V, v' \in D_V, v \cup \theta \models G, v' = A(v \cup \theta))}{(l_1, v) \xrightarrow{a(p),\theta} (l_2, v')}$$

Remark 3. *An ioSTS suspension $\mathcal{S}^{\delta}$ is also associated to its ioLTS semantics suspension by $[\![\mathcal{S}^{\delta}]\!] = [\![\mathcal{S}]\!]^{\delta}$.*

All the paths and action sequences of an ioLTS semantics represent runs and traces of a system.

Definition 4 (Runs and traces). *For an ioSTS $\mathcal{S} = < L, l0, V, V0, I, \Lambda, \rightarrow >$, interpreted by its ioLTS semantics $[\![\mathcal{S}]\!] =< Q, q0, \sum, \rightarrow >$, a run $q0\alpha_0...\alpha_{n-1}q_n$ 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 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 the ioSTS $\mathcal{S}_1 =< L_1, l0_1, V_1, V0_1, I_1, \Lambda_1, \rightarrow_1 >$ with the compatible ioSTS $\mathcal{S}_2 = < L_2, l0_2, V_2, V0_2, I_2, \Lambda_2, \rightarrow_2 >$, 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 rule:*

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

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

Notation	Meaning
$?intent(a,d,c,t,ed)$	an intent composed of: action a, data d, action category c, data type t, extra data ed
ACT_r	set of actions which require a response
ACT_{nr}	set of actions which do not require a response
C	set of categories
T	set of types
URI	set of URI
RV	set of predefined and random values
Inj	set of SQL and XML injections
!Display	display of a screen by an Activity
!ComponentExp	Exception raised by a component
!SystemExp	Exception raised by the system

Table 1: Android component notations

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. These notations are also given in Table 1. 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 $intent(a,d,c,t,ed)$ is a kind of bundle of information which 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 which represent additional data (AD, 2013). 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 by means of a list of *intent filters*. Both can be exploited by a malicious application to send attacks to components at runtime: sending implicit malicious intents is easier than sending explicit ones though since the targeted component has to be known for the latter. But, because extracting a component list in an Android system is possible, 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 (AD, 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 $AuthAct_{Activity} = \{?intent(a,d,c,t,ed), !Display(A), !SystemExp, !ComponentExp\}$.

3.2 Vulnerability Patterns

Several formalisms have been proposed to represent vulnerabilities e.g., regular expressions, temporal and deontic logics, core typed languages ((Chaudhuri, 2009)) or state machines. We chose the latter because it sounds more 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 firstly 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* which aim to recognise the vulnerability status over component executions: runs of a vulnerability pattern starting from the initial location and ended by *Vul* express functional behaviours com-

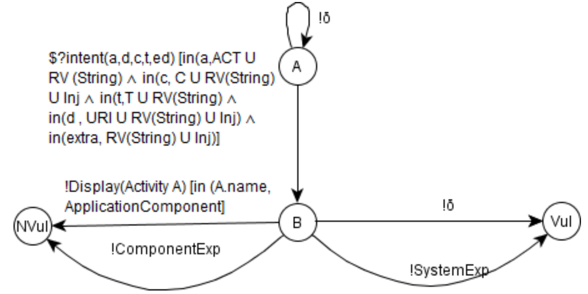

Figure 1: Test pattern for availability testing

posed of malicious intents which 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 matches 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, and Inj is a set gathering XML and SQL injections. *URI* is a set of URI constructed randomly which start by *file:* or *content:* or *https:* and whose paths, which indicate the data location, are constructed over the *RV* set. New sets can be also added in condition that real value sets which the same name would be added to the testing tool.

Definition 7 (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. The intent action must belong 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. This vulnerability pattern means 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. More complex patterns, composed of several input and output actions, can be defined.

