Model-Checking for Android Malware Detection*

Fu Song[1] and Tayssir Touili[2]

[1] Shanghai Key Laboratory of Trustworthy Computing,
East China Normal University, P.R. China
fsong@sei.ecnu.edu.cn
[2] LIAFA, CNRS and Université Paris Diderot, France
touili@liafa.univ-paris-diderot.fr

Abstract. The popularity of Android devices results in a significant increase of Android malwares. These malwares commonly steal users' private data or do malicious tasks. Therefore, it is important to efficiently and automatically analyze Android applications and identify their malicious behaviors. This paper introduces an automatic and scalable approach to analyze Android applications and identify malicious applications. Our approach consists of modeling an Android application as a PushDown System (PDS), succinctly specifying malicious behaviors in Computation Tree Logic (CTL) or Linear Temporal Logic (LTL), and reducing the Android malware detection problem to CTL/LTL model-checking for PDSs. We implemented our techniques in a tool and applied it to analyze more than 1260 android applications. We obtained encouraging results. In particular, we discovered ten programs known as benign that are leaking private data.

1 Introduction

The rapid growth of Android's market results in a significant increase in Android malwares. Although Google introduced a security service *Bouncer* on the Android Play Store (Android Market) in February 2012, according to a recent report, the number of Android malwares has increased from 3063 to 51447 between the first and third quarters of 2012 [1]. These malwares usually steal users' private information such as phone identifiers, location information, or send overpriced messages, etc.

Researchers have done many efforts aimed at addressing these problems [2, 4, 7–10, 12–16, 19, 20, 22, 30]. All these works cannot directly analyze Dalvik codes to identify complicated malwares (Android applications are written in Java and compiled into Dalvik codes. Dalvik codes are a kind of assembly programs that run in Dalvik Virtual Machine, like Java bytecode run in Java Virtual Machine). In this work, we directly analyze Dalvik bytecode rather than translating it into Java and then using Java program analyzers. Indeed, several malwares are written directly in Dalvik. Moreover, decompilation from Android applications to Java does not always work, due to the fact

* This work was partially supported by STCSM Project (No. 14PJ1403200), NSFC Project (No. 61402179), SHMEC-SHEDF Project (No.13CG21), the Open Project of Shanghai Key Laboratory of Trustworthy Computing (No. 07dz22304201301), ANR grant (No.ANR-08-SEGI-006), SHEITC Project (No.130407), Shanghai Knowledge Service Platform for Trustworthy Internet of Things (No. ZF1213).

[1] http://www.f-secure.com.

J. Garrigue (Ed.): APLAS 2014, LNCS 8858, pp. 216–235, 2014.

that existing reverse engineering tools are prone to failure. We propose an efficient and automatic approach that directly analyzes Dalvik codes and can identify complicated malwares. Our approach consists of modeling an Android application as a Pushdown System (PDS) which is a natural model of sequential programs with potentially recursive procedure calls [11], and expressing malicious behaviors in SCTPL [25, 26] and SLTPL [27]. SCTPL (resp. SLTPL) is an extension of Computation Tree Logic (CTL) (resp. Linear Temporal Logic (LTL)) with variables, quantifiers and predicates over the stack that allows to succinctly describe malicious behaviors. The Android malware detection problem is reduced to SCTPL/SLTPL model-checking for PDSs which can be solved by [26, 27].

For instance, let us consider an Android application that intends to steal the IMSI ID of the phone by sending a text message to another phone. This application can obtain the IMSI ID by calling the *getSubscriberId* method whose return value is the IMSI ID. Later, it can call the *sendTextMessage* method with the IMSI ID as third parameter by which the IMSI ID is sent to another phone. Since the IMSI ID is users' private information, it is important to identify whether an Android application steals the IMSI ID or not. We can model the Android application as a PDS and specify this malicious behavior as the following SCTPL formula: $\mathbf{EF}\exists x(x = getSubscriberId() \wedge \mathbf{EF}sendTextMessage(-,-,x,-,-))$, where $-$ denotes the non-important parameters. This formula expresses that the return value of *getSubscriberId* (i.e., the IMSI ID) is assigned to a variable x. Later, *sendTextMessage* is called with x as third parameter. However, this formula is not robust enough and malwares could easily get around by some obfuscation techniques. For example, a malware could encrypt the IMSI ID such that the sent text (i.e., third parameter of *sendTextMessage*) does not have any explicit relation with the IMSI ID. E.g., the malware *Hongtoutou* uses the DES algorithm to encrypt the IMSI ID by the secret key 48734154. To overcome this problem, in this paper, we introduce a predicate *encode* to express the existence of a relation between two variables. More precisely, the predicate $y = encode(x, l)$ expresses that the value of y depends on the value of x at the control point l. Thus, the above malicious behavior can be specified in a more precise manner using the following SCTPL formula: $\mathbf{EF}\exists x \exists l(x = getSubscriberId() \wedge Loc(l) \wedge \mathbf{EF}\exists y(sendTextMessage(-,-,y,-,-) \wedge y = encode(x,l)))$. This formula specifies that the return value of *getSubscriberId* is assigned to a variable x at a control point l (i.e., $Loc(l)$ holds). Later, *sendTextMessage* is called with y as third parameter such that the value of y depends on the value of x at l.

However, it is not trivial to determine whether a configuration of the PDS model satisfies predicates of the form $y = encode(x, l)$ or not. To solve this problem, we propose an algorithm based on the saturation procedure of [11]. Our algorithm computes an *annotation function* from which we can infer whether a configuration satisfies $y = encode(x, l)$ or not. Thus, we can check whether an Android application has some malicious behaviors by applying SCTPL and SLTPL model-checking for PDSs.

We implemented our techniques in a tool and applied it to check 1260 Android malwares. We obtained interesting results. Our tool was able to detect all these malwares. We also applied our tool to check 71 applications from Android Compatibility Test Suite which are regarded as benign applications. We found that ten of them leak private data and three of them do some malicious behaviors such as record videos without the

```
1  class Myactivity extends Activity{
2    public String id;
3    public onCreate(){
4      TelephonyManager m = Context.getSystemService(``phone");
5      id=m.getDeviceId();
6      return;     }
7
8    public  onPause(){
9      SmsManager s=SmsManager.getDefault();
10     String text=encrypt(id, key);
11     s.sendTextMessage(``1",``2", text, intent, intent);
12     return;     }  }
```

Fig. 1. A simplified program that leaks the device ID of the phone via text message

users' knowledge. To our knowledge, the results we obtained for these 71 applications are previously unknown. Our approach could also be applied to detect other malicious programs, such as iOS programs, Windows programs, etc.

Outline: Section 2 presents the background of Android applications needed in this paper. Section 3 recalls the definition of PDSs and shows how to model an Android application as a PDS. Section 4 gives the definitions of SCTPL and SLTPL, and shows how to express malicious behaviors of Android applications in SCTPL/SLTPL. Section 5 proposes an algorithm computing the annotation function. Section 6 gives the experimental results. Section 7 shows related work. Due to lack of space, proofs are omitted. They can be found in the full version of this paper [28].

2 Android Applications

Android provides four base classes: *Activity*, *Service*, *Content Providers* and *Broadcast Receiver*, each of them consists of several methods that could be invoked by the Android operating system (OS) when its state is changed. These methods are called *callback methods*. For instance, the *Activity* class has two callback methods *onCreate* and *onPause* which will be invoked respectively by the Android OS when an *activity* is launched and is about to start resuming a previous activity. Also, there are other classes containing callback methods. E.g., the *OnClickListener* interface has the *onClick* method which will be called when the application is at the idle state and the corresponding button was clicked by the user. An Android application should define one or more classes that extend *Activity*, *Service*, *Content Providers* or *Broadcast Receiver* and the extended classes can override callback methods to implement their own functionalities. Moreover, an Android application does not necessarily have a *main* method (i.e., the entry point of a normal program). Instead, an application may have several entry points that are some callback methods of the four base classes. The Android OS can start an application by calling one of these callback methods. Malicious Android applications can also override the callback methods to execute a malicious task.

For example, Fig. 1 presents a simplified Android application that defines the *Myactivity* class which extends the *Activity* class. It overrides the *onCreate* and *onPause* methods. In the *onCreate* method, a *TelephonyManager* object *m* is obtained by calling

the *getSystemService* method at line 4. It calls the *getDeviceId* method of the object *m* and assigns the return value to the variable *id* at line 5. The return value of *getDeviceId* is the unique device ID (called *IMEI*) of the phone which is private. In the *onPause* method, it calls the *getDefault* method to obtain a *SmsManager* object *s* at line 9. Then, it encrypts the obtained device ID (i.e., IMEI) by calling the *encrypt* method and assigns the result to the variable *text* at line 10. Finally, it sends the value in the variable *text* via a text message by invoking the *sendTextMessage* method of *s* at line 11. Note that this program will send the user's device ID to other phones via text messages. Thus, this program may be malicious. It is important to analyze Android applications and tell the user what the applications will do before installing them.

3 Program Model

We will use pushdown systems (PDSs) to model Android applications. PDSs are suitable to model sequential programs with (potentially recursive) procedure calls [11]. The translation from the code of an Android application to a PDS is different from the standard translation from sequential programs to PDSs as it has to take into account the specificity of Android applications such as the existence of callback methods, the way these methods are called, and the absence of the main function.

