Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

EspyDroid+: Precise reflection analysis of android apps

Jyoti Gajrani[a,*], Umang Agarwal[b], Vijay Laxmi[a], Bruhadeshwar Bezawada[c], Manoj Singh Gaur[d], Meenakshi Tripathi[a], Akka Zemmari[e]

[a] *Malaviya National Institute of Technology Jaipur (MNIT Jaipur), India*
[b] *Govt. Engineering College Ajmer, India*
[c] *Mahindra École Centrale, Hyderabad, India*
[d] *Indian Institute of Technology Jammu, India*
[e] *LaBRI, Bordeaux INP, University of Bordeaux, CNRS, France*

ARTICLE INFO

Article history:
Received 8 August 2019
Revised 18 November 2019
Accepted 13 December 2019
Available online 13 December 2019

Keywords:
Instrumentation
Reflection
Run-time
Malware
Static
Dynamic
Privacy leaks
Android

ABSTRACT

Malicious smartphone apps use reflection APIs to exfiltrate user data and steal personal information. These malware use reflection along with parameter obfuscation and encryption to evade detection by static analysis. Dynamic analysis is a possible approach to detect such run-time malicious behavior. However, dynamic analysis of a software, usually, results in the exploration of a large, potentially exponential, number of program branches. Many of these program paths are not useful to analyze the reflection APIs, and significantly affect the efficiency of the dynamic analysis. In this paper, we propose a hybrid analysis approach named EspyDroid+[1] that overcomes the drawbacks of static analysis in analyzing the obfuscated and run-time dependent parameters of reflection APIs. EspyDroid+ incorporates *Reflection Guided Static Slicing (RGSS)*, an efficient approach to deal with exploration of large number of program paths by pruning irrelevant program paths and ensures that the resultant paths get executed during the subsequent dynamic analysis. We observed that EspyDroid+ successfully removed 59.91% of the total paths on a test dataset consisting of 660 apps without any loss of semantics. We conclude that EspyDroid+ is effective, fast, and scalable in uncovering reflection API induced privacy leaks.

1. Introduction

Since the release of Android, research community and anti-virus industry are continually discovering and reporting various security issues of Android apps. Manual analysis of malware is not feasible looking at the growth rate of Android apps. Each day, malware authors are incorporating various advanced techniques in malicious apps to hinder their analysis by automated tools. Reflection and run-time binding are few primary techniques that are popularly used by malware authors. Andrubis (Lindorfer et al., 2014) is an online research approach that analyses uploaded apps and from the analysis of apps collected over a span of four years of span, they reported that reflection is employed by 57.08% of Android malware samples. Ripple (Zhang et al., 2018), a static analysis based Reflection analysis approach for Android apps, analyzed 6141 Android malware apps collected from the VirusShare project (VirusShare, 0000) and reported that 48.13% of these apps use reflection. DroidRA (Li et al., 2016), a static analysis based approach for analysis of Reflection in Android apps, showed that 76.4% of analyzed 438 apps use reflection in primary app code. These statistics motivated us towards developing a more precise and scalable solution for analyzing reflection APIs.

Static analysis based approaches have several inherent limitations. First, static analysis misses many reflection targets as malware authors typically combine reflection with run-time dependent code. This includes run-time dependency of parameters of reflection APIs using array indices, substrings, encryption, polymorphism, etc. (Rasthofer et al., 2016). Second, static analysis based on type inference like Ripple (Zhang et al., 2018) generates many false positives, leading to low precision.

Dynamic analysis based solutions improve analysis in such situations by executing the app to observe its behavior. The downside, however, is that there exist a large number of paths to be explored during dynamic analysis (Liang et al., 2013; Mahmood et al., 2012; Nath et al., 2013; Rastogi et al., 2013). Therefore, dynamic analysis suffers from the path explosion challenge and thus, cannot achieve the desired coverage.

To address the limitations of static and dynamic analysis, a few hybrid approaches (Bodden et al., 2011; Liu et al., 2017) have been described in literature. Tamiflex (Bodden et al., 2011) is

* Corresponding author.
E-mail addresses: 2014rcp9542@mnit.ac.in (J. Gajrani), vlaxmi@mnit.ac.in (V. Laxmi), bru@mechyd.ac.in (B. Bezawada), manoj.gaur@iitjammu.ac.in (M.S. Gaur), mtripathi.cse@mnit.ac.in (M. Tripathi), zemmari@labri.fr (A. Zemmari).

[1] An early version of this paper was published in Gajrani et al. (2017a)

https://doi.org/10.1016/j.cose.2019.101688
0167-4048/© 2019 Elsevier Ltd. All rights reserved.

an approach for reflection analysis of Java programs. However, Tamiflex misses many true targets, which were behind complex GUI operations. MIRROR (Liu et al., 2017) is an optimized solution compared to Tamiflex for analyzing Java programs. MIRROR reduces the paths to be executed by identifying relevant paths. However, the traditional Java language specific solutions are not directly applicable for Android due to asynchronous and event-driven paradigm with multiple-entry points for app and callbacks. Specifically, in Android complex GUI operations like sending email, SMS, providing appropriate text-inputs are too complex to simulate during dynamic analysis. Not only the simulation of events but also the correct ordering of these operations is a crucial task. As a result of these factors, pure dynamic analysis does not scale well for large apps having many GUI elements.

Android provides Monkey (UI/Application Exerciser Monkey, 0000) as part of Android SDK, which generates the specified number of random events during testing. The limitations of Monkey are: 1) the events are random and hence, may not be relevant for the testing objective, 2) many repeated events are generated, 3) a low probability of generating the correct text-input, and 4) no guaranteed coverage. To provide a better solution, compared to Monkey (UI/Application Exerciser Monkey, 0000), more advanced automated GUI exploration approaches for Android (Borges Jr, 2017; Hu et al., 2017; Mao et al., 2016; Shah et al., 2014) have been proposed. Most of these approaches still have about 30% to 60% code coverage. This implies that, at most, only 60% of the requisite reflection APIs can be analyzed by dynamic analysis if these APIs are distributed uniformly. Therefore, dynamic analysis based on the exploration of a complete app does not guarantee that the desired APIs will be executed.