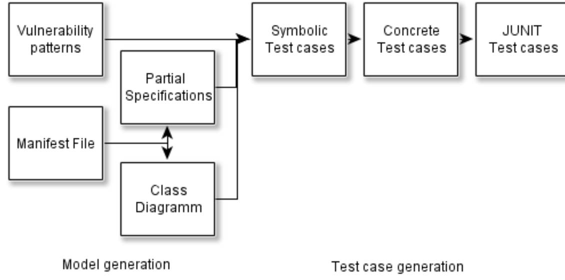

Figure 2: Test case generation

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 8 (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 different steps of our approach are illustrated in Figure 2 and 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. 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 computed and the unreachable transitions are pruned 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

Experimenting directly Android components with vulnerability patterns would lead to several issues. For example, an Activity which displays a screen on a Smartphone has a different purpose than a service component which does not interact directly with a user. Performing blind testing without considering the component features would often lead to false negative results. Besides, Android applications gather plentiful of information that can be used to produce partial models in order to refine the test case generation:

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 information about the relationships between components. This step aims to later reduce the test case generation. For instance, the verification of data Integrity has to be done on components which interact with Content-provider components only. 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 while $S2_{ct}$ 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 set of $\Lambda_{Si_{ct}} (i = 1, 2)$ is equal 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. For instance, the call of an intent composed of a PICK action, in ACT_r, implies both the display of a screen and the receipt of a response. 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 call 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 guards 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 3 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.

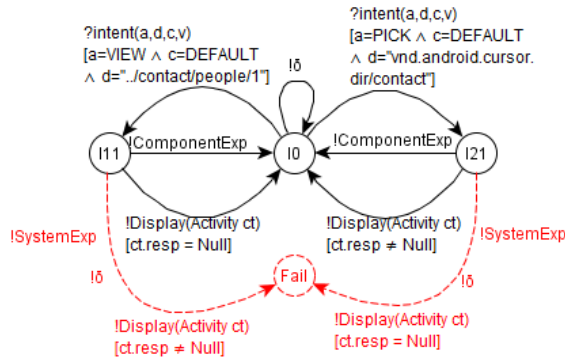

Figure 3: An Activity specification

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 $\quad it := 0; G := \emptyset$;
3 $\quad \mathcal{S}_{ct} = (S1_{ct}, S2_{ct})$ is a partial specification of ct / $\Lambda_{Si_{ct}} (i = 1,2) = AuthAct_{type(ct)}$;
4 $\quad$ Add $l0_{Si_{ct}} \xrightarrow{!\delta}_{Si_{ct}} l0_{Si_{ct}}$ to $\rightarrow_{Si_{ct}}; (i = 1,2)$
5 $\quad$ **if** *type of ct == Activity* **then**
6 $\quad\quad$ **foreach** *IntentFilter(act,cat,data) of ct in MF* **do**
7 $\quad\quad\quad it := it + 1$;
8 $\quad\quad\quad$ **if** $act \in ACT_r$ **then**
9 $\quad\quad\quad\quad$ 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 $\quad\quad\quad$ **else if** $act \in ACT_{nr}$ **then**
11 $\quad\quad\quad\quad$ 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 $\quad\quad\quad$ **else**
13 $\quad\quad\quad\quad$ 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 $\quad\quad\quad$ Add $(l_{it},1) \xrightarrow{!CompExp}_{S1_{ct}} l0_{S1_{ct}}$ to $\rightarrow_{S1_{ct}}$;
15 $\quad\quad\quad G := G \wedge \neg G1$;
16 $\quad\quad$ Add $l0_{S2_{ct}} \xrightarrow{?intent(a,d,c,v),G,A=(x:=x)_{x \in V_{S2_{ct}}}}_{S2_{ct}} l_1$ $\xrightarrow{!di_2^{(2)}} l0_{S2_{ct}}, l_1 \xrightarrow{!CompExp}_{S2_{ct}} l0_{S2_{ct}}$ to $\rightarrow_{S2_{ct}}$;
17 $\quad$ **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 $\quad\quad$ **foreach** $a \in \Lambda^O_{Si_{ct}}$ **do**
19 $\quad\quad\quad G_a = \bigwedge_{l_1 \xrightarrow{a(p),G,A}_{Si_{ct}} l} \neg G$;
20 $\quad\quad$ 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 and to formalise the confidence level of the test of CUT w.r.t. a vulnerability pattern $\mathcal{VP}$:

- 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). 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 9 (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. $\mathcal{VP}$.*