3.1 Pushdown Systems

A *Pushdown System* (PDS) is a tuple $\mathcal{P} = (P, \Gamma, \Delta)$, where P is a finite set of control locations, Γ is the finite stack alphabet and $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ is a finite set of transition rules. A configuration of $\mathcal{P}$ is pair $\langle p, \omega \rangle$ with $p \in P$ and $\omega \in \Gamma^*$. If $((p, \gamma), (q, \omega)) \in \Delta$, we write $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$. W.l.o.g., for every $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle \in \Delta$, we assume $|\omega| \leq 2$ [11].

The successor relation $\rightsquigarrow_{\mathcal{P}} \subseteq (P \times \Gamma^*) \times (P \times \Gamma^*)$ is defined as follows: if $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$, then $\langle p, \gamma\omega' \rangle \rightsquigarrow_{\mathcal{P}} \langle q, \omega\omega' \rangle$ for every $\omega' \in \Gamma^*$. If $\langle p, \gamma\omega' \rangle \rightsquigarrow_{\mathcal{P}} \langle q, \omega\omega' \rangle$, then $\langle q, \omega\omega' \rangle$ is a successor of $\langle p, \gamma\omega' \rangle$. A path is a sequence of configurations $c_0 c_1 ...$ such that for every $i \geq 0$, $c_i \rightsquigarrow_{\mathcal{P}} c_{i+1}$. Let $\rightsquigarrow^*_{\mathcal{P}} \subseteq (P \times \Gamma^*) \times (P \times \Gamma^*)$ be the transitive and reflexive relation of $\rightsquigarrow_{\mathcal{P}}$ such that for every $c, c' \in P \times \Gamma^*$, $c \rightsquigarrow^*_{\mathcal{P}} c$, and $c \rightsquigarrow^*_{\mathcal{P}} c'$ iff there exists $c'' \in P \times \Gamma^*$: $c \rightsquigarrow_{\mathcal{P}} c''$ and $c'' \rightsquigarrow^*_{\mathcal{P}} c'$. Let $post^* : 2^{P \times \Gamma^*} \longrightarrow 2^{P \times \Gamma^*}$ be the *successor function* such that for every $C \subseteq 2^{P \times \Gamma^*}$, $post^*(C) = \{c \in P \times \Gamma^* \mid \exists c' \in C : c' \rightsquigarrow^*_{\mathcal{P}} c\}$.

To finitely represent (potentially) infinite sets of configurations of PDSs, we use multi-automata.

Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, a *Multi-Automaton* (MA) [3] is a tuple $\mathcal{M} = (Q, \Gamma, \delta, I, F)$, where Q is a finite set of states, $\delta : (Q \times \Gamma) \times Q$ is a finite set of transition rules, $I \subseteq Q$ is a set of initial states corresponding to the control locations P, $F \subseteq Q$ is a finite set of final states

Let $\rightarrow_\delta : Q \times \Gamma^* \times Q$ be the transition relation such that for every $q \in Q$: $q \xrightarrow{\epsilon}_\delta q$ and $q \xrightarrow{\gamma\omega}_\delta q'$ if there exists a state $q'' \in Q$ such that $(q, \gamma, q'') \in \delta$ and $q'' \xrightarrow{\omega}_\delta q'$. A configuration $\langle p, \omega \rangle \in P \times \Gamma^*$ is accepted by $\mathcal{M}$ iff $p \xrightarrow{\omega}_\delta q$ for some $q \in F$. A set of configurations $C \subseteq P \times \Gamma^*$ is *regular* iff there exists a MA $\mathcal{M}$ such that $\mathcal{M}$ exactly accepts the set of configurations C. Let $L(\mathcal{M})$ be the set of configurations accepted by $\mathcal{M}$.

3.2 Modeling Android Applications as PDSs

In this section, we show how to model an Android application as a PDS. Given an application with a set N of control points (excluding the control points of declaration statements, e.g., the control point 2 in Fig. 1.), we construct a PDS $\mathcal{P} = (\{p\}, N \cup \{\gamma_\perp\}, \Delta)$ with p as the unique control location and $N \cup \{\gamma_\perp\}$ as the stack alphabet, where $\gamma_\perp \notin N$ is used to handle entry points and callback methods. The PDS transition rules model the control flow of the application. (In our implementation, we use Smali[2], a disassembler for Android applications, to disassemble the application into control flow graphs.) Intuitively, the configuration $\langle p, \gamma_\perp \rangle$ is the initial configuration of the PDS model. It denotes that the run of the PDS is at the idle state (i.e., the application does not execute any statement). A configuration $\langle p, \gamma\omega \rangle$ such that $\gamma \in N$ denotes that the run of the application is at the control point γ and ω is the return addresses of the calling procedures (i.e., the procedures that have not returned yet). Formally, Δ is computed as follows: for every control point $\gamma \in N$ s.t. *stmt* is the statement at the control point γ:

1. If *stmt* is a function call $v = f(v_1, ..., v_m)$ and γ' is the next control point of γ, then $\langle p, \gamma \rangle \hookrightarrow \langle p, f_e \gamma' \rangle \in \Delta$, where f_e is the entry point of the procedure f and γ' is regarded as the return address of f;
2. If *stmt* is a return statement *return v*, then $\langle p, \gamma \rangle \hookrightarrow \langle p, \epsilon \rangle \in \Delta$, where ϵ is the empty word;
3. If *stmt* is neither a function call nor a return statement and γ' is the next control point of γ, then $\langle p, \gamma \rangle \hookrightarrow \langle p, \gamma' \rangle \in \Delta$;
4. Moreover, for every callback method *proc* in the application, $\langle p, \gamma_\perp \rangle \hookrightarrow \langle p, proc_e \gamma_\perp \rangle \in \Delta$, where $proc_e$ is the entry point of *proc*.

The first three items describe the standard construction of a PDS model from a sequential program as shown in [11]. The last item models the invoking of callback methods. As explained previously, an Android application can override the callback methods that are invoked by the Android OS. This implies that some callback methods may not be reachable if we only use the first three items, but they can be called by the Android OS. For example, let us consider the program shown in Fig. 1. The function *onCreate* is only called by the Android OS when the activity is launched and can be an entry point of the application. The *onClick* method of an *ok* button that implements the *OnClickListener* interface is called only when the *ok* button is clicked by the user. That is why we add the last item by which all the callback methods could be invoked in any order whenever the application is at an idle state, i.e., the PDS is at the configuration $\langle p, \gamma_\perp \rangle$. From the view point of the application, we associate all the function calls of callback methods to the control point $\gamma_\perp$. The resulting PDS model is a sound over-approximation of the application.

4 Android (Malicious) Behaviors Specifications

In this section, we recall the definition of the logics SLTPL [27] and SCTPL [26], and show how to use them to describe Android (malicious) behaviors.

[2] `http://code.google.com/p/smali`

Hereafter, we fix the following notations. Let $\mathcal{X} = \{x_1, x_2, ...\}$ be a finite set of variables ranging over a finite domain $\mathcal{D}$. Let $B : \mathcal{X} \cup \mathcal{D} \longrightarrow \mathcal{D}$ be an environment function that assigns a value $v \in \mathcal{D}$ to each variable $x \in \mathcal{X}$ and such that $B(v) = v$ for every $v \in \mathcal{D}$. $B[x \leftarrow v]$ denotes the environment function such that $B[x \leftarrow v](x) = v$ and $B[x \leftarrow v](y) = B(y)$ for every $y \neq x$. Let AP be a finite set of atomic propositions, $AP_{\mathcal{X}}$ be a finite set of atomic predicates in the form of $a(\alpha_1, ..., \alpha_m)$ such that $a \in AP$, $\alpha_i \in \mathcal{X} \cup \mathcal{D}$ for every $1 \leq i \leq m$, and $AP_{\mathcal{D}}$ be a finite set of atomic predicates of the form $a(\alpha_1, ..., \alpha_m)$ such that $a \in AP$, $\alpha_i \in \mathcal{D}$ for every $1 \leq i \leq m$.