Further, almost all dynamic analysis based tools fail in the cases where the application depends on user inputs. Approaches such as (Mahmood et al., 2012; Rasthofer et al., 2017; Rastogi et al., 2013) work on providing relevant text inputs; however, it is challenging to simulate correct text-inputs like user-name, password, email-id, phone number, etc., even at the initialization for many apps. Trimdroid (Mirzaei et al., 2016) is an approach for GUI testing of Android applications using an enhanced combinatorial method of providing input-data along with automated program analysis, and formal specifications. However, Trimdroid requires source code to analyze the dependencies between GUI elements and further analysis, which restricts its applicability to analysis of malware and store apps.

This paper proposes, EspyDroid+, an approach that provides a right balance of recall and precision. Precision is the fraction of correctly analyzed targets within all identified targets of reflection APIs.

Low precision implies many incorrect targets identified. Recall is the fraction of correctly analyzed targets within all targets including missed. Low recall implies that many targets are missed.

EspyDroid+ unites dynamic analysis with a static analysis technique of our own design: *Reflection Guided Static Slicing (RGSS)*, to reduce the number of program paths to be explored. *RGSS* generates an optimized app by removing all components that do not lead to the invocation of any reflection call. Further, we maximize the likelihood of execution of the reflection APIs by rewriting various conditional statements through bytecode instrumentation. The aim is to force the reflection APIs to execute at run-time. The main contributions of this paper are as follows:

- We propose EspyDroid+, a hybrid analysis approach for reflection analysis in Android apps to resolve obfuscated reflection calls precisely in comparison to the static state-of-the-art approaches.
- EspyDroid+ uses *RGSS*, an enhanced static analysis approach customized to prune the app. Specifically, prior to the dynamic analysis, *RGSS* generates a sliced app by eliminating irrelevant paths having no reflection APIs. This reduces the impact of the state-space explosion problem faced by the ensuing dynamic analysis phase of EspyDroid+.
- We assess EspyDroid+ on 660 apps collected from diverse sources including benchmark dataset F-Droid (Free and Open Source App Repository, 0000), VirusTotal dataset, malicious apps from malgenome project (Jiang and Zhou, 2012), dataset developed Canadian Institute for Cybersecurity (CIC) Canadian Institute for Cybersecurity, and state-of-the-art dataset (Zhang et al., 2018). *RGSS* prunes 59.91% of total paths on the test dataset of 660 apps. EspyDroid+ outperforms other approaches in terms of both recall and precision.

Organization. The paper is organized as follows: Section 2 provides a motivating example and the challenges in state-of-the-art research. We discuss our proposed solution, EspyDroid+ in Section 3. Section 4 presents implementation details and evaluation results. Section 5 describes few limitations of our proposal. We describe related work in Section 6 and conclude in Section 7 outlining possible future work.

2. Background

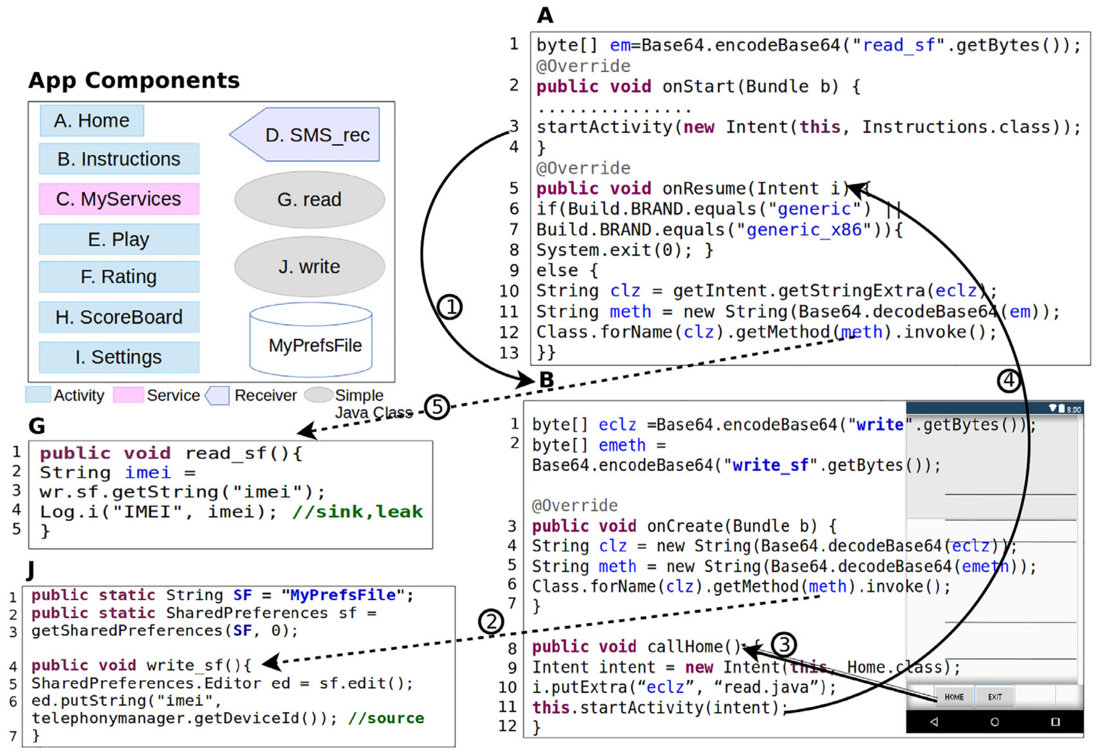
Android provides four types of components as basic blocks of the app: Activity, service, content provider and broadcast receiver. Activity facilitates Graphical User Interface (GUI) and runs in foreground; all the background tasks are implemented as part of service; content provider manages data storage and access through files, SQLite databases, Internet, etc.; broadcast receiver registers itself for one or more events, and is notified by Android OS on the occurrence of these events. Java classes in the app, which do not extend any of these four components, are termed non-component classes. The interaction among components of same or different apps is through inter-component communication (ICC) feature called Intent. The communication APIs use Intent objects, which specify the target component explicitly or implicitly with the help of intent-filters. All the components are declared in the app's manifest file *AndroidManifest.xml*. There is no single entry point of a component unlike the *main()* function in Java. Instead, Android supports defining callback methods (e.g., *onClick()*, *onBackPressed()*, etc.) in components and invocation of these is by the Android system based on the event. Android also provides another way of storing/retrieving data in the form of a key-value pair called `SharedPreferences`.

