Accidental Sensitive Data Leaks Prevention via Formal Verification

Madalina G. Ciobanu[1], Fausto Fasano[1], Fabio Martinelli[2], Francesco Mercaldo[1,2] and Antonella Santone[1]

[1]*Department of Biosciences and Territory, University of Molise, Pesche (IS), Italy*

[2]*Istituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche, Pisa, Italy*

Keywords: Android, Security, Model Checking, Formal Methods, Privacy.

Abstract: Our mobile devices, if compared to their desktop counterpart, store a lot of sensitive and private information. Considering how easily permissions to sensitive and critical resources in the mobile environment are released, for example in Android, sometimes the developer unwittingly causes the leakage of sensitive information, endangering the privacy of users. Starting from these considerations, in this paper we propose a method aimed to automatically localise the code where there is the possibility of information leak. In a nutshell, we discuss a method aimed to check whether a sensitive information is processed in a way that violates specific rules. We employ code instrumentation to annotate sensitive data exploiting model checking technique. To show the effectiveness of the proposed method a case study is presented.

1 INTRODUCTION

Mobile applications are widely used in different sectors with billions of smartphone owners using mobile apps daily. The evolution of mobile software requires more attention, appropriate skills, and a better comprehension for the development, maintenance, and engineering of applications phases.

Nowadays, mobile apps need to seamlessly interact with back-end servers, which can be accomplished with numerous alterations and adjustments during the development phase (Harleen K. Flora, 2014). Smartphones, more than desktop and laptop computers, have lots of sensors that could increment usability but, there again, increment the overall complexity of the apps. With such sensors, indeed, it is possible to find position, rumor level, light, usage angle, movement and so on. Many of such sensors and peripherals produce sensitive data that must be managed following specific regulations. The GDPR[1] introduces penalties including important administrative fines [2] that can be imposed for any infringement of the Regulation, such as in case of personal data processing without user consent or transferring personal data to a non GDPR-compliant recipient. It is worth noting that such regulation applies to any subject that collects, uses and/or processes European citizen's personal data, regardless of whether the processing takes place in the Union or not.

To this aim, a growing attention is paid to the way apps collect, process, and transfer personal data to back-end servers. Most of this attention concerns malware detection (Fasano et al., 2019; Martinelli et al., 2017a; Martinelli et al., 2017b; Ciobanu et al., 2019a; Ciobanu et al., 2019b). However, even in case of a trusted app, whose behaviour is legitimate, the risk of an accidental sensitive data leak remains, as software developers are in charge of assessing the Regulation compliance. As an example scenario, a software house may decide to avoid its app to request write permission to shared internal storage in case of sensitive information, as well as to read card data or to mount local file systems when processing particular information, unless the user is aware of how the data is being processed. On the other hand, the same permissions may be legitimate when processing other non-sensitive entities or when storing aggregated data. Since the permission to read/write from providers/external storage as well as to send data through the internet is independent of the informa-

[1]The General Data Protection Regulation (GDPR) (EU) 2016/679 is a regulation in EU law on data protection and privacy.

[2]Fines can be up to 20 million euros, or 4% of the firm's worldwide annual revenue from the preceding financial year, whichever amount is higher. https://gdpr.eu/fines

Ciobanu, M., Fasano, F., Martinelli, F., Mercaldo, F. and Santone, A.
Accidental Sensitive Data Leaks Prevention via Formal Verification.
DOI: 10.5220/0009380608250834
In *Proceedings of the 6th International Conference on Information Systems Security and Privacy (ICISSP 2020)*, pages 825-834
ISBN: 978-989-758-399-5; ISSN: 2184-4356
Copyright © 2022 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

tion itself, there is no ready-to-use solution to check whether an app is accidentally leaking sensitive information by unsafely saving it to the device memory of sending it to the back-end, without the explicit consent of the user. Despite code inspection can be very effective to identify such situation, several developers underrate it in favor of testing (Scanniello et al., 2013) even when the former technique would be more adequate.

In this paper, we propose a method to formally check whether a sensitive information is processed in a way that violates specific rules. The proposed method uses code instrumentation to annotate sensitive data and is based on model checking technique using temporal logic formulae to check whether the sensible data processing can break some defined rules. Moreover, the approach assists the software engineer in the identification of the reason why the rule is not satisfied, providing relative traces from the simulation environment.

The rest of the paper proceeds as follows: In Section 2 background notions about model checking are provided and the proposed method to detect sensitive information leakage in Android environment is presented. In Section 3 a case study that illustrates the proposed method applied to the real scenario is presented. Related work is discussed in Section 4, while conclusion and future work are discussed in Section 5.