From a vulnerability property $V(\mathcal{S}_{ct}) = (\mathcal{VP}||S1_{ct}, \mathcal{VP}||S2_{ct})$, test cases are extracted by splitting the parallel compositions into several test cases and by concretising them with values. Intuitively, if n transitions carrying an intent can be fired from a location l of $(\mathcal{VP}||Si_{ct})(i = 1,2)$, then n test cases are constructed to experiment CUT with the n intents and so on for each next location reachable from the location l. While splitting $(\mathcal{VP}||Si_{ct})(i = 1,2)$, actions have to be concretised with set of values to make concrete test cases. This is particularly the case for the $intent(a,d,c,t,ed)$ actions composed of several variables. In the paper, we adopt a Pairwise technique to construct sets of tuples of values (Cohen et al., 2003) instead of using a cartesian product. 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. These steps are performed with Algorithm 2 which covers recursively, in a finite number of times, the locations of $(\mathcal{VP}||Si_{ct})(i = 1,2)$ while splitting/concretising transitions.

Algorithm 2 achieves the test case generation by covering each ioSTS suspension $(\mathcal{VP}||Si_{ct})(i = 1,2)$, from its starting location with the *Cover* function which takes a location, a set of internal variable valuations and a test case being constructed. When the current location has outgoing transitions $l \xrightarrow{a(p),G,A} l_2$ labelled by an output action, these ones are added to the current test case (lines 4,5). Then, the algorithm continues to cover the ioSTS from each arrival location l_2 of these transitions upon condition that the guard G has a solution. This is checked by the call of a *Solver* function (lines 6-8). Otherwise the next transitions reachable from l_2 are implicitly pruned and are not added to the test case. In the second part of the algorithm, if the current location has outgoing transitions $l \xrightarrow{a(p),G,A} l_2$ labelled by an input action (line 9), the current test case is stored into tc' to construct several new test cases originating from tc'. For each transition t carrying an input action and for each tuple of values $p = (p_1 = v_1, ..., p_n = v_n)$ computed with the *Pairwise* function (lines 11-13), the algorithm checks whether the guard G evaluates to true with p and with the current internal valuations V, before adding t, concretised with p, to the test case tc. The arrival location l_2 is covered recursively. Once all the transitions accessible from l_2 are covered, the final test case tc is added to the set TC (line 21). The algorithm ends when all the transitions labelled by an input action are covered at least on time and concretised by all the values computed by the Pairwise function. Since the algorithm may produce a large set of test cases, depending on the number of tuple of values given by the Pairwise function, the algorithm also ends when the test case set $TC_{Si_{ct}}$ reaches a cardinality of $tcnb$ (lines 20, 21). This condition limits the test case number but also allows to balance the generation of test cases executing 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 4. It originates from the ioSTS suspension $S1_{ct}$ of Figure 3 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 complete paths of a vulnerability property, starting from its initial locations and concretised with values that meet the original guards. Hence, one can deduce that the test case traces belong to the trace set of the vulnerability property:

Proposition 10. *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 from $V(\mathcal{S}_{ct})$. We have $\forall tc \in TC$, $Traces(tc) \subseteq (Traces(\mathcal{VP}||S1_{ct}) \cup Traces(\mathcal{VP}||S2_{ct}))$.*

Algorithm 2: Test case generation

input : A vulnerability property $V(S_{ct})$, *tcnb* the maximal number of test cases per ioSTS
output: Test case set TC

1 Cover(location l,Variable set V, test case tc)
2 BEGIN
3 **if** *l has outgoing transitions labelled by output actions* **then**
4   **foreach** $t = l \xrightarrow{a(p),G,A}_{Si_{ct}} l_2$ **do**
5   Add t to $\rightarrow_{tc}$;
6   **foreach** $t = l \xrightarrow{a(p),G,A}_{Si_{ct}} l_2$ **do**
7   **if** *$Solve(G,V)$ returns a non-empty solution* $(p_1 = v_1, ..., p_n = v_n)$ **then**
8   Cover(l_2,$A(V)$,tc);
9 **if** *l has outgoing transitions labelled by input actions* **then**
10   $tc' := tc$;
11   **foreach** $l \xrightarrow{a(p),G,A}_{Si_{ct}} l_2$ *with* $p = (p_1, ..., p_n)$ **do**
12     $P = pairwise((p_1, ..., p_n), G)$;
13   **foreach** $(p_1 = v_1, ..., p_n = v_n) \in P$ **do**
14       $tc := tc'$;
15   **if** *$Solve(G, V \cup \{(p_1 = v_1, ..., p_n = v_n)\}$=true)* **then**
16   Add $l \xrightarrow{a(p),G \wedge (p_1=v_1,...,p_n=v_n),A} l_2$ to $\rightarrow_{tc}$;
17   Cover(l_2,$A(V)$,tc);
18   **if** $Card(TC_{Si_{ct}}) > tcnb$ **then**
19   STOP;
20   **else**
21           $TC_{Si_{ct}} := TC_{Si_{ct}} \cup \{tc\}$;
22 END
23 reset tc ;
24 Cover($l0_{S1_{ct}}$, $V0_{S1_{ct}}$, tc);
25 reset tc;
26 Cover($l0_{S2_{ct}}$, $V0_{S2_{ct}}$, tc);
27 $TC = \bigcup_{i=1,2} TC_{Si_{ct}}$;