2.1. A representative example

This section presents an example app that illustrates how run-time binding of reflection APIs limits state-of-the-art static analysis approaches. Through this representative example app, we illustrate the challenges in dynamic analysis. The code base of the example app is a representative of obfuscation techniques used by various malware families and some real-world apps.

The developed app consists of mainly eight components including six activity components, namely: *Home (A)*, *Instructions (B)*, *Play (E)*, *Rating (F)*, *ScoreBoard (H)*, *Settings (I)*, one service component named *MyServices (C)*, one receiver component named *SMS_rec (D)*, and two simple Java classes, named *read (G)* and *write (J)* as shown in Fig. 1. The figure includes the code of only the relevant components to save space and the complete app code is available at https://github.com/jyotigajrani/EspyDroidPlus. The paper uses alphabets in brackets for referring particular class and *X.n* refers to the n^{th} line of class *X* in Fig. 1.

The example reads the IMEI number of the host device and leaks the same to the system log using reflection APIs and obfuscation of parameters. Primarily, activities *A* and *B* contain reflection



A
```
1  byte[] em=Base64.encodeBase64("read_sf".getBytes());
   @Override
2  public void onStart(Bundle b) {
   ...............
3  startActivity(new Intent(this, Instructions.class));
4  }
   @Override
5  public void onResume(Intent i){
6  if(Build.BRAND.equals("generic") ||
7  Build.BRAND.equals("generic_x86")){
8  System.exit(0); }
9  else {
10 String clz = getIntent.getStringExtra(eclz);
11 String meth = new String(Base64.decodeBase64(em));
12 Class.forName(clz).getMethod(meth).invoke();
13 }}
```

B
```
1  byte[] eclz =Base64.encodeBase64("write".getBytes());
2  byte[] emeth =
   Base64.encodeBase64("write_sf".getBytes());

   @Override
3  public void onCreate(Bundle b) {
4  String clz = new String(Base64.decodeBase64(eclz));
5  String meth = new String(Base64.decodeBase64(emeth));
6  Class.forName(clz).getMethod(meth).invoke();
7  }

8  public void callHome(){
9  Intent intent = new Intent(this, Home.class);
10 i.putExtra("eclz", "read.java");
11 this.startActivity(intent);
12 }
```

G
```
1 public void read_sf(){
2 String imei =
3 wr.sf.getString("imei");
4 Log.i("IMEI", imei); //sink,leak
5 }
```

J
```
1 public static String SF = "MyPrefsFile";
2 public static SharedPreferences sf =
3 getSharedPreferences(SF, 0);

4 public void write_sf(){
5 SharedPreferences.Editor ed = sf.edit();
6 ed.putString("imei",
  telephonymanager.getDeviceId()); //source
7 }
```

Fig. 1. A representative example.

API calls. The execution of *A.3* leads to invocation of *onCreate()* lifecycle method of component *B*, which calls the method stored in variable *meth* of class *clz* (B.4-B.6). However, the values are bound to this variable at run-time due to encoded values. This obscures both string (Li et al., 2016) and type inference (Zhang et al., 2018) based static reflection analysis approaches in computing targets. The method actually calls the *write_sf()* method of class *J*. The *write_sf()* method reads the IMEI of the device by invoking *getDeviceID()* method and stores it in `SharedPreferences` "MyPrefsFile" (J.6-J.7). On return from *B* to *A*, *onResume()* life-cycle method of *A* is called. Execution of A.12 invokes the method through reflection APIs where parameter *meth* is encoded through `Base64` encoding. The method name is resolved to *reaf_sf()* of the class read at run-time. The method reads the IMEI from "MyPrefsFile" and sends it to Log (sink) (G.4).

The example illustrates the following challenges for analysis by state-of-the-art approaches (Bai et al., 2018; Chen and Xu, 2001; Gajrani et al., 2017a; Jayaraman et al., 2005; Li et al., 2016; Liu et al., 2017; Rasthofer et al., 2016; Zhang et al., 2018; Zhauniarovich et al., 2015):

- **Late binding of reflection targets:** Android malware families like *Obad*, *FakeNotify* use reflection to call sensitive methods and use encrypted method names, which are decrypted at run-time. The encrypted parameters get their actual values only at run-time. Malware families such as *Fakeinstaller.AH* (Ruiz, Oct 2012) leak private data through the massive use of reflection, which obscure calls to sensitive source and sink APIs. Similarly, the targets of reflection APIs (A.12) have late binding and this thwarts static analysis solutions because of their inability to uncover parameter values in such cases.
- **Inter-component communication:** In Android, due to the widespread usage of ICC, resolving of a value at any point has a dependency on the execution of code present in another component. This dependency is in terms of the values of global variables declared and initialized in other components, values written in shared preferences, values passed from one component to another in Intent extras, etc. Ripple (Zhang et al., 2018) illustrates an example of *AngryBird* app where reflection targets depend on data-flow through Intents. The example of Fig. 1 shows that class name *read.java* is passed as extra information through the Intent from *B* to *A* activity (B.10).

Program slicing is a technique of reducing program in such a way that only those statements and variables that affect the execution of target instruction are computed (called slices) without affecting the program behavior (Weiser, 1981). However, traditional algorithms for computing instruction level slices using def-use chains (Hoffmann et al., 2013; Liu et al., 2017) fail for event based environments like Android when execution of any instruction is dependent on many factors like Intents, GUI events, `SharedPreferences`, reflection, run-time dependency, etc.

Rasthofer et al. (2016) proposed Harvester, an approach for automatically extracting run-time values of parameters of various APIs of interest. Harvester computes instruction-level backward slices starting from the APIs of interest and afterwards executes each of the slice directly independent of its position in the original app which eliminates the need of any GUI interaction. The values of interest (VOIs) such as some class and method names that are dynamically loaded and invoked via reflection are logged at run-time. Harvester is resilient against code obfuscation where the parameters are obfuscated using various ways to hinder static analysis. However, the major limitation of Harvester is that it cannot calculate VOIs in cases of inter component dependency due to the limitation of the highly optimized backward slicing algorithm which is limited within the scope of a single class. One such case is shown in the example of Fig. 1 where the values cannot be inferred by the independent execution of components. The slice computed using instruction-level slicing is for component *A* is shown in Fig. 3. As seen from the slice, the value of parameter at Line A.10 is coming from Intent, which is not captured in the traditional slice.