4.1 The SCTPL Logic

SCTPL can be seen as an extension of CTL with variables, quantifiers and predicates over the stack. Variables are parameters of atomic predicates and can be quantified by the existential and universal quantifiers. Formally, the set of *SCTPL formulas* is given by (where $x \in \mathcal{X}$ and $a(x_1, ..., x_m) \in AP_{\mathcal{X}}$):

$$\varphi ::= a(x_1, ..., x_m) \mid \neg\varphi \mid \varphi \wedge \varphi \mid \forall x\, \varphi \mid \mathbf{EX}\varphi \mid \mathbf{EG}\varphi \mid \mathbf{E}[\varphi \mathbf{U} \varphi].$$

Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, let $\lambda : AP_{\mathcal{D}} \rightarrow 2^{P \times \Gamma^*}$ be a labeling function that assigns to each predicate a regular set of configurations. Let $c \in P \times \Gamma^*$ be a configuration of $\mathcal{P}$. $\mathcal{P}$ satisfies a SCTPL formula ψ in c, denoted by $c \models_\lambda \psi$, iff there exists an environment B such that $c \models^B_\lambda \psi$, where $c \models^B_\lambda \psi$ is defined by induction as follows:

- $c \models^B_\lambda a(x_1, ..., x_m)$ iff $c \in \lambda\Big(a(B(x_1), ..., B(x_m))\Big)$.
- $c \models^B_\lambda \psi_1 \wedge \psi_2$ iff $c \models^B_\lambda \psi_1$ and $c \models^B_\lambda \psi_2$.
- $c \models^B_\lambda \forall x\, \psi$ iff $\forall v \in \mathcal{D},\ c \models^{B[x \leftarrow v]}_\lambda \psi$.
- $c \models^B_\lambda \neg\psi$ iff $c \not\models^B_\lambda \psi$.
- $c \models^B_\lambda \mathbf{EX}\, \psi$ iff there exists a successor c' of c s.t. $c' \models^B_\lambda \psi$.
- $c \models^B_\lambda \mathbf{E}[\psi_1 \mathbf{U} \psi_2]$ iff there exists a path $\pi = c_0 c_1 ...$ of $\mathcal{P}$ with $c_0 = c$ s.t. $\exists i \geq 0,\ c_i \models^B_\lambda \psi_2$ and $\forall 0 \leq j < i, c_j \models^B_\lambda \psi_1$.
- $c \models^B_\lambda \mathbf{EG}\psi$ iff there exists a path $\pi = c_0 c_1 ...$ of $\mathcal{P}$ with $c_0 = c$ s.t. $\forall i \geq 0$: $c_i \models^B_\lambda \psi$.

Intuitively, $c \models^B_\lambda \psi$ holds iff the configuration c satisfies ψ under the environment B. We will freely use the following abbreviations: $\mathbf{EF}\psi = \mathbf{E}[true \mathbf{U} \psi]$, $\mathbf{AG}\psi = \neg\mathbf{EF}(\neg\psi)$, and $\exists x \psi = \neg\forall x \neg\psi$.

Theorem 1. *[26] SCTPL model-checking for PDSs is decidable.*

4.2 The SLTPL Logic

Similarly, SLTPL can be seen as an extension of LTL with variables, quantifiers and predicates over the stack. The set of *SLTPL formulas* is given by (where $x \in \mathcal{X}$ and $a(x_1, ..., x_m) \in AP_{\mathcal{X}}$): $\varphi ::= a(x_1, ..., x_m) \mid \neg\varphi \mid \varphi \wedge \varphi \mid \forall x\, \varphi \mid \mathbf{X}\varphi \mid \varphi \mathbf{U} \varphi$.

Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$ and a path $\pi = c_0 c_1 ...$ of $\mathcal{P}$, let $\pi(i)$ denote c_i and π^i denote the *suffix* starting from $\pi(i)$. Let c be a configuration of $\mathcal{P}$. $\mathcal{P}$ satisfies a SLTPL formula ψ in c (denoted by $c \models_\lambda \psi$) iff there exists an environment B such that c satisfies ψ under B (denoted by $c \models^B_\lambda \psi$). $c \models^B_\lambda \psi$ holds iff there exists an execution π starting from c such that π satisfies ψ under B (denoted by $\pi \models^B_\lambda \psi$), where $\pi \models^B_\lambda \psi$ is defined by induction as follows:

- $\pi \models^B_\lambda a(x_1, ..., x_m)$ iff $\pi(0) \in \lambda\Big(a(B(x_1), ..., B(x_m))\Big)$;
- $\pi \models^B_\lambda \neg\psi_1$ iff $\pi \not\models^B_\lambda \psi_1$;
- $\pi \models^B_\lambda \psi_1 \wedge \psi_2$ iff $\pi \models^B_\lambda \psi_1$ and $\pi \models^B_\lambda \psi_2$;
- $\pi \models^B_\lambda \forall x\, \psi$ iff for every $v \in \mathcal{D}$, $\pi \models^{B[x\leftarrow v]}_\lambda \psi$;
- $\pi \models^B_\lambda \mathbf{X}\, \psi$ iff $\pi^1 \models^B_\lambda \psi$;
- $\pi \models^B_\lambda \psi_1 \mathbf{U} \psi_2$ iff there exists $i \geq 0$ s.t. $\pi^i \models^B_\lambda \psi_2$ and $\forall j,\ 0 \leq j < i : \pi^j \models^B_\lambda \psi_1$;

We will freely use the following abbreviations: $\mathbf{F}\psi = true\mathbf{U}\psi$, $\mathbf{G}\psi = \neg\mathbf{F}(\neg\psi)$ and $\exists x\psi = \neg\forall x\neg\psi$.

Theorem 2. *[27] SLTPL model-checking for PDSs is decidable.*

4.3 SLTPL and SCTPL for Android Applications

In the context of Android applications, usually AP consists of the method names. For the sake of readability, predicates such as $f(x_1, ..., x_n)$ in $AP_\mathcal{X}$ will sometimes be written as $x_1 = x_n.f(x_2, ..., x_{n-1})$ when x_1 denotes the return value of f and x_n denotes the object having the method f, where $x_2, ..., x_{n-1}$ are f's parameters. The labeling function λ is syntactically extracted from the application. For every function call $v = f(v_1, ..., v_m)$, every $\omega \in \Gamma^*$, $\langle p, \gamma\omega\rangle \in \lambda(v = f(v_1, ..., v_m))$ iff $v = f(v_1, ..., v_m)$ is called at the point γ.

Example 1. Consider the program shown in Fig. 1 and the following SCTPL formula: $\phi = \exists x_1 \exists x_2 \mathbf{EF}\Big(x_1 = x_2.getDeviceId() \wedge (\mathbf{EF}\exists x_3 \exists x_4\ x_3 = encrypt(x_1, x_4) \wedge \exists x_5 \exists x_6 \exists x_7 \exists x_8 \exists x_9\ \mathbf{EF} x_5.sendTextMessage(x_6, x_7, x_3, x_8, x_9))\Big)$, we have: $\mathcal{X} = \{x_1, ..., x_9\}$ is the set of variables appearing in ϕ; $AP = \{getDeviceId, sendTextMessage, encrypt\}$ is the set of atomic propositions corresponding to method names (we only list the propositions that are used in ϕ); $AP_\mathcal{X} = \{x_1 = x_2.getDeviceId(), x_3 = encrypt(x_1, x_4), x_5.sendTextMessage(x_6, x_7, x_3, x_8, x_9)\}$ is the set of predicates appearing in ϕ; $\mathcal{D} = \{m, id, s, key, \text{“}phone\text{”}, text, \text{“}1\text{”}, \text{“}2\text{”}, intent\}$ is the set of variables and constants that appear in the program; $AP_\mathcal{D} = \{id = m.getDeviceId(), s.sendTextMessage(\text{“}1\text{”}, \text{“}2\text{”}, text, intent, intent), text = encrypt(id, key)\}$ is the set of function calls; the labeling function λ is given as follows: $\lambda(id = m.getDeviceId()) = \{\langle p, l_5\omega\rangle \mid \omega \in \Gamma^*\}$, $\lambda(s.sendTextMessage(\text{“}1\text{”}, \text{“}2\text{”}, text, intent, intent)) = \{\langle p, l_{10}\omega\rangle \mid \omega \in \Gamma^*\}$ and $\lambda(text = encrypt(id, key)) = \{\langle p, l_9\omega\rangle \mid \omega \in \Gamma^*\}$.

Simplified Formulas: A variable x is *non-important* in a formula if x is quantified by $\exists$ and occurs only in one atomic predicate. All the non-important variables will be replaced by “$-$”. Let us consider the behavior that sends the IMEI (or an encrypted version of it so that it becomes difficult to check that the IMEI is sent) to other phones via text messages as shown in Fig. 1. We can specify this behavior in the SCTPL formula ϕ (in Example 1). ϕ states that there exist a *TelephonyManager* object x_2 and a variable x_1 such that the return value of the *getDeviceId* method of x_2 (i.e., IMEI) is assigned to x_1. Later, there exist a variable x_3 and a key x_4 such that *encrypt* is invoked with parameters x_1 and x_4, the return value is assigned to x_3 (i.e., the IMEI stored in x_1 is encrypted with the key x_3 and the encrypted IMEI is stored in x_3). Finally, there exist a *SmsManager* object x_5 and variables $x_6, ..., x_9$ such that the *sendTextMessage* method of x_5 is called with

parameters x_6, x_7, x_3, x_8 and x_9 (i.e., the encrypted IMEI is sent by calling *sendTextMessage*). The variable x_2 is quantified by $\exists$ and only occurs in $x_1 = x_2.getDeviceId()$, then, we can simplify $x_1 = x_2.getDeviceId()$ as $x_1 = -.getDeviceId()$ which is written as $x_1 = getDeviceId(-)$. The same holds for the variables $x_4, .., x_9$. Thus, the formula is simplified as $\Phi_{id} = \exists x_1 \mathbf{EF}\Big(x_1 = getDeviceId(-) \wedge (\mathbf{EF}\exists x_3\ x_3 = encrypt(x_1, -) \wedge \mathbf{EF}sendTextMessage(-, -, x_3, -, -, -))\Big)$.