2 PROPOSED METHOD

In this section we briefly introduce background notions about model checking, in the next we describe the proposed method to detect sensitive data leakage in Android environment.

2.1 Model Checking and Mu-calculus Logic

Model checking is a formal method for determining if a model of a system satisfies a correctness specification (Clarke et al., 2001; Ceccarelli et al., 2014; Santone, 2002; Santone, 2011; Barbuti et al., 2005; Gradara et al., 2005). A model of a system consists of a Labelled Transition System (LTS). A specification or property is a logical formula. A model checker then accepts two inputs, an LTS and a temporal formula, and returns *true* if the system satisfies the formula and *false* otherwise.

An LTS comprises some number of states, with arcs between them labelled by activities of the system. An LTS is specified by:

- a set S of states;
- a set L of labels or actions;
- a set of transitions $T \subseteq S \times L \times S$.

Transitions are given as triples $(start, label, end)$.

With the aim to express proprieties of the system we consider the modal mu-calculus (Stirling, 1989) which is one of the most important logics in model checking.

The syntax of the mu-calculus is the following, where K ranges over sets of actions (i.e., $K \subseteq L$) and Z ranges over variables:

$$\varphi \quad ::= \quad \texttt{tt} \mid \texttt{ff} \mid Z \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid [K]\varphi \mid \langle K \rangle \varphi \mid \nu Z.\varphi \mid \mu Z.\varphi$$

A fixpoint formula may be either $\mu Z.\varphi$ or $\nu Z.\varphi$ where μZ and νZ *binds* free occurrences of Z in φ. An occurrence of Z is free if it is not within the scope of a binder μZ (resp. νZ). A formula is *closed* if it contains no free variables. $\mu Z.\varphi$ is the least fixpoint of the recursive equation $Z = \varphi$, while $\nu Z.\varphi$ is the greatest one. From now on we consider only closed formulae.

Scopes of fixpoint variables, free and bound variables, can be defined in the mu-calculus in analogy with variables of first order logic.

The satisfaction of a formula φ by a state s of a transition system is defined as follows:

- each state satisfies tt and no state satisfies ff;
- a state satisfies $\varphi_1 \vee \varphi_2$ ($\varphi_1 \wedge \varphi_2$) if it satisfies φ_1 or (and) φ_2. $[K]\,\varphi$ is satisfied by a state which, for every execution of an action in K, evolves to a state obeying φ. $\langle K \rangle\,\varphi$ is satisfied by a state which can evolve to a state obeying φ by performing an action in K.

For example, $\langle a \rangle\,\varphi$ denotes that there is an a-successor in which φ holds, while $[a]\,\varphi$ denotes that for all a-successors φ holds.

The precise definition of the satisfaction of a closed formula φ by a state s (written $s \models \varphi$) is given in Table 1.

We consider the CWB-NC[3] (Concurrency WorkBench of the New Century) as formal verification environment. It is one of the most popular environments for verifying systems. In the CWB-NC the verification of temporal logic formulae is based on model checking (Clarke et al., 2001).

2.2 Information Leakage Detection

In this section, we describe our approach aimed to detect possible sensitive information leakage in Android

[3] https://www3.cs.stonybrook.edu/~cwb/

Table 1: Satisfaction of a closed formula by a state.

$p \not\models \texttt{ff}$		
$p \models \texttt{tt}$		
$p \models \varphi \wedge \psi$	iff	$p \models \varphi$ and $p \models \psi$
$p \models \varphi \vee \psi$	iff	$p \models \varphi$ or $p \models \psi$
$p \models [K]_R\, \varphi$	iff	$\forall p'.\forall \alpha \in K.p \xrightarrow{\alpha}_{K \cup R} p'$ implies $p' \models \varphi$
$p \models \langle K \rangle_R\, \varphi$	iff	$\exists p'.\exists \alpha \in K.p \xrightarrow{\alpha}_{K \cup R} p'$ and $p' \models \varphi$
$p \models \nu Z.\varphi$	iff	$p \models \nu Z^n.\varphi$ for all n
$p \models \mu Z.\varphi$	iff	$p \models \mu Z^n.\varphi$ for some n

where:

- for each n, $\nu Z^n.\varphi$ and $\mu Z^n.\varphi$ are defined as:

$$\nu Z^0.\varphi = \texttt{tt} \qquad \mu Z^0.\varphi = \texttt{ff}$$
$$\nu Z^{n+1}.\varphi = \varphi[\nu Z^n.\varphi/Z] \qquad \mu Z^{n+1}.\varphi = \varphi[\mu Z^n.\varphi/Z]$$

where the notation $\varphi[\psi/Z]$ indicates the substitution of ψ for every free occurrence of the variable Z in φ.

applications. It is worth noting that the approach is not limited to the Android platform, but, in this paper, we will focus on this platform because of its broad diffusion, the easiness and multiplicity of ways data can be shared with other applications, and the availability of numerous source code repositories to assess the approach.