- **Conditional constraint:** Malware families such as Pincer, Basebridge, NickSpy (Jiang and Zhou, 2012) hinder dynamic analysis through various anti-emulation techniques (Gajrani et al., 2015), time-bombs, logic-bombs, and other conditions. These are presented as conditional checks. The example of Fig. 1 adds anti-emulation constraint (*A.6-A.7*) which check value of BRAND. The value is always "generic" or "generic_86" for emulator while for real device, the value depends on brand of device.
- **Complex GUI:** Malware apps hide malicious code behind complex GUI widgets to hinder analysis by automated tools based on Monkey UI/Application Exerciser Monkey as these are not designed to tackle such complex GUI events. Moreover, providing correct text-input for automated solutions is very challenging.

The above discussion highlights the need for developing approaches that overcome these complex challenges for the ensuring the privacy of Android Apps.

2.2. Problem statement and scope

We are assuming that the app is malicious and developed by attacker. A "Source" is an API that accesses user's private data, and a "Sink" is an API that potentially leaks this data outside the application. The attacker obfuscates the path from Source to Sink APIs through use of reflection APIs. Further, the attacker is aware of evasive techniques, specifically, reflection with inter-component communication and obfuscation of parameters. Most of the reflection aware static analysis techniques are not able to reveal the leaks.

Dynamic analysis is useful in such scenarios, dynamic analysis must generate the necessary GUI and system events leading to the execution of required malicious behavior during testing to achieve efficiency. Also, the dynamic analysis approach, if applied on apps having a large number of program paths results in non-trivial time of analysis and limited code coverage, which makes it unsuitable for large stores. This poses the requirement of an approach that can reduce the state space of dynamic analysis. The state-of-the-art solution (Rasthofer et al., 2016) reduces the space sufficiently but results in loss of information such as data passed between components through inter-component communication.

We aim to design a framework for identifying such hidden leaks with a good balance between state space reduction and accuracy. The key idea is to strengthen dynamic analysis by reducing the state space of Android apps by considering reflection APIs as the target. The target is to support state-of-the-art static analyzers in analyzing otherwise missed leaks without modifying them to improve the recall.

$$\begin{tikzcd} \textbf{A.1} \arrow[r] & \textbf{A.10} \arrow[r] & \textbf{A.11} \arrow[r] & \textbf{A.12} \end{tikzcd}$$

Fig. 3. Traditional slice for Fig. 1.

The problem scope is to unravel all reflection induced leaks by finding obfuscated or run-time dependent targets of reflection APIs. We assume that the reflection calls and ICC calls/RPC calls are not obfuscated.

3. Proposed solution: EspyDroid+

The main contribution of EspyDroid+ is a better slicing algorithm, which enables more reflective calls to be covered by pruning components not leading to reflective calls, and without any loss of accuracy. This makes EspyDroid+ well suited for large applications. The next important contribution is that, EspyDroid+ added the solution for handling anti-analysis constraints. Moreover, EspyDroid+ can trigger more reflective calls because of the handling of non-UI events/actions. EspyDroid+ can handle reflective calls present in both UI and non-UI components unlike state-of-the-art, which mainly focus on UI components. In summary, our approach helps to achieve scalability, which was otherwise a constraint for dynamic analysis. We discuss these aspects in further detail in this section.

Fig. 2 shows the overview of our framework that mainly consists of three phases. The main contribution of EspyDroid+ is the *RGSS* phase, which reduces the state space of dynamic analysis by disconnecting irrelevant paths that never lead to the invocation of reflection APIs. *RGSS* never prunes any direct or indirect relevant components. We keep all the paths towards a relevant component, so if at least one path leading to target code is executed then it will give one set of values of obfuscated parameters of reflection APIs. This over-approximation based approach implies that EspyDroid+ never results in loss of information with respect to analysis targets. *RGSS* customizes the app to drive the app execution towards target APIs. Also, it also performs customization in relevant paths by constraint rewriting to ensure the execution of targets. The second phase is intelligent UI exploration, which executes the app providing various events and inputs. This phase ensures that reflection APIs and their run-time parameters get logged during execution. The last phase is reformation of non-reflection APIs corresponding to reflection APIs using parameter values obtained in the previous phase. Following subsections present each of the modules in detail.

3.1. Reflection guided static slicing

We describe and implement *Reflection Guided Static Slicing (RGSS)* to prune those components of the app, which are not directly or indirectly relevant for the execution of reflection code during dynamic analysis. Specifically, we aim to prune GUI compo-

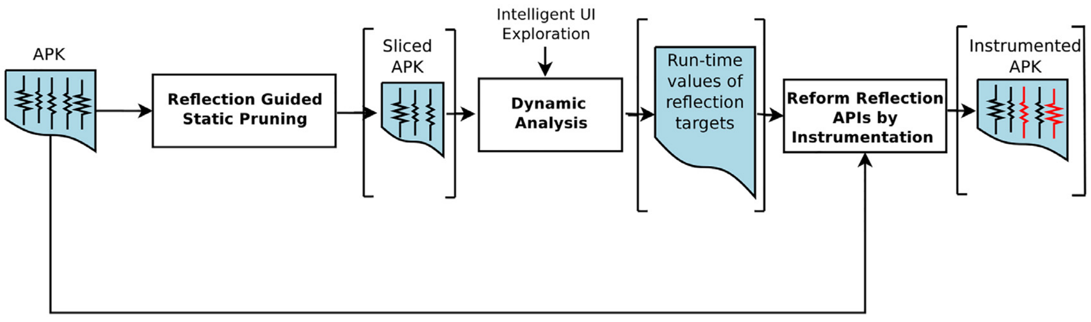

APK
Reflection Guided Static Pruning
Sliced APK
Intelligent UI Exploration
Dynamic Analysis
Run-time values of reflection targets
Reform Reflection APIs by Instrumentation
Instrumented APK

Fig. 2. Design of EspyDroid+.