4.3 Test case execution definition

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

Proposition 11 (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 $[\![tc]\!]||CUT^{\delta}$.*

Remark 12. *A test case tc obtained from a vulnerability pattern $\mathcal{VP}$, can be experimented on*

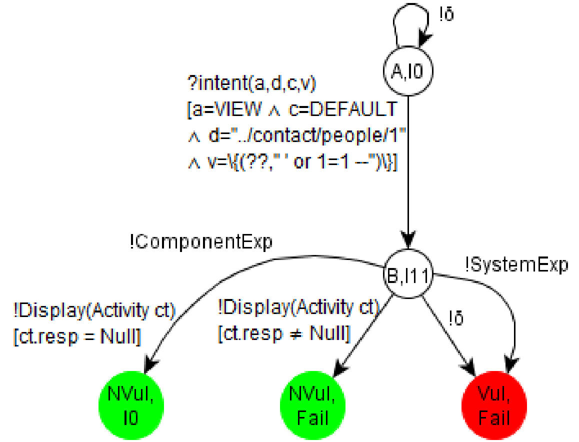

Figure 4: A test case example

CUT since tc and $\mathcal{VP}$ are compatible. Indeed, tc is produced from a vulnerability property $V(\mathcal{S}_{ct}) = (\mathcal{VP}||S1_{ct}, \mathcal{VP}||S2_{ct})$ such that $\Lambda_{Si_{ct}}(i = 1,2) = AuthAct_{type(ct)}$. The action set of $\mathcal{VP}$ is also equal to $AuthAct_{type(ct)}$. We deduce that tc, $Si_{ct}(i = 1,2)$ and $\mathcal{VP}$ are compatible.

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 incorrect behaviours described in the partial specification of the component. Indeed, the verdict locations *VUL/FAIL* and *NVUL/FAIL* augment the expressiveness of the vulnerability status by pointing out that CUT does not also respect the component normal functioning w.r.t. the Android documentation:

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 $\mathcal{VP}$ 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.*

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)
$(Traces_{VUL}(\mathcal{VP}||S1_{ct}) \cup Traces_{VUL}(\mathcal{VP}||S2_{ct})) \cap Traces(CUT^{\delta}) \neq \emptyset$ (Proposition 10)
$Traces_{VUL}(\mathcal{VP}||Si_{ct}) = Traces_{Vul}(\mathcal{VP}) \cap$

$Traces_{L_{Si_{ct}}}(Si_{ct})$ since Si_{ct} is compatible with $\mathcal{VP}$ (Algorithm 1 and Lemma 1)
We have $(Traces_{Vul}(\mathcal{VP}) \cap (Traces_{L_{S1_{ct}}}(S1_{ct}) \cup Traces_{L_{S1_{ct}}}(S2_{ct}))) \cap Traces(CUT^{\delta}) \neq \emptyset$. Hence, $Traces_{Vul}(\mathcal{VP}) \cap Traces(CUT^{\delta}) \neq \emptyset$ (a) and $(Traces_{L_{S1_{ct}}}(S1_{ct}) \cup Traces_{L_{S1_{ct}}}(S2_{ct})) \cap Traces(CUT^{\delta}) \neq \emptyset$ (b). From (a), we obtain $CUT \not\models \mathcal{VP}$ (Definition 7). Consequently, CUT is vulnerable to $\mathcal{VP}$.
If $\sigma \in Traces_{VUL/FAIL}(tc)$ then, from (b) we have $(Traces_{Fail}(S1_{ct}) \cup Traces_{Fail}(S2_{ct})) \cap Traces(CUT^{\delta}) \neq \emptyset$. σ represents an incorrect behaviour of the partial specification $\mathcal{S}_{ct} = (S1_{ct}, S2_{ct})$. □

In practice, the parallel composition of a test case *tc* with a component under test *CUT* is done with the testing framework detailed in the next Section.