The proposed method, on one hand uses code instrumentation to provide domain expert knowledge and, on the other hand, models the Android application as a labelled transition system (LTS) capturing the behaviour of the app, and evaluating temporal properties directly on this LTS. Figure 1 shows the workflow of the proposed method.

The first step of the proposed method is the *Code instrumentation*, aiming at providing basics for the *Model checking* phase. In particular, two kind of annotations can be specified: (i) which data should be considered sensitive and (ii) where, in the app, the user is informed about the way her data are stored or processed. An example of the former instrumented code is shown in Figure 2. Here, the `mData` variable stores meta-data of a picture in the device photo gallery. Amongst the picture meta-data, information can be found, such as the geolocation, other EXIF data, user tags, and the picture rating that could be considered sensitive. As so, in our example, the app developer decided to annotate the variable, thus informing the verification tool that the aforementioned variable must be checked against the specified rules. An example of the latter instrumented code is shown in Figure 3. In this case, the annotation is used as a checkpoint to state where the user is informed and determine if she agreed to the sensitive data processing or not.

The second phase of the method consists in a formal verification of a set of rules. We focus here on a GDPR-compliance rule stating that personal data processing without user consent or transferring personal data to a non GDPR-compliant recipient is forbidden.

In order to describe mobile applications we adopted the Milner's Calculus of Communicating Systems (CCS) (Milner, 1989) language specification and express behavioural properties using mu-calculus branching temporal logic (Stirling, 1989). Similarly to previous works that adopted such methodology (Canfora et al., 2018), we developed a tool aimed to first generate a model in the CCS specification starting from the analyzed application source code where, for each instruction, we define a transformation function to map source code into CCS process specifications. Afterwards, we specify the set of properties we want to guarantee. In particular, codes described as CCS processes are first mapped to labelled transition systems and then verified with a model checker, i.e., the Concurrency Workbench of New Century (CWB-NC) (Cleaveland and Sims, 1996) verification environment.

In this work, we formulated logic rules to verify whether the value of a labelled variable (one of the annotated variable in the previous step of the approach) is accessed while saving data to the shared storage or sending data through the Internet without the explicit consent of the user. In