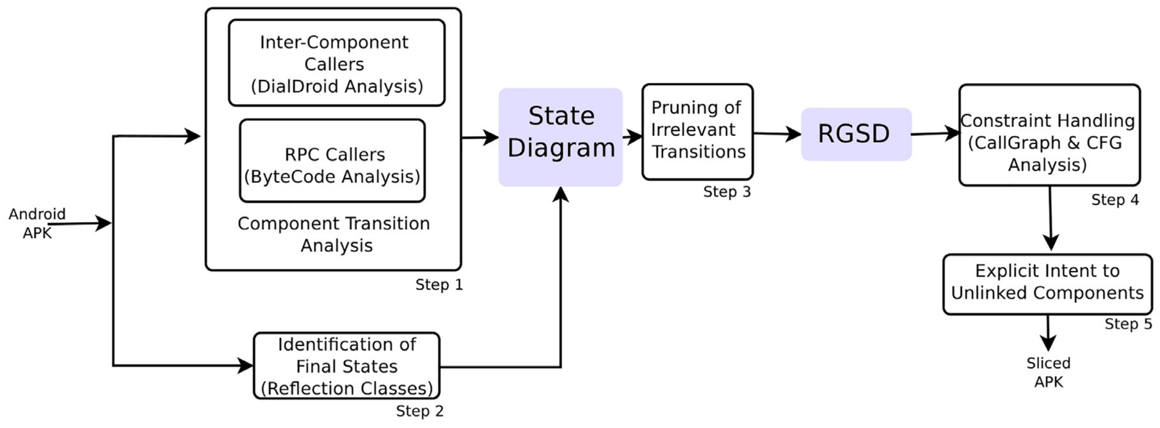

Android APK
Inter-Component Callers (DialDroid Analysis)
RPC Callers (ByteCode Analysis)
Component Transition Analysis
Step 1
Identification of Final States (Reflection Classes)
Step 2
State Diagram
Pruning of Irrelevant Transitions
Step 3
RGSD
Constraint Handling (CallGraph & CFG Analysis)
Step 4
Explicit Intent to Unlinked Components
Step 5
Sliced APK

Fig. 4. Reflection guided static slicing.

nents, i.e., *Activities* to reduce the number of paths to be explored in dynamic analysis. This phase generates a sliced app, which indirectly improves the reachability to target APIs.

Android provides a large number of ways to share and store data such as `Intents`, `Bundle`, `SharedPreferences`, `ContentProviders`, and `JSONObject`. Traditional program slicing will not work on Android due to various reasons such as no-entry point, no explicit invocation of methods, and call-backs. We describe a coarse-grained approach of component level slicing, which does not remove any code from any component except breaking links towards components not containing any relevant API. Our aim is not to compute the smallest slices based on instruction-level slicing but in contrast a safe slice to avoid any loss in the computation of target parameters. *RGSS* includes five steps to construct sliced `apk` as shown in Fig. 4.

We represent an app as 5 tuple state diagram (Q, Σ, δ, q_0, F), where each state $q \in Q$ represents the app's class and each input event $e \in (\Sigma)$ is an instruction whose execution leads to another state. On execution of e, there will be a transition (δ) from one state to another state. The start state q_0 is launcher activity of app that is always unique for any app in Android and defined in the manifest file. EspyDroid+ is implemented on top of Soot (Lam et al., 2011), a bytecode analysis and optimization framework. We extract q_0 using `ProcessManifest` class provided by Soot. All the states having reflective calls are final states (F).

We generate a reduced state diagram by eliminating irrelevant transitions and call it Reflection Guided State Diagram (*RGSD*). We first describe an approach to model standard state diagram from an Android app and then an algorithm designed for generating *RGSD* from the original state diagram.

Step 1: Components transition analysis. This step constructs state diagram from app bytecode. We only include the app's classes as states. In Android, the communication between two components is through ICC methods, which are standard and predefined. A large body of research including IccTA (Li et al., 2015), Epicc, Amandroid (Wei et al., 2014), A3E (Azim and Neamtiu, 2013), DialDroid (Bosu et al., 2017) worked on finding ICC links and targets of ICC calls. We found DialDroid the most suitable as it provides the instruction along with 'caller state' and 'called state' relation. We leverage DialDroid to extract information of ICC links. DialDroid provides information of communication among all four types of Android components using explicit intents, implicit intents, content providers, call to broadcast receivers. The extracted information of ICC links is used to construct state diagram. Reflection APIs can be part of component classes or simple Java classes.

The communication between two non-components is possible through remote procedure calls (RPC). As none of the above tools

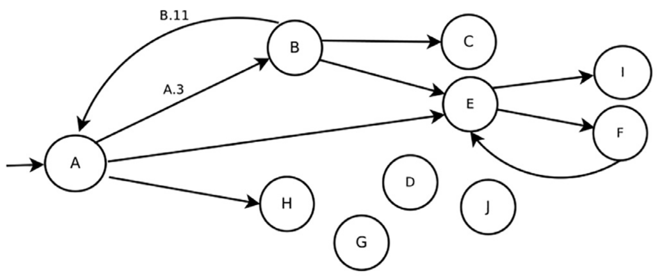


Fig. 5. Result of Step 1 for example of Fig. 1.

works on processing RPC calls, we identified RPC links using callgraph and bytecode analysis. We added code for capturing unsupported edges of *Thread* and *AsyncTask* in Soot generated callgraph. Similar to ICC, the extracted RPCs are added in the state diagram. Our implementation stores the state diagram as Maps (key-value pairs) having caller component name + method name + Σ (i.e., instruction) as key and callee as value. According to Soot's terminology, each instruction of Jimple is called `unit`. The state diagram for example of Fig. 1 is shown in Fig. 5.

Step 2: Extraction of final states. While processing application classes for the presence of RPC calls from their intermediate code, we also parse reflection APIs and store the result in the form of "*CN-MN*" where *MN* is a method of class *CN* containing reflection APIs. All of Android's framework classes are excluded while parsing reflection APIs. These classes are the final states (F) of state diagram and given as input to the next step. The reflection APIs are maintained in the configuration file which is the input of this step along with `apk`. For the example of Fig. 1, this step generates $F = \{A, B, C, D\}$.