4.4 Expressing Android (Malicious) Behaviors in SCTPL and SLTPL

In this section, we show how to use SCTPL/SLTPL to express malicious behaviors. We need a special predicate of the form $y = encode(x, l)$ to express that the value of y is computed by encrypting the value of x at the control point l and a predicate of the form $Loc(l)$ to denote that the control point is l, where a configuration $\langle p, \gamma\omega\rangle$ for every $\omega \in \Gamma^*$ satisfies $Loc(l)$ iff $l = \gamma$.

4.4.1 The Predicate *Encode*

The formula Φ_{id} given at the end of Section 4.3 is not robust enough for specifying the behavior that sends the device ID (which may be encrypted) to other phones via text messages. A malware writer could use other approaches to change the IMEI instead of calling the *encrypt* method. For example, a malware writer can replace the statements at lines 9 and 10 in Fig. 1 by the following code:

```
for(int i=0; i<id.length();i++){
    String text=id.get(i);
    text=text+i;
    s.sendTextMessage(``1'',``2'', text, intent, intent);
}
```

where for every i from 0 upto the length $id.length()$ of the string id (i.e., the IMEI), first, a letter at position i in id is obtained by calling $id.get(i)$ which is assigned to the variable *text*, then the position number i is appended to the string stored in *text* (i.e., $text = text + i$). Finally, the string stored in *text* is sent by calling *sendTextMessage*. By doing this, the IMEI is sent one letter by one letter, and each letter is sent appended with its position. E.g., suppose the IMEI is the string *abcd*, then $a0$, $b1$, $c2$ and $d3$ are sent one by one. Thus, to make the behavior specification more robust, we introduce a new predicate *encode*. Intuitively, $y = encode(x, l)$ expresses that the value of the variable y depends on the value of the variable x at the control point l. Formally, a configuration $\langle p, \gamma\omega\rangle$ satisfies a predicate $y = encode(x, l)$ iff the run of the program starting from the entry point reaches the control point γ such that the value of y depends on the value of x at the control point l. We can specify the above behavior in a more precise way as follows: $\Psi_{id} = \mathbf{EF}\exists x_1 \exists l\Big(x_1 = getDeviceId(-) \wedge Loc(l) \wedge \mathbf{EF}\exists x_3(sendTextMessage(-, -, x_3, -, -, -) \wedge x_3 = encode(x_1, l))\Big)$. Ψ_{id} states that the return value of *getDeviceId* (i.e., IMEI) is assigned to a variable x_1 at the control point l. Later, *sendTextMessage* is called with x_3 as third parameter when the value of x_3 depends on the value of x_1 at l (i.e., the (encrypted) IMEI is sent via text messages).

Table 1. Privates data sources and sinks

Descriptions of source functions
The return value of *getLatitude* or *getlongitude* is the location of the phone
The first parameter of *onLocationChanged* contains the location data of the phone
The return value of *getDeviceId* is the IMEI id of the phone
The return value of *getSubscriberId* is the IMSI id of the phone
The return value of *getDeviceSoftwareVersion* is the IMSI/SV of the phone
The return value of *getLine1Number* is the phone number (PN)
The return value of *getNetworkCountryIso* is the Phone's Iso country code (ISOC)
The first parameter of *getNetworkCountryIso* is the incoming phone number (IPN)
The return value of *getResult* of *AccountManagerFuture* class contains the authentication token (AT) of the phone
The return value of *query* or *managedQuery* is the contact or calendar data (CC) of the phone
The second parameter of *setOutputFile* contains the media data (MD) of the phone
The return value of *getExternalStorageDirectory* contains the SD card (SDC) data of the phone
The return value of *getConnectionInfo* contains the WiFi network connection information of the phone
The return value of *getStringExtra* of the *Intent* class contains the data of an Intent object
Descriptions of sink functions
The third (resp. fourth) parameter of *sendTextMessage* (resp. *sendMultipartTextMessage*) leaks data via a text messages.
The first and second parameters of *d,e,i,v,w,wtf* leak data by writing into log files
The first and second parameters of *loadurl* leaks data via network connections
The first-fourth parameters of ⟨@1init⟩@1 in the *URL* class leak data via network connections
The fourth-eighth parameters of *set* in the *URL* class leak data via network connections
The first parameter of *setRequestProperty* leak data via network connections
The first and second parameters of *execute* in the *http* class can leak data via network connections
The first parameter of *write* or *println* leak data by writing data to files
The first parameter of ⟨*init*⟩ of the *Intent* class leak data to other applications or components

4.4.2 Malicious Behaviors in SCTPL or SLTPL

Information-Leaks: A *source function* is a function that will return a private data through a return value or a parameter. A *source port* is a variable that stores the private data of a source function. A *sink function* is a function which can leak some private data through some parameters of the function. A *sink port* is a parameter of a sink function that can leak some private data. An *information-leak* is the behavior where a sink function is called and its sink port stores some private data (usually got from a source function). This kind of malicious behavior could be specified in SCTPL as the pattern:

$$\mathbf{EF}\exists x \exists l(f_1(x) \wedge Loc(l) \wedge \mathbf{EF}\exists y(f_2(y) \wedge y = encode(x, l)))$$

where f_1 (resp. f_2) is a source (resp. sink) function and x (resp. y) is a source (resp. sink) port such that the value of y relies on the value of x at the control point l. E.g., the application shown in Fig. 1 has an information leak behavior that sends the IMEI of the phone to the other phone via text messages. The formula Ψ_{id} is an instance of the pattern, where $getDeviceId$ and $sendTextMessage$ are the source and sink functions, respectively. x_1 and x_3 are the source and sink ports. In Table 1, we give all the source and sink functions considered in this work.

Background Picture Taking: An application may take a picture using a camera of the phone without the user's knowledge. To take a picture, an application first creates a new *Camera* object to access a particular hardware camera by invoking the *open* method of the *Camera* class. Next, it calls the *setPreviewDisplay* or *setPreviewTexture* method with the *Camera* object as first parameter to set a surface to preview, and then calls the *takePicture* method with the *Camera* object to take a picture. Calling

setPreviewDisplay or *setPreviewTexture* will inform the user about a camera access. But, without calling them before taking the picture (i.e., calling *takePicture*) after the *Camera* object is created (i.e., calling *open*) will take a picture without informing the user. Thus, this behavior is malicious. We can specify this behavior in a SLTPL formula as follows: $\Psi_{bp} = \mathbf{F}\exists x_1 \exists l_1 \Big(x_1 = open(-) \wedge Loc(l_1) \wedge \exists x_2 (\neg((setPreviewDisplay(x_2,-) \vee setPreviewTexture(x_2,-)) \wedge x_2 = encode(x_1,l_1))\ \mathbf{U}\ \exists x_3\ takePicture(x_3) \wedge x_3 = encode(x_1,l_1))\Big)$. The formula Ψ_{bp} states that a *Camera* object x_1 is created by calling *open* at l_1. Later, a picture is taken by calling *takePicture* with x_3 as its first parameter such that the value of x_3 is obtained from the value of x_1 at l_1 (i.e., $x_3 = encode(x_1, l_1)$), since the *Camera* object stored in x_1 can be assigned to another variable x_3. Between calling *open* and *takePicture*, there does not exist a variable x_2 such that *setPreviewDisplay* or *setPreviewTexture* is called with x_2 as first parameter and the value of x_2 is obtained from the value of x_1 at l_1 (i.e. $x_2 = encode(x_1, l_1)$). This means that a picture is taken without informing the user.

Background Video Recording: Android provides the *MediaRecorder* class to record a video using a camera of the phone. To do this, an application first creates a *MediaRecorder* object, then calls the *setVideoSource* method to choose a camera (a phone may have two cameras). The application should call the *setPreviewDisplay* method to set a surface to show a preview of the video. Thus, an application recording the video without calling *setPreviewDisplay*, i.e., informing the user, is malicious. We can specify this behavior in SCTPL as follows: $\Psi_{bv} = \exists x_1 \exists l_1 \mathbf{E}[\neg(setPreviewDisplay(x_1,-) \wedge Loc(l_1))\ \mathbf{U}\ \exists x_2 setVideoSource(x_2,-) \wedge x_2 = encode(x_1,l_1) \wedge \mathbf{AG}\neg\exists x_3 setVideoSource(x_3,-) \wedge x_3 = encode(x_1,l_1)]$. Ψ_{bv} states that there does not exist a *MediaRecorder* object x_1 such that the calling of *setVideoSource* with x_2 as its first parameter such that the value of x_2 depends on x_1 (i.e., $x_2 = encode(x_1, l_1)$) is not preceded by calling *setPreviewDisplay* with x_1 as its first parameter at l_1. Later, in all the future paths, *setVideoSource* will not be called with x_3 as its first parameter such that the value of x_3 is obtained from the value of x_1 at l_1 (i.e., $x_3 = (x_1, l_1)$).