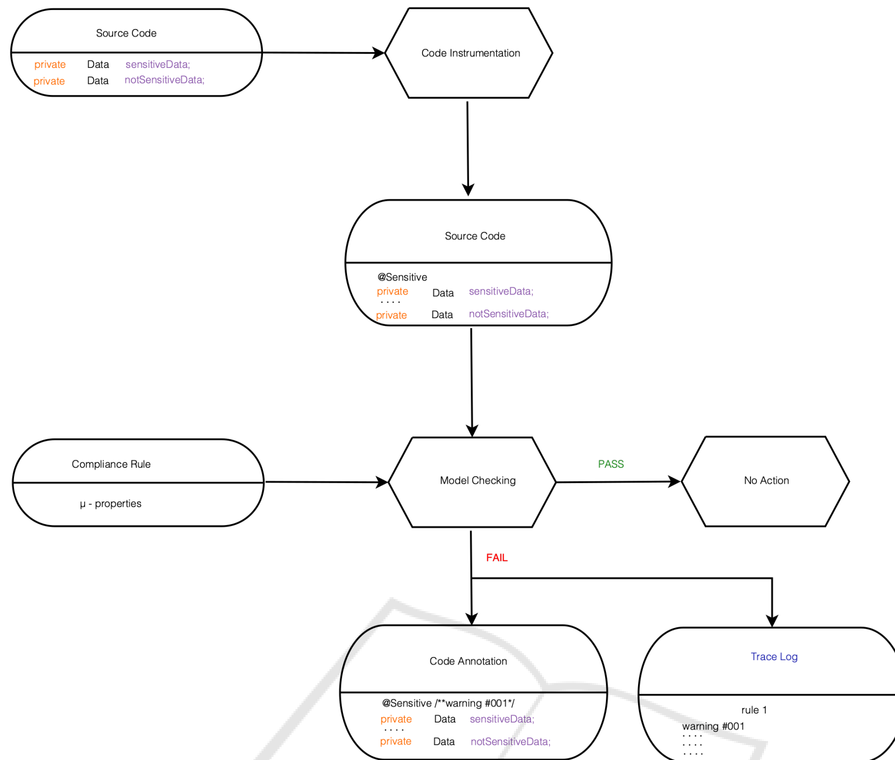

Figure 1: The proposed method the detect sensible data leakage.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

```
/**
 * r/w {@link IPhotoProperties} Implementation for android databse/contentprovider {@link ContentValues}.
 */
public class PhotoPropertiesMediaDBContentValues implements IPhotoProperties {

    /** meta-data of the photo*/
    @Sensitive
    private ContentValues mData;

    /** last modification date of the photo meta-data */
    private Date mXmpFileModifyDate;

    @Override
    public Double getLatitude() {
        return  mData.getAsDouble(FotoSql.SQL_COL_LAT);
    }

    @Override
    public Double getLongitude() {
        return  mData.getAsDouble(FotoSql.SQL_COL_LON);
    }
```

Figure 2: Code snippet showing sensitive data annotation for APhotoManager. [4]

this paper, in order to simplify the rules, we suppose that whenever a variable is accessed before the use of specific permission invocation (i.e., the *android.permission.INTERNET* request or the *android.permission.WRITE_EXTERNAL_STORAGE* request) there could be a data leakage. Moreover, the explicit consent of the user has been identified by annotating the relative code (see. Figure 3). To pass the CWB-NC model checker, the execution of the code must precede in time the access to the variable in the execution stack.

Whether at least one property is not verified, the proposed framework will mark the relative variable by appending a warning code to the corresponding annotation. Moreover, a *Trace Log* will be created

[4] A Photo Manager App available on F-Droid repository at https://f-droid.org/en/packages/de.k3b.android.androFotoFinder/

```
public void onCreate(Bundle savedInstanceState) {

    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_map_geo_picker);

    button = (Button) findViewById(R.id.cmd_photo);

    // add button listener
    button.setOnClickListener(new OnClickListener() {

        @Override
        public void onClick(View arg0) {
            AlertDialog.Builder alertDialogBuilder = new AlertDialog.Builder(context);
            alertDialogBuilder.setTitle("Confirm access to photo meta-data");
            alertDialogBuilder.setMessage("Your Picture meta-data will be saved to the Local Storage. Do you agree?");
            alertDialogBuilder.setCancelable(false);
            alertDialogBuilder.setPositiveButton("Yes", new DialogInterface.OnClickListener() {
                public void onClick(DialogInterface dialog, int id) {
                    @SensitiveGrant
                    // if this button is clicked
                    R.id.cmd_photo.saveData();
                }
            }).setNegativeButton("No", new DialogInterface.OnClickListener() {
                public void onClick(DialogInterface dialog, int id) {
                    @SensitiveDeny
                    // if this button is clicked, just close
                    // the dialog box and do nothing
                    dialog.cancel();
                }
            });

            // create alert dialog and show it
            AlertDialog alertDialog = alertDialogBuilder.create();
            alertDialog.show();
        }
    });
}
```

Figure 3: Code snippet showing user awareness enforcement.

that contains details about the rule that was not verified and detailed information about the reason why the compliance is not guaranteed (e.g., the trace that leads to a sensitive data leakage). In case all the rules are verified, there is no action to take.

Note that, apart from the code instrumentation, that is responsibility of the app developer, the rest of the process is automatic. In particular, the construction of the LTS is completely automatic. Furthermore, the model checker tool can be automated to continuously verify, e.g., in the background, the specified logic formulae on the formal model, and interact with the development environment when a rule cannot be verified because of changes to the source code of the application.

3 THE CASE STUDY

In this section, we present the temporal logic formulae aimed to assist the developer notifying the users about possible sensitive data leaks.

As case study, we consider the *A Photo Manager* app, an Android application to manage local photos [4]. Below we show three code snippet belonging to the *A Photo Manager* app.

In detail, we show three code snippets implementing behaviours that can conduct to leakage of sensitive information. We show the Java code snippets extracted by exploiting the *Bytecode Viewer* tool [5].

Figure 4 shows a code snippet related to the *equals* method of the *GeoLocation* belonging to the *com.draw.lang* package. This snippet accesses the current latitude and longitude (as evidenced by rows 9 and 11 in the Figure 4 snippet).

Figure 5 shows a code snippet related to the *getCacheDirectory* method of the *StorageUtils* class in the *com.nostra13.universalimageloader.utils* package. The snippet shown in Figure 5 is checking whether it is possible to access to the external storage (by invoking the *hasExternalStoragePermission* method). It can be of interest to highlight that on the Android external storage once a file is stored, all the installed application can read and write the resource without explicit consent to the user (Canfora et al., 2018).