Step 3: *State Diagram* to *RGSD* conversion. This step is converting *State Diagram* to *RGSD*, which means removal of all transitions not relevant to reach the final states. We do not remove any code or class from the app except the irrelevant transitions. Instruction level slicing is not always safe because sometimes the slices do not execute correctly or generate exceptions due to various features of Android as discussed in Section 2.1. The key idea is to apply depth-first backward traversal from F for finding useful transitions. The complexity of the Algorithm is $O(Q + \delta)$, which is the complexity of the backward depth-first traversal.

Algorithm 1 describes this step. The results of previous steps, i.e., state diagram and F are input to this step. The algorithm uses a queue, called *processQueue* for maintaining the states to be processed. Initially, we add F to *processQueue* as the backward analysis starts with final states (Line 1). *RQ* is the list for storing all relevant states such that $RQ \subseteq Q$. As all the final states are also relevant states, so we add these to *RQ* (Line 2). $R\delta$, initialized empty, is the set of all relevant transitions such that $R\delta \subseteq \delta$ (Line 3). The traversal starts with processing initialized *processQueue* and continues until the queue becomes empty. The empty queue signifies that the processing has finished. The algorithm dequeues one element at a time from the *processQueue* at a time, stores in *CQ* and finds its predecessors (Line 6). If the node has predecessors, then all the predecessors that are not already present in the queue, are enqueued to *processQueue*. This check is done to ensure that loops are not created during the traversal. These predecessors are added to *RQ* list. All the transitions among each of the predecessors and *CQ* are relevant transitions, and so these are added to $R\delta$ (Lines 9–15).

Algorithm 1: *State diagram* to *RGSD*.

```
Input: SD - State Diagram (Output of Step 1), F - Final States
       (Output of Step 2)
Output: RGSD
/* Analysis of Relevant Transitions                              */
1  processQueue.enque(F) ;   // processQueue for maintaining
   states to be processed
2  RQ ← F ;                  // RQ for storing relevant states
3  Rδ ← ∅ ;                  // Rδ for storing relevant transitions
4  UnlinkedStates ← ∅;
5  while processQueue.empty() is not TRUE do
6      CQ ← processQueue.deque() ; // CQ for storing current
       state
7      PRE ← Predecessors of CQ;
8      if PRE.size > 0 then
9          for pre ∈ PRE do
10             if processQueue.contains(pre) is not TRUE then
11                 processQueue.enque(pre);
12                 RQ.add(pre) ;
13                 Rδ.add(δ(pre, CQ, unit));
14             end
15         end
16     end
17     else
18         if CQ! = q_0 then
19             UnlinkedStates.add(CQ);
20         end
21     end
22 end
/* Pruning of Irrelevant Transitions                             */
23 for each δ in SD do
24     if ! δ.in(Rδ) then
25         Remove δ by swapping corresponding unit with NOP;
26     end
27 end
```

App components like broadcast receivers and services may be invoked based on system events like SMS_RECEIVED, POWER_LESS if the apps register for intent filters for events in manifest files. However, these components may be final states or predecessors of some final states. These components may not have any incoming transition in state diagram as they do not have any call through implicit or explicit Intent. During the path computation, we come across such states, and these are not initial states (q_0) either. This implies that there is no path to these states from launcher activity.

We aim not to miss any such paths during dynamic analysis. Therefore, we prepared a separate list, *UnlinkedStates* containing all such states that do not have any paths from the start state, but which are either final states or leading to a path towards final states. The list is used as input in step 5 where we add direct

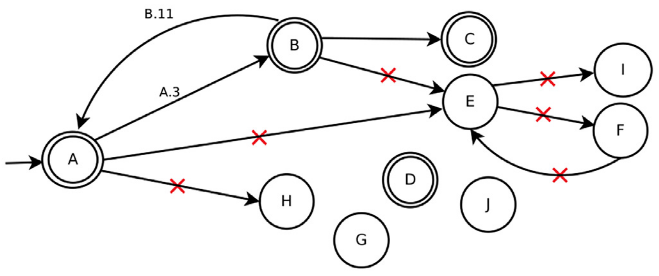


Fig. 6. Result of Step 3 for example of Fig. 1.

Intent for these states. Once the processing of queue finishes, we have a list of all relevant transitions $R\delta$.

We remove all the transitions that are not part of $R\delta$ and have activity as target (Lines 23–27). From an implementation point of view, this implies swapping the corresponding `unit` in Jimple with NOP. Fig. 6 shows the *RGSD* after this step.

Step 4: Constraint handling. It is observed that intelligent malware families put various anti-analysis constraints like emulator checks (Vidas and Christin, 2014) e.g., Pincer, time-bombs (CIDRE, 2016) e.g., DroidKungFu, complex logic bombs (Fratantonio et al., 2016) e.g., SMSSpy, location bombs (Fratantonio et al., 2016) e.g., Adware_1, commands from servers (Wang et al., 2017) e.g., Anserverbot, battery above certain limit, availability of some file, presence of call logs, etc., for hindering dynamic analysis. Therefore, as the next step of *RGSS*, we handled taming of anti-analysis conditions. The example of Fig. 1 represent one such constraint in *A* Activity (A.6). The constraint checks Build Brand for values which signify that the execution environment is an emulator. The malicious activity is triggered only when the app is executed on the device. This will restrict the code execution during dynamic analysis whenever the condition is not satisfied. To handle such constraints, we instrument Jimple to satisfy conditions preventing the execution of malicious code.