5 IMPLEMENTATION AND EXPERIMENTATION

5.1 Implementation

The above security testing method has been implemented in a prototype tool called *APSET* (Android aPplications SEcurity Testing), publicly available in a Github repository [1]. It takes as inputs vulnerability patterns written in dot format [2] and an Android application project. Then, it generates JUNIT test cases and executes them on Android emulators or devices.

The guard solving in Algorithm 2 is performed by the SMT (Satisfiability Modulo Theories) solver Z3 [3] that we have chosen since it allows a direct use of arithmetic formulae. However, it does not support String variables. So, we extended the Z3 expression language with new predicates, and in particular with String-based predicates (in, streq, contains, etc.). A predicate stands for a function over ioSTS internal variables and parameters which returns a Boolean. Basically, our tool takes Z3 expressions enriched with predicates, the latter are evaluated and replaced with Boolean values. Then, a Z3 script, composed of internal variables valuations, parameter valuations and a guard, is dynamically written before calling Z3. If the guard is satisfiable (not satisfiable), Z3 returns *sat* (*unsat* respectively). Z3 returns *unknown* when the guard satisfiability is undecidable.

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

For instance, the guard [a=VIEW ∧ c=DEFAULT∧ d="../contact/people/1"∧ v={" ' or 1=1 –")}] of the test case depicted in Figure 4 is written with *(and (streq(a,"VIEW") true) (streq(c,"DEFAULT") true) (streq(d, "../contact/people/1") true) streq(v," ' or 1=1 –")*. This Solve procedure can be upgraded easily with any new predicate.

```
1  //intent setting
   mIntent.setAction(android.intent.action.VIEW) ;
3  mIntent.addCategory(android.intent.category.
       DEFAULT);
   mIntent.setData(Uri.parse(content://contacts/
       people/1));
5  mIntent.putExtra(string_key_value1 ,?? ) ;
   mIntent.putExtra(string_key_value2 ,"' or 1=1--")
       ;
7  setActivityIntent(mIntent) ; try {
   //Call of the Activity with the defined intent
9  mActivity=getActivity();
   //test the output quiescence
11 assertNotNull(VULNERABLE, mActivity);
   //test if the Activity is displayed
13  assertTrue(VULNERABLE, Display());
   //test of the response
15 assertTrue(response.getResultData() == null);
   //Component exceptions.
17 } catch(Exception Ex){
       assertTrue(true);
19 } }
```

Figure 5: A JUNIT test case

IoSTS test cases are converted into JUNIT test cases in order to be executed with a *test runner* (set of control methods to run tests). The test case actions are successively converted into Java code. As an example, Figure 5 gives the JUNIT test case derived from the ioSTS of Figure 4. This conversion can be summarised by the following steps:

- a transition $l \xrightarrow{?intent(a,d,c,t,ed),G,A} l2$ is converted into the sending of an intent composed of parameter values given by the guard G,
- the transitions carrying output actions which correspond to both observations and verdicts are translated into Java code and JUNIT assertions. The VUL verdict is provided by the VULNERABLE message in the assertion codes. The NVUL verdict is implicitly obtained if no VULNERABLE message is produced while testing. The Fail verdict is obtained each time an assertion fails. For an output action $!a(p)$ which does not correspond to the raise of an exception, we assume having a corresponding function a() used to complete the assertion. For instance, consider the transition $(B, l11) \xrightarrow{!Display(Activity\ act)[ct.resp \neq NULL]}$

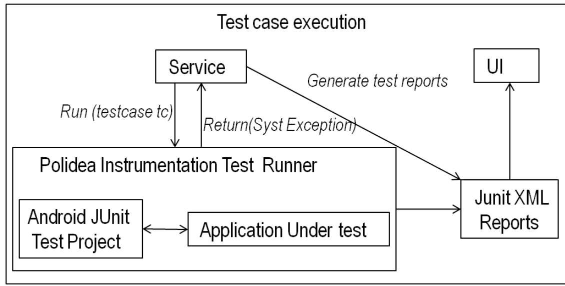

Figure 6: Test case execution framework

$(NVul, Fail)$ of the test case of Figure 4. The action is translated by the two assertions in lines (13,15). The first assertion fails with the VULNERABLE message if Display() returns false. The next assertion fails if a response is provided whereas no response is expected. In this case, the verdict becomes VUL/FAIL.