The code snippet shown in Figure 6 is related to the *NetworkAvailablityCheck* class constructor belonging to the *org.osmdroid.titleprovider.modules* package. This snippet requires the permission to access to the network exploiting the *android.permission.ACCESS_NETWORK_STATE* permission: basically this permission allows applications to access information about networks, for this reason this request can be considered as a potential

[5] https://github.com/Konloch/bytecode-viewer

```
/*package com.draw.lang;
    class GeoLocation*/

    public boolean equals(Object var1) {
        if (this == var1){
            return true;
        }else if (var1 != null && this.getClass() == var1.getClass()){
            GeoLocation var2 = (GeoLocation)var1;
            if (Double.compare(var2_latitude, this._latitude != 0)){
                return false;
            } else {
                return Double.compare(var2._longitude, this._longitude) == 0;
            }
        } else {
            return false;
        }
    }

```

Figure 4: Code snippet accessing device latitude and longitude.

```
/*package com.nostra13.universalimageloader.utils;
    class StorageUtils*/

    public static File getCacheDirectory(Context var0, boolean var1){
        String var2;
        try{
            var2 = Environment.getExternalStorageState();
        } catch (NullPointerException var4) {
            var2 = "";
        } catch (IncompatibleClassChangeError var5){
            var2 = "";
        }
        File var3 = null;
        if (var1 && "mounted".equals(var2) && hasExternalStoragePermission(var0)) {
        var3 = getExternalCacherDir(var0);
        } else {
            var3 = null
        }
        File var7 = var3;
        if (var3 == null){
            var7 = var0.getCacheDir();
        }

        var3 = var7;
        if (var7 == null){
            StringBuilder var8 = new StringBuilder();
            var8.append("/data/data/");
            var8.append(var0.getPackageName());
            var8.append("/cache/");
        }
        return var3;
    }
```

Figure 5: Code snippet accessing external storage.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

information leak.

Table 2 shows the temporal logic formulae to detect whether the mobile developer previously informed the user before using a sensitive resources.

The temporal logic formulae are related to the detection of the following behaviours:

- with respect to the φ formula, related to the snippet in Figure 4, the formula is satisfied whether a log is invoked (with the aim to advise the user) *before* using the information about the current device localisation;
- with respect to the χ formula, related to the snippet shown in Figure 5, this formula is *true* whether an instance of the log is invoked *before* asking for external storage usage;
- with respect to the ψ formula, related to the snippet shown in Figure 6, the ψ temporal logic property is satisfied whether a log instance is required *before* the device is using information related to the network state.

Table 2: Temporal logic formulae for mobile secure programming verification.

$\varphi_1 = \nu X.[checkcastcomdrewlangGeoLocatio]\,\texttt{ff} \wedge [-checkcastcomdrewlangGeoLocation, pushlog]\,X$

$\varphi_2 = \nu X.[store]\,\texttt{ff} \wedge [-store, pushlog]\,X$

$\varphi_3 = \nu X.[load]\,\texttt{ff} \wedge [-load, pushlog]\,X$

$\varphi = \varphi_1 \wedge \varphi_2 \wedge \varphi_3$

$\chi_1 = \nu X.[push]\,\texttt{ff} \wedge [-push, pushlog]\,X$

$\chi_2 = \nu X.[invokegetCacheDirectory]\,\texttt{ff} \wedge [-invokegetCacheDirectory, pushlog]\,X$

$\chi_3 = \nu X.[invokegetExternalStorageState]\,\texttt{ff} \wedge [-invokegetExternalStorageState, pushlog]\,X$

$\chi_4 = \nu X.[pushmounted]\,\texttt{ff} \wedge [-pushmounted, pushlog]\,X$

$\chi_5 = \nu X.[invokehasExternalStoragePermission]\,\texttt{ff} \wedge [-invokehasExternalStoragePermission, pushlog]\,X$

$\chi_6 = \nu X.[invokegetExternalCacheDir]\,\texttt{ff} \wedge [-invokegetExternalCacheDir, pushlog]\,X$

$\chi_7 = \nu X.[invokegetCacheDir]\,\texttt{ff} \wedge [-invokegetCacheDir, pushlog]\,X$

$\chi = \chi_1 \wedge \chi_2 \wedge \chi_3 \wedge \chi_4 \wedge \chi_5 \wedge \chi_6 \wedge \chi_7$

$\psi_1 = \nu X.[invokegetSystemService]\,\texttt{ff} \wedge [-invokegetSystemService, pushlog]\,X$

$\psi_2 = \nu X.[checkcastandroidnetConnectivityManager]\,\texttt{ff} \wedge$
$[-checkcastandroidnetConnectivityManager, pushlog]\,X$