Algorithm 2 explains our approach for handling constraints. For each of the useful transitions, i.e., `unit` identified by Step 3, we determine its root method and the first statement of root method

Algorithm 2: Rewrite constraints.

```
Input: Rδ, All Useful Transitions
Output: Constraints Rewritten in Jimple
1  cfg ← Generate ControlFlowGraph() of App;
2  for each unit ∈ Rδ do
3      RM ← CalculateRootMethod(unit);
4      From ← First statement of RM;
5      CFP ← getControlFlowPathBetween(From, unit, cfg);
6      for each STMT ∈ CFP do
7          if STMT instanceof IfStmt then
8              if unit is in TRUE branch of STMT then
9                  NEWSTMT ← IfStmt with condition 1 == 1;
10             end
11             else
12                 NEWSTMT ← IfStmt with condition 1 == 0;
13             end
14             Swap STMT with NEWSTMT;
15         end
16     end
17 end
```

using inter-procedural analysis within the class using callgraph. The callgraph has caller-callee relation within the class. By root method, we mean the topmost caller of the method that contains

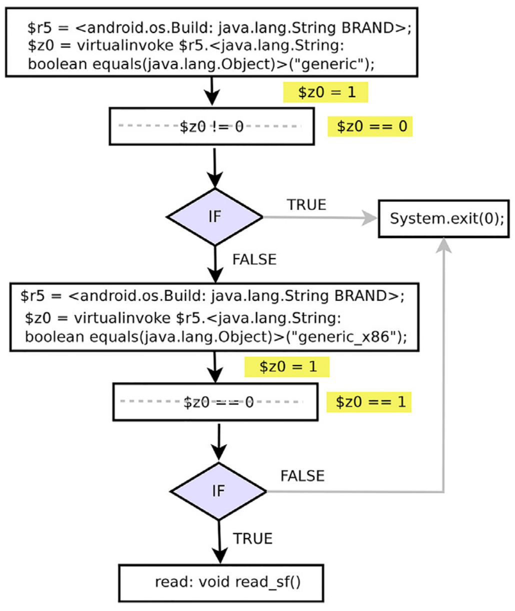

$r5 = <android.os.Build: java.lang.String BRAND>;
$z0 = virtualinvoke $r5.<java.lang.String:
boolean equals(java.lang.Object)>("generic");
$z0 = 1
$z0 != 0
$z0 == 0
IF
TRUE
System.exit(0);
FALSE
$r5 = <android.os.Build: java.lang.String BRAND>;
$z0 = virtualinvoke $r5.<java.lang.String:
boolean equals(java.lang.Object)>("generic_x86");
$z0 = 1
$z0 == 0
$z0 == 1
IF
FALSE
TRUE
read: void read_sf()

Fig. 7. Result of Step 4 for example of Fig. 1.

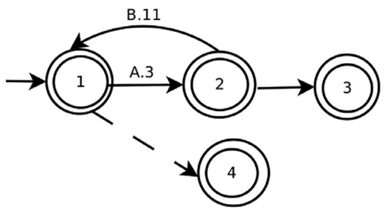


Fig. 8. Result of Step 5 for example of Fig. 1.

the transition. The reason for considering root method is because in Android we do not have a compulsory main method (line 3–4).

Next, we compute the control-flow path from root method's first statement to the current useful `unit` using the control flow graph (line 5). We determine whether the targets are in *THEN* (branch target) or *ELSE* (fall through) branch. Accordingly, we construct *IF* statement with boolean TRUE or FALSE as condition as shown in lines 8–13 of Algorithm 2. The new *IF* condition is swapped with original IF condition in intermediate representation Jimple to enforce the execution of targets (line 14). The main challenge here was callback/lifecycle methods, which do not have any direct call. We take advantage of FlowDroid's (Arzt et al., 2014) advance method of constructing callgraph, which adds *DummyMainMethod* having edges to life-cycle methods. The Fig. 7 shows the control flow graph for component *A* of Fig. 1. The dark arrows show the path required to be executed to trigger malicious code. The code in boxes (yellow) indicates the instrumented code. Algorithm 2 has the complexity of $O(R\delta^{*}CFP)$.

The challenge was that rewriting *FOR* or *WHILE* loops in this way makes the program logically incorrect. The instrumentation of condition with TRUE makes the loop infinite and with FALSE makes the loop never executable. To handle this, we exclude loops by using *LoopFinder* LoopFinder class of Soot while instrumenting.

The other impact of constraint rewriting is that if 2 or more branches of conditional statement contain reflection code, only one of them will be kept and the others will be automatically removed. This is due to Soot's optimization feature, which removes all other branches if one of the branch is definitely true. As we could only execute one branch, in this case, there is some loss. However, in this case, we could construct more than one app where each app corresponds to each branch of original code.

Step 5: Explicit intent to unlinked components. Step 3 provides the *UnlinkedStates*, i.e., the list of all states that are useful but whose path from launcher state is not available. These are the states primarily corresponding to non-GUI components receivers and services. The reason of missed callers of these components is Android's event-driven execution. Invocation of some components is the result of system events like SMS received, call received etc, or some custom event like user-defined action in intent-filter registered in manifest file by app. MalGenome dataset contains malicious apps, which register for specific events (Zhou and Jiang, 2012). Such components are called by Android when the event for which the component subscribe occurs. Therefore, the suspicious code present in these components may not execute during dynamic analysis. Android 8.1 supports 191 different broadcast actions and 26 different service actions. Simulating these large number of events during dynamic analysis is not practical and acceptable. Only the events relevant to app under analysis can be simulated. AppsPlayground (Rastogi et al., 2013) artificially raises few system events like boot completed event by using a fast reboot of VM. However, the approach causes system inconsistencies and it is not possible to simulate all events artificially.

One solution for this is proposed in Dynodroid (Machiry et al., 2013), which instruments Android SDK for observing the system events registered by app. The instrumented SDK is loaded along with emulator which observes broadcast receivers and system services. Thereafter, the appropriate intent along with random data is serialized and events are triggered by modified Activity *Manager tool (am)*. The modifications are done to add support for all types of arguments, which were missed in original *am* tool. The principal limitation of DynoDroid (Machiry et al., 2013) is that for finding system events registered by app and thus the components registering these, SDK modification is required, which is not generic for all versions of Android. In contrast, EspyDroid+ gets the name of these components and corresponding events through previous steps and then instruments the code for sending intent to each of the component class in *UnlinkedStates* in Jimple. The intent for a specific component is constructed with a specified action and corresponding intent data with valid random values. This data is used in constructing *putExtra* and *putString* calls.

The selection of instrumentation point is done based on the type of unlinked component. The intent for receivers and services are added in launcher state while the intent for activity is placed in final state. This is because non-GUI components work in background through launcher without affecting the app's original functionality. While linking activity may affect the app's original functionality therefore, we add intent for unlinked activities in final state, which has path from launcher. The preference is given to the final state, which is not having any outgoing transitions to preserve the app's original flow. However, if no such final state is found then any final state with the least number of outgoing transitions is selected. For the example of Fig. 1, explicit intent is added as shown in Fig. 8 by the dotted arrow from state 1 to state 4. The overall effectiveness of *RGSS* depends on relevancy of app code with respect to analysis target.