The output action *!ComponentExp*, is converted into a *try/catch* statement. With our test case example, the catch block is composed of an assertion always true (line 17),

- the transitions $l \xrightarrow{!SystemExp} lf$, modelling an exception raised by the Android system are not converted into JUNIT code but are managed directly with the call of a test runner. If it catches an exception, the latter is converted into an assertion composed of the message VULNERABLE if the location lf is labelled by "VUL".

The test execution framework, depicted in Figure 6 is composed of the Android testing execution tool provided by Google, enriched with the tool *PolideaInstrumentation* [4] to yield XML reports. Test cases are executed on Android devices or emulators by an Android Service component which returns an XML report displayed directly on the device (external computers are not required during this step). The test runner starts the CUT and executes iteratively JUNIT test cases in separate processes. This procedure is required to catch the exceptions raised by the Android system when a component crashes. Once all the test cases are executed, the XML report gathers all the assertion results. In particular, the *VULNERABLE* messages exhibit the detection of a vulnerability issue. Table 2 depicts the number of occurrence of these messages in XML reports. These results can be also used with continuous integration servers like Jenkins [5].

The following example illustrates a part of XML report expressing the crash of a component. We obtain a VULNERABLE message inside a failure XML block. Hence, the verdict is VUL/FAIL.

[4] www.polidea.pl/

[5] http://jenkins-ci.org/

Applications		Availability test results		Integrity 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

Table 2: Experimentation Results

Listing 1: An XML test report

```
1 <testsuite errors="0" failures="1" name="
      packagename.test.Intent.ContactActivityTest
      " package="packagename.test.Intent" tests="
      1" time="0.15" timestamp="2013-02-13
      T10:05:02">
    <testcase classname="packagename.test.Intent.
        ContactActivityTest" name="test1" time="
        0.15">
3       <failure> VULNERABLE
        INSTRUMENTATION_RESULT: shortMsg=java.lang
            .NullPointerException
5       INSTRUMENTATION_RESULT: longMsg=java.lang.
            NullPointerException
        INSTRUMENTATION_CODE: 0
7       </failure>
    </testcase>
9 </testsuite>
```

5.2 Experimentation

We experimented several real Android applications provided by the Openium company [6]. Table 2 summarises the results obtained on 10 applications with two vulnerability patterns, the one of Figure 1 and a vulnerability dedicated to integrity testing. Basically, 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* component, 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,

[6] http://www.openium.fr/

the average test case execution time delay, and the number of vulnerability issues detected over the test case number.

With the availability pattern, 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*).

For the second vulnerability pattern, tests were applied only on the applications whose generated class diagrams reveal at least one Content provider component (applications 1, 2, 5 and 10). We detected data Integrity issues with *app 5* and *app 10*. In particular, test reports show that data modifications were detected with *app 5* and data deletion with *app 10* without providing login credentials with the intents.

Table 2 also gives the average test case execution time measured with Mid 2011 computer with a CPU 2.1Ghz Core i5 and 4Gb of RAM. Each test case execution took few seconds for most of them. Some required longer time processing than others though (some milliseconds up to 12s). This difference comes from the application code. For instance, for *app 1*, some Activities perform several successive tasks: the receipt of an intent, the call of a Content provider to insert data into database, the call of a remote Web Service via a Service component and finally the display of a screen. Testing these Activities requires a longer execution time than testing other components such as the Activities of *app 4* which directly display a screen.

Nonetheless, the longer test case execution times do not exceed few seconds. These results are coherent with other Android application testing methods e.g., the tool introduced by (Benli et al., 2012).

6 RELATED WORK

Security testing and the improvement of Android security are not new trends. Below, we compare our approach with some recent works from the literature.