$\psi_3 = \nu X.[invokegetPackageManager]\,\texttt{ff} \wedge [-invokegetPackageManager, pushlog]\,X$

$\psi_4 = \nu X.[pushandroidpermissionACCESSNETWORKSTATE]\,\texttt{ff} \wedge$
$[-pushandroidpermissionACCESSNETWORKSTATE, pushlog]\,X$

$\psi_5 = \nu X.[invokegetPackageName]\,\texttt{ff} \wedge [-invokegetPackageName, pushlog]\,X$

$\psi_6 = \nu X.[invokecheckPermission]\,\texttt{ff} \wedge [-invokecheckPermission, pushlog]\,X$

$\psi = \psi_1 \wedge \psi_2 \wedge \psi_3 \wedge \psi_4 \wedge \psi_5 \wedge \psi_6$

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

```
1  /* package org.osmdroid.tileprovider.modules;
2     class NetworkAvailabilityCheck */
3
4     public NetworkAvailabilityCheck(Context var1) {
5         this.mConnectionManager =
6             (ConnectivityManager) var1.getSystemService("connectivity");
7         this.mIsX(& = "Android-X86".equalsIgnoreCase(Build.BRAND);
8         boolean var2;
9         if (var1.getPackageManager().
10            checkPermission("android.permission.ACCESS_NETWORK_STATE",
11                var1.getPackageName()) == 0) {
12            var2 = true;
13        } else {
14            var2 = false;
15        }
16        this.mHasNetworkStatePermission = var2;
17    }
```

Figure 6: Code snippet requiring the network access.

Clearly in the code snippet shown in Figures 4, 5 and 6 these formulae are not satisfied: as a matter of fact there is no log invocation before the usage of the sensitive instructions, symptomatic that the user is not advised.

With the aim to help the developer to understand the reason why a certain formula is not satisfied, we show in Figure 7 the trace generated from the *CWB-NC* simulation environment. In detail, we are considering the model generated starting from the code snippet in Figure 4.

In this case the φ formula is not satisfied because it never happens that before a *checkcastcomdrewlangGeoLocation*, a *store* and a *load* action there is a *log* action: in fact, as shown from Figure 7, the *pushlog* action is not present, while the *checkcastcomdrewlangGeoLocation*, the *store* and the *load* are present (in Figure 7 are highlighted). The φ formula is satisfied whether a *pushlog* action is present *before* the highlighted actions. In this way the developer is able to localise the exact point in the source code where to invoke the log action (i.e., the exact point where the user should be notified).

4 RELATED WORK

Several studies in current state of the art literature are mainly focused on mobile malware detection (Chen et al., 2016; Suarez-Tangil et al., 2017; Duc and Giang, 2018). These works are mainly exploiting machine learning techniques by extracting distinctive features from samples under analysis to discriminate between malicious applications and trusted ones. Contrarily, in this paper we investigate an automatised method aimed to detect possible sensitive information leakage by providing an useful tool for the developer to inform the user.

Shan et al. in (Shan et al., 2018) investigate about self-hiding behaviours (SHB), e.g. hiding the app, hiding app resources, blocking calls, deleting call records, or blocking and deleting text messages. First of all the authors provide an in-deep characterization of SHB, then they present a suite of static analyses to detect such behaviour. They define a set of detection rules able to catch SHB. They test their approach against more than 9,000 Android applications.

Dynamic taint analysis is a method to analyze executable files by tracing information flow (Kim et al., 2014; Dalton et al., 2010). This approach has been applied to detect sensitive information leaking of network servers (Li et al., 2014). Differently from us, besides being focused on network servers attacked by malwares or hackers, sensitive data tagging is automatic, thus not using the domain expert knowledge.

Taint analysis is also considered by FlowDroid (Arzt et al., 2014), a tool focused on the Android application life-cycle and callback methods designed with the aim to reduce missed leaks and false positives. Authors provide also an Android-specific benchmark suite i.e., DROIDBENCH, to evaluate taint analysis tools focused on Android information leakage. Also the Leakminer tool (Yang and Yang, 2012) considers taint analysis for detecting sensitive information exfiltration in Android.