Dynamically Loaded Code Execution: In Android, an application can dynamically load classes from libraries and call functions in these classes. To do this, it first calls *loadclass* to load a class from a library. Then, the return value is the class object. Later, it calls the *getMethod* method with the class object as its first parameter. This returns the method. Finally, it can call the method by calling *invoke* with the method as parameter. The loaded classes may perform malicious behaviors that cannot be identified by statically checking the application. Thus, it is important to tell the user whether an application executes some dynamically loaded code. To check this, we use the following SCTPL formulas: $\Psi_{dc} = \mathbf{EF}\exists x_1 \exists l_1 \Big(x_1 = loadClass(-,-) \wedge Loc(l_1) \wedge \exists x_2 \exists x_3 \exists l_2 \mathbf{EF}(x_3 = getMethod(x_2,-) \wedge Loc(l_2) \wedge x_2 = encode(x_1,l_1) \wedge \exists x_4 \mathbf{EF} invoke(x_4,-,-) \wedge x_4 = encode(x_3,l_2))\Big)$. Ψ_{dc} states that the x_1 class is dynamically loaded by calling *loadClass* at l_1. Next, *getMethod* is called with x_2 as first parameter such that the value of x_2 is obtained from the value of x_1 at l_1 (i.e., $x_2 = encode(x_1, l_1)$), since the class object stored in x_1 may be assigned to another variable x_2. Later, *invoke* is called at l_2 with x_4 as the first parameter such that the value of x_4 is obtained from the return value x_3 of

the previous *getMethod* method call, i.e., $x_4 = encode(x_3, l_2)$ and the method stored in x_4 (x_3) is invoked by the application.

Harvesting Installed Applications: Android provides the *getInstalledPackages* method of the *PackageManager* class to access information of the installed applications, their components, and permissions. An application harvesting the installed applications is dangerous, as the installed applications are users' private data. To harvest the installed applications, an application can call *getInstalledPackages* which returns a list of installed applications. Then, the application can traverse this list using the *hasNext* function of the *Iterator* class or the *get* function of the *List* class. Since a conditional statement is modeled as two non-deterministic PDS transition rules when we model an application as a PDS, then, the traversing of the list which checks whether all the elements are visited will be an infinite loop. This is an over-approximation of the control flow of the application. We can specify this behavior in the following SLTPL formula: $\Psi_{hi} = \mathbf{F}\exists x_1 \exists l_1 (x_1 = getInstalledPackages(-,-) \wedge Loc(l_1) \wedge \mathbf{GF}\exists x_2 \wedge (get(x_2,-) \vee hasNext(x_2)) \wedge x_2 = encode(x_1, l_1))$. Ψ_{hi} states that a list x_1 of installed applications is obtained by calling *getInstalledPackages* at l_1. Later, it will infinitely often access this list by calling *get* or *hasNext* with x_2 as first parameter such that the value of x_2 is obtained from the value of x_1 at l_1 (i.e., $x_2 = encode(x_1, l_1)$). Note that the always operator **G** specifies the infinite loop that traverses the list of installed applications.

Native Codes Execution: Android Applications have a way to execute native codes that are written in other languages such as C/C++. Applications can execute native libraries by calling the *loadLibrary* method (i.e., the Java Native Interface) or the *exec* method of the *Runtime* object. As these codes are not in Android assembly language and may contain malicious behaviors, it is crucial to tell the user whether an application will execute codes in native libraries. For this, we check whether the *loadLibrary* or *exec* is called or not by the following formula: $\Psi_{nc} = \mathbf{EF}(loadLibrary(-) \vee exec(-))$. Ψ_{nc} checks whether the function *loadLibrary* or *exec* is called.

Downloading Data from Servers: Many applications download payloads from servers. They may download malicious applications and the downloading costs network flow. Thus, it is important to check whether an application downloads data from some servers. An application can use the *getInputStream* method of the *URLConnection*, *HttpURLConnection* or *HttpsURLConnection* classes to obtain an *InputStream* object. The *InputStream* object allows the application to read data from the server by calling the *read* method of the *InputStream* class. By doing so, the data read from the *InputStream* object is put at the buffer pointed by the second parameter of the *read* method. Then, an application can write the data to a file by calling the *write* method of the *FileOutputStream* class with the buffer as second parameter. Thus, it is important to check whether an application reads data using an *InputStream* object and then writes this data into a file. We can express this behavior in a SCTPL formula as follows: $\Psi_{dd} = \mathbf{EF}\exists x_1 \exists l_1 \Big(x_1 = getInputStream(-) \wedge Loc(l_1) \wedge \mathbf{EF}\exists x_2 \exists x_3 \exists l_2 (read(x_2, x_3) \wedge Loc(l_2) \wedge x_2 = encode(x_1, l_1) \wedge \mathbf{EF}\exists x_4\ write(-, x_4, -, -) \wedge x_4 = encode(x_3, l_2))\Big)$. Ψ_{dd} expresses that the return value of *getInputStream* is assigned to the variable x_1 (i.e., x_1 is an *InputStream* object) at l_1. Next, *read* is called at control point l_2 with x_2 and x_3 as its parameters such that the value of x_2 is obtained from the value of x_1 at l_1

(i.e., $x_2 = encode(x_1, l_1)$), since the *InputStream* object stored in x_1 may be assigned to another variable x_2. This means that the application reads data from a server by calling the *read* method of the *InputStream* object and the data is put at the buffer x_3. Later, *write* is called with x_4 as its second parameter whose value is obtained from the value of x_3 at l_2 which stores the data from the server.

Remark 1. Note that we need both SLTPL and SCTPL to be able to express the Android malicious behaviors. Indeed, the SCTPL formula Ψ_{bv} cannot be expressed in SLTPL, whereas the SLTPL formula Ψ_{hi} cannot be expressed in SCTPL.

5 Model-Checking Android Applications

As described previously, we model an Android application as a PDS and specify Android malicious behaviors in SCTPL/SLTPL. Then, to check whether an application contains a malicious behavior or not, it is sufficient to check whether the PDS model satisfies the SCTPL/SLTPL formula expressing the malicious behavior. However, it is non-trivial to decide whether or not a configuration $\langle p, \gamma\omega\rangle$ satisfies a predicate of the form $v_2 = encode(v_1, l)$, since one cannot easily determine whether the value of v_2 depends on the value of v_1 at the control point l or not. To solve this problem, in this section, we propose an approach to compute an *annotation function* that allows us to determine whether a configuration satisfies predicates of the form $v_2 = encode(v_1, l)$. Intuitively, the annotation function associates to each control point n of the program a *dependency function*, where the dependency function assigns to each variable x a set of pairs (y, l) expressing that the value of x at the control point n depends on the value of y at l. The annotation function is computed by an extension of the saturation procedure of [11] which computes all the reachable configurations represented by a MA of the PDS model. We assign to each transition of the MA a dependency function and update the dependency function during the saturation procedure according to the side-effects of the program statements. To distinguish variables in SCTPL/SLTPL formulas from those that appear in the applications, from now on, we will use x, y, z to denote variables in SCTPL/SLTPL formulas, and use v, v_1, v_2... to denote variables in applications.

5.1 Annotating the Program with *encode* Predicates

Let us fix a PDS $\mathcal{P} = (P, \Gamma, \Delta)$ modeling a given Android application. For every $\gamma \in \Gamma$, let $Proc(\gamma)$ be the procedure that contains the control point γ. Let G be the set of global variables used in an application and L_{proc} be the set of local variables in the procedure $proc$. For each procedure $proc$, let R_{proc} be a local variable of the procedure $proc$ which denotes the return value of $proc$ after the return statement. The formal parameters $p_1, ..., p_m$ of each procedure are local variables of this procedure. Let L be the set of local variables used in the application. Let $\theta : G \cup L \longrightarrow 2^{(G\cup L)\times\Gamma}$ be a *dependence function* that assigns to each variable $v \in G \cup L$ a set of pairs (v', γ) such that v depends on the value of v' at the control point γ. Let Θ be the set of dependence functions. Let $\uplus : \Theta\times\Theta \longrightarrow \Theta$ be a function such that for every $\theta, \theta' \in \Theta$, every variable $v \in G \cup L$: $(\theta \uplus \theta')(v) = \theta(v) \cup \theta'(v)$.

Let $\mathcal{M} = (Q, \Gamma, \delta, I, F)$ be the MA where $Q = \{p, q_f\}$, $I = \{p\}$, $F = \{q_f\}$ and $\delta = \{(p, \gamma_\perp, q_f)\}$. $\mathcal{M}$ accepts the configuration $\langle p, \gamma_\perp \rangle$ (i.e., the initial configuration). We create a new MA $\mathcal{M}^* = (Q^*, \Gamma, \delta^*, I, F)$ with ϵ-transition rules and an annotation function $\rho : \delta^* \longrightarrow \Theta$ that associates each transition rule of $\mathcal{M}^*$ with a dependence function θ such that $L(\mathcal{M}^*) = post^*(L(\mathcal{M}))$, and for every control point γ, $\mathcal{M}^*$ has a transition rule $t = (p, \gamma, q_1)$ such that $q_1 \xrightarrow{\omega}_{\delta^*} q_f$ and $(v, l) \in \rho(t)(v')$ iff a configuration $\langle p, \gamma\omega' \rangle$ satisfies $v' = encode(v, l)$.