Firstly, security testing, based upon formal models, has been studied in several works. (Le Traon et al., 2007) proposed a test generation technique to check whether security rules modelled with the OrBAC language hold on implementations under test. A mutation testing technique is considered for testing the access control policy robustness. (Marchand et al., 2009) proposed a method where test cases are generated from a specification and invariants or rules describing security policies. The main difference with our work is that we do not assume having a specification or a test case set. Instead, we propose a specification generation for Android components. (Mouelhi et al., 2008) introduced a test case generation for Java applications from security policies described with logic-based languages e.g., OrBAC, to describe access control properties. Policy enforcement points are injected into the code which is later tested with a mutation testing approach. Our work is not dedicated to access control policy. We also do not modify the original code. Furthermore, instead of considering a mutation technique to concretise test cases, we combine the use of SQL, XML injections, values known for relieving bugs and random testing.

(Marback et al., 2013) proposed a threat modelling based on trees. This method produces test cases from threat trees and transforms them into executable tests. Although the use of trees is intuitive for Industry, formal models offer several other advantages such as the description of the testing verdict without ambiguity. Furthermore, specifications are not considered in this related work, so false positive or negative results may be discovered with a higher rate than with our method.

Other works, dealing with Android security, have been also proposed recently. Some works focused on the definition of a more secure Android system. For instance, (Ongtang et al., 2009) proposed a monitoring technique to check the system integrity.

The analysis of the Android IPC mechanism were also studied by (Chin et al., 2011b). They described the permission system vulnerabilities that applications may exploit to perform unauthorised actions. Vulnerability patterns, that can be used with our method, can be directly extracted from this work. Other studies deal with the security of pre-installed Android Applications and show that target applications receiving oriented intents can re-delegate wrong permissions (Zhong et al., 2012; Grace et al., 2012). Some tools have been developed to detect the receipt of wrong permissions by means of malicious intents. In our work, we consider vulnerability patterns to model more general threats based on availability, integrity or authorisation, etc. (Jing et al., 2012) proposed a model-based conformance testing framework for the Android platform. Like in our approach, partial specifications are constructed from Manifest files. Nevertheless, the authors do not consider the Android documentation to augment the expressiveness of these specifications and consider implicit intents only. 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, provided in the paper, is based on the intent functioning and cannot be modified. Our work can take as input any vulnerability pattern.

7 CONCLUSION

In this paper, 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. We also intend to extend the automatic test case generation from the partial specifications of a component to automatically test some security vulnerabilities without providing patterns or to test other features such as robustness. Furthermore, our method tests components one by one without considering the underlying intents sent to other components. It could be interesting to check if these intents could be intercepted at runtime to inject malicious data for eventually detecting further vulnerabilities.

REFERENCES

AD (2013). Android developer page. In *http:// developer.android.com/index.html, last accessed feb 2013*.

Benli, S., Habash, A., Herrmann, A., Loftis, T., and Simmonds, D. (2012). A comparative evaluation of unit testing techniques on a mobile platform. In *Proceedings of the 2012 Ninth International Conference on Information Technology - New Generations*, ITNG '12, pages 263–268, Washington, DC, USA. IEEE Computer Society.

Chaudhuri, A. (2009). Language-based security on Android. In *PLAS '09: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, pages 1–7, New York, NY, USA. ACM.

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

Chin, E., Felt, A. P., Greenwood, K., and Wagner, D. (2011b). Analyzing inter-application communication in android. In *Proc. of the 9th International Conference on Mobile Systems, Applications, and Services*.

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.

Grace, M., Zhou, Y., Wang, Z., and Jiang, X. (2012). Systematic detection of capability leaks in stock Android smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*.

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.

Le Traon, Y., Mouelhi, T., and Baudry, B. (2007). Testing security policies: going beyond functional testing. In *ISSRE'07 (Int. Symposium on Software Reliability Engineering)*.

Marback, A., Do, H., He, K., Kondamarri, S., and Xu, D. (2013). A threat model-based approach to security testing. *Software: Practice and Experience*, 43(2):241–258.

Marchand, H., Dubreil, J., and Jéron, T. (2009). Automatic Testing of Access Control for Security Properties. In *TESTCOM/FATES 2009*.

Mouelhi, T., Fleurey, F., Baudry, B., and Traon, Y. (2008). 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*, pages 537–552.

Ongtang, M., McLaughlin, S., Enck, W., and McDaniel, P. (2009). Semantically rich application-centric security in android. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 340–349.

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.