All of the aforementioned approaches are mainly focused on the identification of data leaking from the user perspective. In fact, as emerging from the current state-of-the-art discussion, the following proposal represents the first attempt to provide a tool for the developer to avoid the (unaware) usage of sensitive resource without notifying the user.

```
cwb-nc> sim COREEFILETESTCLASSCOMDREWLANGGEOLOCATIONpublicbooleanequalsObjectargUuno24
COREEFILETESTCLASSCOMDREWLANGGEOLOCATIONpublicbooleanequalsObjectargUuno24
1: -- load --> COREEFILETESTCLASSCOMDREWLANGGEOLOCATIONpublicbooleanequalsObjectargUuno25
cwb-nc-sim> 1
COREEFILETESTCLASSCOMDREWLANGGEOLOCATIONpublicbooleanequalsObjectargUuno25
1: -- checkcastcomdrewlangGeoLocation --> COREEFILETESTCLASSCOMDREWLANGGEOLOCATIONpublicbooleanequalsObjectargUuno28
cwb-nc-sim> 1
COREEFILETESTCLASSCOMDREWLANGGEOLOCATIONpublicbooleanequalsObjectargUuno28
1: -- store --> COREEFILETESTCLASSCOMDREWLANGGEOLOCATIONpublicbooleanequalsObjectargUuno29
cwb-nc-sim> 1
COREEFILETESTCLASSCOMDREWLANGGEOLOCATIONpublicbooleanequalsObjectargUuno29
1: -- load --> COREEFILETESTCLASSCOMDREWLANGGEOLOCATIONpublicbooleanequalsObjectargUuno30
cwb-nc-sim> 1
COREEFILETESTCLASSCOMDREWLANGGEOLOCATIONpublicbooleanequalsObjectargUuno30
1: -- t --> COREEFILETESTCLASSCOMDREWLANGGEOLOCATIONpublicbooleanequalsObjectargUuno33
cwb-nc-sim> 1
COREEFILETESTCLASSCOMDREWLANGGEOLOCATIONpublicbooleanequalsObjectargUuno33
1: -- load --> COREEFILETESTCLASSCOMDREWLANGGEOLOCATIONpublicbooleanequalsObjectargUuno34
cwb-nc-sim> 1
COREEFILETESTCLASSCOMDREWLANGGEOLOCATIONpublicbooleanequalsObjectargUuno34
1: -- t --> COREEFILETESTCLASSCOMDREWLANGGEOLOCATIONpublicbooleanequalsObjectargUuno37
cwb-nc-sim> 1
COREEFILETESTCLASSCOMDREWLANGGEOLOCATIONpublicbooleanequalsObjectargUuno37
1: -- invokecompare --> COREEFILETESTCLASSCOMDREWLANGGEOLOCATIONpublicbooleanequalsObjectargUuno40
cwb-nc-sim> 1
```

Figure 7: Trace generated from the simulation environment.

5 CONCLUSION AND FUTURE WORK

In this paper we propose a method to assess security related properties of Android applications. This is a current topic, especially in the context of GDPR regulation compliance. Indeed, the software developer may be answerable to provide access to sensitive information without the explicit user consent as well as for transferring personal data towards a non GDPR-compliant recipient. In the Android environment, data can be transferred through several channels, e.g., via the network connection or by storing them in shared memory that can be accessed by other applications. In these cases, there is the possibility to infringe the regulation and cause data leakage towards unauthorized subjects or non compliant recipients. Despite this could be considered a malicious behaviour typical from spyware and other similar malwares, there could be the possibility that the leakage is merely due to shallowness or bad programming practices, such as using an unsafe connection or a shared data store.

To overcome this issue and support the software engineer during the development and verification phases, we propose a semi-automatic method based on code instrumentation and model checking techniques. In our approach, the software engineer is in charge of identifying the sensitive information her app processes and the methods she defined to inform the user about the sensitive data processing or transfer. This can be done using Java custom annotations or logging instructions. Once the code is instrumented, formal methods are applied to check predefined rules aimed at verifying if there exist execution threads bringing about sensitive information leaks.

We also presented how the proposed method can be applied to an existing application to identify possible information leaks and improve the software security by fixing the flaw. The proposed method can be easily extended to other programming language and platforms provided that the related translator from the source to the CCS specification is available.

As future work, we plan to evaluate the method by applying it to several open source Android applications to identify possible sensitive information leaks and understand how much widespread are similar bad programming practices. Finally, in case we observe a broad diffusion of unchecked sensitive data leaks, we intend to develop a tool that automatically fix some of the most common regulation infringements by injecting configurable snippets directly in the application's source code. This would provide the developers with a tool to significantly improve their application compliance to specific regulations, with a limited effort.

ACKNOWLEDGMENTS

This work has been partially supported by MIUR - SecureOpenNets and EU SPARTA and CyberSANE projects, the Formal Methods for IT Security Lab[6],

[6]https://dipbioter.unimol.it/ricerca/laboratori/metodi-formali-per-la-sicurezza-informatica/

and the MOSAIC Research Center[7] at the University of Molise.

REFERENCES

Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. (2014). Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM SIGPLAN Notices*, 49(6):259–269.