Let $Var(exp)$ be the set of variables used in the expression exp. The computation of $\mathcal{M}^*$ and ρ consists of two steps. First, we construct the MA $\mathcal{M}^*$ accepting $post^*(L(\mathcal{M}))$. Then, we annotate the transition rules of $\mathcal{M}^*$ with an adequate dependence function (i.e. compute ρ). We use the saturation procedure of [11] to compute $\mathcal{M}^*$ by adding a finite number of transition rules into $\mathcal{M}$ based on the following rules: Initially, $\mathcal{M}^*$ equals $\mathcal{M}$;

- For every transition rule $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle \in \Delta$, add a new state $p'_{\gamma'}$, and a new transition rule $(p', \gamma', p'_{\gamma'})$ into $\mathcal{M}^*$;
- Add new transition rules into $\mathcal{M}^*$ according to the following saturation rules: for every $p \xrightarrow{\gamma}_{\delta^*} q$ in $\mathcal{M}^*$,
  - If $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle \in \Delta$, we add a new transition rule $(p'_{\gamma'}, \gamma'', q)$ into $\mathcal{M}^*$;
  - If $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle \in \Delta$, we add a new transition rule (p', γ', q) into $\mathcal{M}^*$;
  - If $\langle p, \gamma \rangle \hookrightarrow \langle p', \epsilon \rangle \in \Delta$, we add a new transition rule (p', ϵ, q) into $\mathcal{M}^*$;
  - If $(p', \epsilon, p) \in \delta^*$, we add a new transition rule (p', γ, q) into $\mathcal{M}^*$.

The annotation function ρ is computed according to the following rules:

β_0: For every transition rule $t = (p, \gamma, q)$ in $\mathcal{M}^*$, let $\rho(t)(v) = \emptyset$ for any $v \in G \cup L$;

β_1: For every transition $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle \in \Delta$, every $t = (p, \gamma, q)$ and $t' = (p', \gamma', q)$ in $\mathcal{M}^*$, $\rho(t') = \rho(t') \uplus \theta$, where

- $\beta_{1.1}$: If the statement at the control point γ is an assignment $v = exp$, then, $\forall v' \in G \cup L \setminus \{v\}$: $\theta(v') = \rho(t)(v')$ and $\theta(v) = \{(v, \gamma)\} \cup \bigcup_{v' \in Var(exp)} \rho(t)(v')$;
- $\beta_{1.2}$: Otherwise, $\theta = \rho(t)$;

β_2: For every $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle \in \Delta$ s.t. $v = f(v_1, ..., v_m)$ is called at γ, every $t = (p, \gamma, q)$, $t' = (p', \gamma', p'_{\gamma'})$ and $t'' = (p'_{\gamma'}, \gamma'', q)$ in $\mathcal{M}^*$:

- $\beta_{2.1}$: $\rho(t') = \rho(t') \uplus \theta$, where $\forall v' \in G$: $\theta(v') = \rho(t)(v')$, $\forall v' \in L_f$: $\theta(v') = \emptyset$ and $\forall i \in \{1, ..., m\}$: $\theta(p_i) = \{(p_i, \gamma')\} \cup \rho(t)(v_i)$ (note that $p_1, ..., p_m$ are formal parameters of f);
- $\beta_{2.2}$: $\rho(t'') = \rho(t'') \uplus \theta'$, where $\forall v' \in L_{Proc(\gamma)}$: $\theta'(v') = \rho(t)(v')$, $\forall v' \in G$: $\theta'(v') = \emptyset$ and $\theta'(v) = \{(v, \gamma)\} \cup \{(R_f, \gamma'')\}$; (Note that $Proc(\gamma)$ denotes the procedure that contains the control point γ, i.e., where $v = f(v_1, ..., v_m)$ is called.)

β_3: For every $\langle p, \gamma \rangle \hookrightarrow \langle p', \epsilon \rangle \in \Delta$ s.t. *return* v is the statement at the control point γ, every $t = (p, \gamma, q)$ and $t' = (p', \epsilon, q)$ in $\mathcal{M}^*$, $\rho(t') = \rho(t') \uplus \theta$, where $\forall v' \in G \cup L_{Proc(\gamma)}$: $\theta(v') = \rho(t)(v')$ and $\theta(R_{Proc(\gamma)}) = \{(v, \gamma)\} \cup \rho(t)(v)$;

β_4: For every $t = (p, \epsilon, q)$, $t' = (q, \gamma, q')$ and $t'' = (p, \gamma, q')$ in $\mathcal{M}^*$, $\rho(t'') = \rho(t'') \uplus \theta$, where $\forall v \in G$: $\theta(v) = \rho(t)(v)$; $\forall v \in L_{Proc(\gamma)}$: $\theta(v) = \rho(t')(v)$; moreover, $\forall v \in G \cup L_{Proc(\gamma)}$ s.t. $(R_f, \gamma) \in \rho(t')(v)$ where $\rho(t)(R_f) \neq \emptyset$: $\theta(v) = \rho(t')(v) \cup \rho(t)(R_f)$.

Item β_0 initializes the annotation function ρ such that the value of any variable v at each control point does not depend on any variable at any location. Then, by iteratively applying Items $\beta_1, ..., \beta_4$ until there does not exist any transition t in $\mathcal{M}^*$ such that $\rho(t)$ can be updated, we can get the annotation function ρ such that for every configuration $\langle p, \gamma\omega\rangle \in L(\mathcal{M}^*)$ with $\gamma \in \Gamma$, $\langle p, \gamma\omega\rangle$ satisfies $v' = encode(v, l)$ iff there exists a transition rule $t = (p, \gamma_0, q_1) \in \delta^*$ such that $(v, l) \in \rho(t)(v')$ and $q_1 \xrightarrow{\omega'}_{\delta^*} q_f$ for some $\omega' \in \Gamma^*$. The intuition behind these rules is explained as follows.

Item β_1 expresses that if $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\rangle$ is a transition of the PDS, $t = (p, \gamma, q)$ and $t' = (p', \gamma', q)$ are in $\mathcal{M}^*$, then, the procedure depends on whether the statement is an assignment or not. If $v = exp$ is the assignment statement at the control point γ (Item $\beta_{1.1}$), we associate the set of pairs $\{(v, \gamma)\} \cup \bigcup_{v' \in Var(exp)} \rho(t)(v')$ to the variable v in the dependence function of the transition rule t'. This means that after executing the $v = exp$ statement, the value of v at the control point γ' depends on the variables in $Var(exp)$ and on itself at γ. Moreover, the values of the other variables v' remain the same as at γ. Therefore, the set of variables they depend on remain the same as at γ, i.e., $\theta(v') = \rho(t)(v')$. Item $\beta_{1.2}$ states that if the statement at the control point γ does not change the value of any variable, we associate the dependence function $\rho(t') \uplus \rho(t)$ to the transition rule t'.

Item β_2 states that if $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle$ is a transition rule of the PDS such that $v = f(v_1, ..., v_m)$ is called at γ, $t = (p, \gamma, q)$, $t' = (p', \gamma', p'_{\gamma'})$ and $t'' = (p'_{\gamma'}, \gamma'', q)$ are transition rules in $\mathcal{M}^*$, then, we update the dependence functions of t' in Item $\beta_{2.1}$ and t'' in Item $\beta_{2.2}$, respectively. Note that γ' denotes the entry point of the procedure f and γ'' is its corresponding return address. Item $\beta_{2.1}$ updates the dependence function of t' (i.e., the control point γ') by setting $\rho(t') = \rho(t') \uplus \theta$ such that (1) for every global variable $v' \in G$, $\theta(v') = \rho(t)(v')$ (i.e., at the entry point γ' of the function f, the set of variables that v' depends on remain the same as in γ), (2) for every local variable $v' \in L_f$ of the procedure f: $\theta(v') = \emptyset$; (3) for every parameter p_i of f, $\theta(p_i) = \{(p_i, \gamma')\} \cup \rho(t)(v_i)$, since according to parameter passing p_i equals v_i and p_i at γ' also depends on its value at γ'. Item $\beta_{2.2}$ updates the dependence function of t'' (i.e., the control point γ'') by setting $\rho(t'') = \rho(t'') \uplus \theta'$ such that (1) for every local variable $v' \in L_{Proc(\gamma)}$: $\theta'(v') = \rho(t)(v')$, this records the set of variables on which the local variables of the procedure $Proc(\gamma)$ at the caller-site depend on. This information will be used when the procedure f returns, i.e., at the control point γ'', see Item β_4; (2) for every global variable $v' \in G$: $\theta'(v') = \emptyset$ (since global variables may be changed in the procedure f, we update these global variables when f returns, see Item β_4); (3) the variable v is associated with the specific variable R_f and control location γ'' which denotes that v depends on R_f at γ'' and the value of v at γ' depends on its value at γ i.e., $\theta'(v) = \{(v, \gamma)\} \cup \{(R_f, \gamma'')\}$. R_f will be replaced by the real return value of f when f returns, see Item β_4.

Item β_3 expresses that if $\langle p, \gamma\rangle \hookrightarrow \langle p', \epsilon\rangle$ is a transition rule of the PDS such that *return* v is the statement at the control point γ (w.l.o.g., we assume that each function will return a value), and $t = (p, \gamma, q)$ and $t' = (p', \epsilon, q)$ are transition rules in $\mathcal{M}^*$, then, we update the annotation function of the transition t' by setting $\rho(t') = \rho(t') \uplus \theta$ such that for every variable $v' \in G \cup L_{Proc(\gamma)}$, $\theta(v') = \rho(t)(v')$ (since the values of these variables remain the same as in γ), and since at this point the variable $R_{Proc(\gamma)}$ denoting the return value of the procedure $Proc(\gamma)$ is instantiated with v, it depends on the set of variables

that v depends on at γ and on itself at γ. The transition rule $t' = (p', \epsilon, q)$ will be used in Item β_4 to pass the return value to the caller-side.