Barbuti, R., De Francesco, N., Santone, A., and Vaglini, G. (2005). Reduced models for efficient ccs verification. *Formal Methods in System Design*, 26(3):319–350.

Canfora, G., Martinelli, F., Mercaldo, F., Nardone, V., Santone, A., and Visaggio, C. A. (2018). Leila: formal tool for identifying mobile malicious behaviour. *IEEE Transactions on Software Engineering*.

Ceccarelli, M., Cerulo, L., and Santone, A. (2014). De novo reconstruction of gene regulatory networks from time series data, an approach based on formal methods. *Methods*, 69(3):298–305. cited By 10.

Chen, S., Xue, M., Tang, Z., Xu, L., and Zhu, H. (2016). Stormdroid: A streaminglized machine learning-based system for detecting android malware. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 377–388. ACM.

Ciobanu, M. G., Fasano, F., Martinelli, F., Mercaldo, F., and Santone, A. (2019a). A data life cycle modeling proposal by means of formal methods. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 670–672.

Ciobanu, M. G., Fasano, F., Martinelli, F., Mercaldo, F., and Santone, A. (2019b). Model checking for data anomaly detection. *Procedia Computer Science*, 159:1277–1286.

Clarke, E. M., Grumberg, O., and Peled, D. (2001). *Model checking*. MIT Press.

Cleaveland, R. and Sims, S. (1996). The ncsu concurrency workbench. In Alur, R. and Henzinger, T. A., editors, *CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 394–397. Springer.

Dalton, M., Kannan, H., and Kozyrakis, C. (2010). Tainting is not pointless. *ACM SIGOPS Operating Systems Review*, 44(2):88–92.

Duc, N. V. and Giang, P. T. (2018). Nadm: Neural network for android detection malware. In *Proceedings of the Ninth International Symposium on Information and Communication Technology*, pages 449–455. ACM.

Fasano, F., Martinelli, F., Mercaldo, F., and Santone, A. (2019). Cascade learning for mobile malware families detection through quality and android metrics. In *International Joint Conference on Neural Networks, IJCNN 2019 Budapest, Hungary, July 14-19, 2019*, pages 1–10.

[7] https://dipbioter.unimol.it/ricerca/laboratori/centro-di-ricerca-mosaic/

Gradara, S., Santone, A., and Villani, M. L. (2005). Using heuristic search for finding deadlocks in concurrent systems. *Information and Computation*, 202(2):191–226.

Harleen K. Flora, Xiaofeng Wang, S. C. (2014). Adopting an agile approach for the development of mobile applications. *Journal of Computer Applications*, 94:43–50.

Kim, J., Kim, T., and Im, E. G. (2014). Survey of dynamic taint analysis. In *2014 4th IEEE International Conference on Network Infrastructure and Digital Content*, pages 269–272.

Li, W., Yan, Y., Tu, H., and Xu, J. (2014). A dynamic taint tracking based method to detect sensitive information leaking. In *The 16th Asia-Pacific Network Operations and Management Symposium*, pages 1–4.

Martinelli, F., Marulli, F., and Mercaldo, F. (2017a). Evaluating convolutional neural network for effective mobile malware detection. *Procedia computer science*, 112:2372–2381.

Martinelli, F., Mercaldo, F., and Saracino, A. (2017b). Bridemaid: An hybrid tool for accurate detection of android malware. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 899–901. ACM.

Milner, R. (1989). *Communication and concurrency*. PHI Series in computer science. Prentice Hall.

Santone, A. (2002). Automatic verification of concurrent systems using a formula-based compositional approach. *Acta Informatica*, 38(8):531–564.

Santone, A. (2011). Clone detection through process algebras and java bytecode. In *IWSC*, pages 73–74. Citeseer.

Scanniello, G., Fasano, F., Lucia, A. D., and Tortora, G. (2013). Does software error/defect identification matter in the italian industry? *IET Software*, 7(2).

Shan, Z., Neamtiu, I., and Samuel, R. (2018). Self-hiding behavior in android apps: detection and characterization. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 728–739.

Stirling, C. (1989). An introduction to modal and temporal logics for ccs. In Yonezawa, A. and Ito, T., editors, *Concurrency: Theory, Language, And Architecture*, volume 491 of *LNCS*, pages 2–20. Springer.

Suarez-Tangil, G., Dash, S. K., Ahmadi, M., Kinder, J., Giacinto, G., and Cavallaro, L. (2017). Droidsieve: Fast and accurate classification of obfuscated android malware. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 309–320. ACM.

Yang, Z. and Yang, M. (2012). Leakminer: Detect information leakage on android with static taint analysis. In *2012 Third World Congress on Software Engineering*, pages 101–104. IEEE.