Item β_4 states that if $t = (p, \epsilon, q)$ denoting the return of a procedure f (see Item β_3), $t' = (q, \gamma, q')$ denoting that γ is the return address of the procedure f, and $t'' = (p, \gamma, q')$ denoting that the control point of the program is at the return address γ, are transition rules in $\mathcal{M}^*$, then, we update the annotation function of the transition t'' by setting $\rho(t'') = \rho(t'') \uplus \theta$ such that (1) for every global variable $v \in G$: $\theta(v) = \rho(t)(v)$ (i.e. at the return address γ, the program should use the values of the global variables of the procedure f); (2) for every local variable $v \in L_{Proc(\gamma)}$: $\theta(v) = \rho(t')(v)$ (i.e. the local variables of the procedure $Proc(\gamma)$ depend on the same set of variables at the caller-site in which the function f is called); (3) for every variable $v \in G \cup L_{Proc(\gamma)}$ that depends on the specific variable R_f (i.e. the return value of the procedure f) at γ: $\theta(v) = \rho(t')(v) \cup \rho(t)(R_f)$, since the variable v at γ depends on the same set of variables as R_f. Intuitively, the dependence function of the transition rule t' is updated in Item $\beta_{2.2}$ when a function call is made, thus, $\rho(t')$ records the sets of variables and locations that the local variables of $Proc(\gamma)$ depend on at the caller-side. The dependence function of t is updated in Item β_3 when the procedure f returns, this implies that $\rho(t)$ records the sets of variables and locations that the global variables and the return value R_f depend on at the return point. The transition rule t'' denotes that the control point is at the return address γ, thus, the update θ of the transition rule t'' uses the values of the global variables and R_f in $\rho(t)$ and uses the values of the local variables of $Proc(\gamma)$ in $\rho(t')$.

Complexity: Since the number of variables is bounded, the number of dependence functions is also bounded, at most $\mathbf{O}(|G| \cdot |L| \cdot 2^{|G|\cdot|L|\cdot|\Gamma|})$. The number of transition rules of $\mathcal{M}^*$ is at most $\mathbf{O}((|P| + k) \cdot |P| \cdot |\Delta|)$ where k is the number of pairs $(p', \gamma') \in P \times \Gamma$ such that $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle \in \Delta$ for some $p \in P, \gamma, \gamma'' \in \Gamma$. Then, we can get ρ and $\mathcal{M}^*$ in time $\mathbf{O}((|P| + k) \cdot |P| \cdot |\Delta| \cdot |G| \cdot |L| \cdot 2^{|G|\cdot|L|\times|\Gamma|})$.

Theorem 3. *Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$ modeling a given application, we can compute a MA $\mathcal{M}^*$ and an annotation function ρ in time $\mathbf{O}((|P| + k) \cdot |P| \cdot |\Delta| \cdot |G| \cdot |L| \cdot 2^{|G|\cdot|L|})$, such that for every $\langle p, \gamma\rangle \in P \times \Gamma$, every $\omega \in \Gamma^*$: $\langle p, \gamma\omega\rangle$ satisfies $v' = encode(v, l)$ iff there exists a transition rule $t = (p, \gamma, q) \in \delta^*$ such that $(v, l) \in \rho(t)(v')$ and $q \xrightarrow{\omega}_{\delta^*} q_f$.*

5.2 SCTPL and SLTPL Model-Checking for Android Applications

By Theorem 3, we can determine each predicate of the form $v' = encode(v, l)$ from $\mathcal{M}^*$ and ρ, then, we can obtain the labeling function λ as follows: for every function call $v = f(v_1, ..., v_m)$, we let $\lambda(v = f(v_1, ..., v_m)) = \{\langle p, \gamma\omega\rangle \mid \omega \in \Gamma^*$ such that the call $v = f(v_1, ..., v_m)$ is made at the control point $\gamma\}$; for every predicate $Loc(l)$, $\lambda(Loc(l)) = \{\langle p, \gamma\omega\rangle \mid \omega \in \Gamma^* \wedge \gamma = l\}$; for every predicate $v' = encode(v, l)$, $\lambda(v' = encode(v, l)) = \{\langle p, \gamma\omega\rangle \mid t = (p, \gamma, q) \in \delta^* \wedge (v, l) \in \rho(t)(v') \wedge q \xrightarrow{\omega'}_{\delta^*} q_f\}$ which is a regular set of configurations. By applying Theorems 1 and 2, we can get the following theorem.

Theorem 4. *Given a PDS modeling an Android application and a (malicious) behavior expressed in SCTPL/SLTPL formula, whether the PDS satisfies this behavior or not is decidable.*

Table 2. Results of checking information-leak formulas on the malicious applications

		Private Data													$\mathbf{Sum}_2$
		Location	**IMEI**	**IMSI**	**IMSI/SV**	**PN**	**ISOC**	**IPN**	**AT**	**CC**	**MD**	**SDC**	**WiFi**	**Intent**	
Leak ways	**TextMessage**	2	74	20	0	0	0	0	0	3	0	0	0	1	100
	Log File	315	278	179	0	33	178	26	0	267	0	199	52	275	1802
	Network	138	345	165	9	67	11	23	0	82	0	163	10	439	1443
	File writer	90	424	219	18	27	19	5	0	61	0	331	2	14	1210
	Intent	105	1	0	0	0	0	12	0	7	0	167	5	9	306
$\mathbf{Sum}_1$		348	685	480	18	114	204	48	0	288	0	352	52	529	
Avg. Time(s)		5.79	3.29	3.16	3.39	2.64	8.07	5.69	0	4.77	0	3.97	7.10	5.57	
Avg. Mem(MB)		76.6	63.8	56.3	50.1	48.3	68.8	81.1	0	61.4	0	67.8	61.5	63.8	

6 Experiments

We implemented a **model builder** based on the tool Smali, a disassembler for Android applications. Given an Android application as an *app* file which contains the application's Dalvik code, **model builder** automatically outputs a labeling function λ and a PDS modeling the application. We use the model-checking algorithms of [26, 27] to check whether the PDS model satisfies a given formula describing Android applications' (malicious) behaviors. We applied our tool to check 1331 applications which consists of 1260 confirmed real malwares from the dataset of [29], and 71 applications from the Android Compatibility Test Suite (CTS) [3] considered as benign applications. The size of malwares ranges from 13 KB to 15022 KB. The total size is 1.5 GB. While the size of CTS applications ranges from 2.7 KB to 26748 KB and its total size is 56.8 MB. We checked these applications against all the formulas presented in this paper. The analysis of each application costs only few seconds time and MB memory. This implies that our techniques are efficient and scalable. Our tool was able to detect all these malwares and several previously unknown malicious behaviors in the applications from CTS.

6.1 Information-Leak Android Applications

Table 2 gives the result of checking applications against information-leak formulas. **TextMessage**, **Log File**, **Network**, **File writer** and **Intent** denote different leaking ways that the private data can leak via text messages, log files, network connections, files and *Intent* object, respectively. **Location**, **IMEI**, **IMSI**, **IMSI/SV**, **PN**, **ISOC**, **IPN**, **AT**, **CC**, **MD**, **SDC**, **WiFi** and **Intent** are the private data we considered, denoting the location data, IMEI id, IMSI id, IMSI/SV id, phone number, Iso country code, incoming phone number, authentication token, contact or calendar data, mediate data, SD card data, WiFi connection information of the phone and the data stored in an Intent object, respectively. Our tool can check all the information-leak formulas for each application at the same time. Each cell in Table 2 except the rows **Avg. Time(s)**, **Avg. Mem(MB)**, $\mathbf{Sum}_1$ and the column $\mathbf{Sum}_2$, gives the number of applications that leak the private data indicated by the column title via the (way) approach indicated by the row title. For instance, there are 345 applications in the benchmark leaking the IMEI of the phone

[3] `http://developer.android.com`

via network connections. The $\mathbf{Sum}_1$ row (resp. $\mathbf{Sum}_2$ column) shows the total number of applications that leak the private data indicated by the column title (resp. use the leaking approach indicated by the row title). The **Avg. Time(s)** (resp. **Avg. Mem(MB)**) row gives the average of time (resp. memory) consumption in seconds (resp. MB) used to detect all the applications that leak the private data indicated by the column title.

As shown in Table 2, 685 applications leaks the IMEI of the phone, most of them are leaked via Log files, files and networks. No application in our experiment leaks media data (MD) and authentication token (AT). The detection of these applications costs only several seconds. This implies that our techniques are efficient and scalable.

We checked all the benign programs from Android CTS against all the information leak formulas using only 2569 seconds. The average memory consumption is 13.6 MB. Our tool reports that there are ten benign programs leaking private data, 8 of them have the corresponding permissions which will inform users the use of the private data, while the other two applications (*CtsTelephonyTestCases* and *CtsWidgetTestCases*) do not have permissions to access the private data, i.e., the users do not know the use of the private data. *CtsTelephonyTestCases* accesses *WiFi connection information* by calling the method *getConnectionInfo* of the class *WifiManager* and sends the information to other applications by *Intent* object. *CtsTelephonyTestCases* accesses *Contact and Calender data* by calling the *query* of the class *ContentResolver* and writes the information into a log file.

6.2 Checking the Other Malicious Behaviors

We applied our tool to check the benchmark against the other SCTPL/SLTPL formulas shown in Section 4.4. Table 3 depicts the results of checking all the malicious applications. The **Number of Apps** row shows the number of applications that satisfy the corresponding formula indicated by the column title. The **Avg. Time(s)** (resp. **Avg. Mem(MB)**) row gives the average of time (resp. memory) consumption in seconds (resp. MB) used to detect all the applications that satisfy the corresponding formula, where the time consumption is the sum of the time for computing the MA $\mathcal{M}^*$ and the annotation function ρ and for model-checking. The memory consumption is the maximum of the memory for computing the MA $\mathcal{M}^*$ and the annotation function ρ and for model-checking. From Table 3, we can see that malicious applications rarely take pictures or record videos without users' knowledge. But, many malicious applications executes dynamically loaded codes and harvest installed applications.

Table 3. Results of model-checking the malicious applications

	Ψ_{dd}	Ψ_{bp}	Ψ_{bv}	Ψ_{nc}	Ψ_{dc}	Ψ_{hi}
Number of Apps	491	0	1	679	185	793
Avg. Time(s)	40.04	0	21.77	16.44	11.8	17.51
Avg. Mem(MB)	86.3	0	59	41.1	23.8	78.9

The analysis of all the benign programs against all the SCTPL/SLTPL formulas (excepting information leak formulas) costs 3611.73 seconds. The average of memory consumption is 10.4 MB. 5 applications execute native codes, 2 applications record videos without the users' knowledge and 1 application harvests installed applications. During the analysis of benign programs, our tool automatically avoids to apply model-checking

on an application against a SCTPL/SLTPL formula if no function of in SCTPL/SLTPL formula is called. This improves the efficiency of our tool.

7 Related Work

Many works such as [1, 8, 9, 13, 14, 16, 19] use dynamic and/or static data flow analysis to analyze Android malwares. However, these works consider only information-leak malwares, and do not consider more complicated malicious behaviors. [30] aims to mainly analyze known Android malwares and needs samples to extract behavioral signatures. However, the signature-based techniques can be easily gotten around by malware writers. [9] static analyzes Android applications by translating them (Dalvik codes) into Java source codes and applying existing static analyzers of Java programs. However, as we discussed in the introduction, known reverse engineering tools, such as dex2jar, ded [9] and Dare [21], fail in some cases and it is also possible for malicious developers to write malicious codes at the Dalvik bytecode level that makes the application hard to be retargeted.

[21] proposes a more precise tool translating Dalvik codes into Java. However, the resulting Java source codes may miss some malicious behaviors. In this work, we propose an efficient and automatic approach that directly analyzes Android Dalvik codes. Our approach can analyze information-leak malwares and other more complicated (malicious) behaviors beyond information-leaks.

[17] introduces CTPL to specify malicious behaviors. SCTPL is an extension of CTPL with predicates over the stack [25, 26]. SLTPL is first introduced in [27], to specify malicious behaviors of executable programs. [17, 25–27] do not consider Android malware specifications and cannot be applied to check Android malwares in a precise manner. Indeed, for Android applications, we need predicates of the form $y = encode(x, l)$ which cannot be determined in [17, 25–27]. Moreover, the translation from Android applications to PDSs extends the standard translation from sequential programs to PDSs [11] and the translation used in [25–27] cannot be applied in the Android context due to existence of callback methods, the way these methods are called, and the absence of the main function. Furthermore, the Android malicious behaviors described in this work were not considered in [17, 25–27]. Model-checking and static analysis such as [5, 6, 17, 24] have been applied to detect non Android malwares.

The saturation procedure proposed in this work is an extension of the saturation procedure of [11]. However, [11] does not consider how to compute the annotation function ρ, i.e., the dependence relation between variables. [23] extends PDSs with a weight domain (called weighted PDSs) and their saturation procedure computes the weights of reachable configurations. [18] introduces an extension of weighted PDSs, called extended weighted PDSs, and shows how to compute the weights of reachable configurations by a kind of a saturation procedure. We could define the dependence relation of variables as a weight domain and apply the approaches of [18, 23] to compute the weights of reachable configurations, where each transition rule of the resulting MA is associated with a function over variables. Then, to decide whether the value of a variable depends on some variable at some control point, we have to compose several functions over the weight domain multiple times which can be avoided using our

approach. Indeed, we only need to query the transition rules of the MA $\mathcal{M}^*$ that are labeled by γ.

References

1. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Traon, Y.L., Octeau, D., McDaniel, P.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: PLDI (2014)
2. Beresford, A.R., Rice, A., Skehin, N., Sohan, R.: Mockdroid: Trading privacy for application functionality on smartphones. In: HotMobile, pp. 49–54 (2011)
3. Bouajjani, A., Esparza, J., Maler, O.: Reachability Analysis of Pushdown Automata: Application to Model Checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 135–150. Springer, Heidelberg (1997)
4. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.-R., Shastry, B.: Towards taming privilege-escalation attacks on android. In: NDSS (2012)
5. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. In: 12th USENIX Security Symposium, pp. 169–186 (2003)
6. Christodorescu, M., Jha, S., Seshia, S.A., Song, D.X., Bryant, R.E.: Semantics-aware malware detection. In: IEEE Symposium on Security and Privacy, pp. 32–46 (2005)
7. Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., Wallach, D.S.: Quire: Lightweight provenance for smart phone operating systems. In: USENIX Security Symposium (2011)
8. Enck, W., Gilbert, P., Gon Chun, B., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.: Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In: OSDI (2010)
9. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A study of android application security. In: USENIX Security Symposium (2011)
10. Enck, W., Ongtang, M., McDaniel, P.D.: On lightweight mobile phone application certification. In: ACM Conference on Computer and Communications Security (2009)
11. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithm for model checking pushdown systems. In: Emerson, E.A., Sistla, A.P. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 232–247. Springer, Heidelberg (2000)
12. Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E.: Permission re-delegation: Attacks and defenses. In: USENIX Security Symposium (2011)
13. Gibler, C., Crussell, J., Erickson, J., Chen, H.: AndroidLeaks: Automatically detecting potential privacy leaks in android applications on a large scale. In: Katzenbeisser, S., Weippl, E., Camp, L.J., Volkamer, M., Reiter, M., Zhang, X. (eds.) TRUST 2012. LNCS, vol. 7344, pp. 291–307. Springer, Heidelberg (2012)
14. Grace, M.C., Zhou, Y., Zhang, Q., Zou, S., Jiang, X.: Riskranker: Scalable and accurate zero-day android malware detection. In: MobiSys (2012)
15. Hornyack, P., Han, S., Jung, J., Schechter, S.E., Wetherall, D.: These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In: ACM CCS, pp. 639–652 (2011)
16. Kim, J., Yoon, Y., Yi, K., Shin, J.: Scandal: Static analyzer for detecting privacy leaks in android application. In: Mobile Security Technologies 2012 (2012)
17. Kinder, J., Katzenbeisser, S., Schallhart, C., Veith, H.: Detecting malicious code by model checking. In: Julisch, K., Kruegel, C. (eds.) DIMVA 2005. LNCS, vol. 3548, pp. 174–187. Springer, Heidelberg (2005)
18. Lal, A., Reps, T., Balakrishnan, G.: Extended weighted pushdown systems. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 434–448. Springer, Heidelberg (2005)

19. Mann, C., Starostin, A.: A framework for static detection of privacy leaks in android applications. In: SAC, pp. 1457–1462 (2012)
20. Nauman, M., Khan, S., Zhang, X.: Apex: Extending android permission model and enforcement with user-defined runtime constraints. In: ASIACCS, pp. 328–332 (2010)
21. Octeau, D., Jha, S., McDaniel, P.: Retargeting Android applications to Java bytecode. In: SIGSOFT FSE (2012)
22. Ongtang, M., McLaughlin, S.E., Enck, W., McDaniel, P.D.: Semantically rich application-centric security in android. In: ACSAC (2009)
23. Reps, T.W., Schwoon, S., Jha, S.: Weighted pushdown systems and their application to interprocedural dataflow analysis. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 189–213. Springer, Heidelberg (2003)
24. Singh, P.K., Lakhotia, A.: Static verification of worm and virus behavior in binary executables using model checking. In: IAW, pp. 298–300 (2003)
25. Song, F., Touili, T.: Efficient malware detection using model-checking. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 418–433. Springer, Heidelberg (2012)
26. Song, F., Touili, T.: Pushdown model checking for malware detection. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 110–125. Springer, Heidelberg (2012)
27. Song, F., Touili, T.: LTL model-checking for malware detection. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 416–431. Springer, Heidelberg (2013)
28. Song, F., Touili, T.: Model-checking for Android Malware Detection. Technical report, Shanghai Key Laboratory of Trustworthy Computing (2014), http://research.sei.ecnu.edu.cn/~song/publications/APLAS14.pdf
29. Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: IEEE Symposium on Security and Privacy, pp. 95–109 (2012)
30. Zhou, Y., Wang, Z., Zhou, W., Jiang, X.: Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In: NDSS (2012)