3.2. Intelligent UI exploration

We leverage Intelligent UI exploration module from our earlier work (Gajrani et al., 2017b) along with some improvements. The approach is black-box developed by extending the Robotium framework (Zadgaonkar, 2013). The exploration time depends on the complexity of app GUI. The average exploration time for the experimental data-set of 40 apps as measured in (Gajrani et al., 2017b) was 6 minutes. In comparison, when the same dataset's sliced apps by EspyDroid+ were used for testing, the average exploration time reduced to 3.5 minutes.

In (Gajrani et al., 2017b), we compare our earlier Intelligent UI testing approach with Android Monkey in terms of Class, Method, Block, and Line coverage using Emma tool EMMA on 40 open source apps from F-Droid (Free and Open Source App Repository, 0000). We selected F-Droid dataset for comparing coverage of dynamic analysis as Emma requires source code of apps undergoing testing. The average coverage on 40 apps in percentage for Class, Method, Block, and Line by Monkey is 55, 47, 33, and 35 (42.5) while for our approach, the coverage is 58, 50.2, 42, and 40 (47.55) which shows improvements of our dynamic analysis over Android Monkey.

We could not compare with most relevant and contemporary dynamic analysis framework Harvester (Rasthofer et al., 2016) results on same dataset as Harvester is not available in public domain and the apps are not open source. DROIDPF (Bai et al., 2018), another relevant work presented dynamic analysis results in terms of exploration time and property violation and not especially focused on coverage of reflection APIs.

The main objective of this work was to improve precision of dynamic analysis presented in our earlier work (Gajrani et al., 2017b). We improve the Intelligent UI Exploration in two ways. First, we do not want app to terminate execution if proper text inputs are not given. Therefore,we propose to generate validation complaint text input e.g., email-id, date, phone number,etc., Instead of generating completely random input, we provide domain specific input by maintaining database of valid inputs corresponding to possible labels/hints available for these inputs in app. Second, we employ error detection and handling module. In earlier works, no consideration was given to UI exploration terminating due to any error encountered in app. This resulted in many prematurely terminated unsuccessful runs in dynamic analysis. We augment a solution to detect errors and start the exploration again taking care that erroneous inputs are not repeated. The widgets which cause abnormal app termination due to invalid input or event, insufficient data etc. during execution are recorded and it is ensured that errorneous views are ignored in subsequent runs. This avoids unnecessary and unsuccessful execution of apps.

From implementation point, we create `SharedPreferences` for the app under test and store the information of all widgets causing termination in that. The reason of choosing `SharedPreferences` is because it does not require any permission. The `SharedPreferences` is created with values initially set to *null* during first run. Each time before performing any action on any widget, it is confirmed that the widget is not present in the `SharedPreferences`. Before exploring particular widget, the necessary information to identify that widget is printed to the system logs. If the app terminates with exception then the information from logs is stored to `SharedPreferences` so that the same can be ignored in next execution. The execution of app starts again with the updated `SharedPreferences` file.

3.3. Reform reflection APIs

This phase constructs the code statements that use reflection by their semantically equivalent non-reflection way (traditional Java calling) and instrument these in the original app (Gajrani et al., 2017a). The automated instrumentation is done in Soot's intermediate representation Jimple using dynamically obtained parameters. The non-reflection calls are simple Java calls to these methods constructed with the obtained values of corresponding parameters. The instrumented code before reflection call A.12 of activity *A* is as shown in Listing 1. The instrumentation allows off-the-shelf static analyses tools such as FlowDroid (Arzt et al., 2014), IC3 (Octeau et al., 2015) to correctly interpret originally

```
$r9 = new com.example.read;
$r7=virtualinvoke $r9.<com.example.read: void
     read_sf()>();
```

Listing. 1. Instrumented code.

obfuscated behaviors, which were missed by these tools while analyzing original apps. The original reflective calls are not removed while instrumenting new calls.

4. Implementation and evaluation

RGSS is implemented on top of Soot (Lam et al., 2011). Step 1 finds ICC transitions by leveraging DialDroid (Bosu et al., 2017) which is also based on Soot. We design a parser module to process and store the results of DialDroid as Maps (key-value pairs) having caller component name + method name + Σ (i.e., instruction) as key and callee as value. Both Pruning of Irrevant Transitions (Step 3) and Constraint Handling (Step 4) use Jimple Transformation Pack (jtp) of Soot during instrumentation. We performed all the evaluation on Intel E5-2420v2 2.20 GHz processor with 80 GB memory.

We evaluate EspyDroid+ on diverse datasets to avoid any potential discrimination. The first dataset is composed of 13 medium size open source apps downloaded from F-Droid (Free and Open Source App Repository, 0000). The reason for choosing this dataset is the availability of app source code, which makes it easier for manual verification purpose. We manually add malicious reflection code, which includes ICC dependency and obfuscation in each app. We obtain ICC communication among components of each app and select any two components (caller-callee) at random having communication. The code for collecting IMEI using reflection APIs and sending this IMEI to callee along with ICC call as parameter is added in caller component. Similarly, the code for receiving of IMEI and leaking it through use of reflection APIs is added in callee component. This approach of adding reflection is based on various samples contributed by us to DroidBench.[2] The second dataset composed of same 17 apps as used by Ripple (Zhang et al., 2018) on reflection analysis. All the apps are downloaded from Google Play store. The choice to select this dataset is because the apps contain reflection code and we could compare the results. The average app size is 55.23 MB in this dataset. The apps were quite large as compared to F-Droid dataset and all the apps were having complex GUI, and large number of components, which is a challenge for state-of-the-art. The third dataset is composed of randomly selected samples (2017 and 2018) downloaded from VirusTotal. As the fourth dataset, we choose various families from Malgenome project (Jiang and Zhou, 2012). Specifically, we selected the apps from families that massively use reflection to hide calls to sensitive API methods used for leaking sensitive data. The last dataset is composed of 413 apps which are classified by authors in four categories i.e., SMSMalware, Adware, Scareware, and Ransomware. This dataset is created by the Canadian Institute for Cybersecurity (CIC) Canadian Institute for Cybersecurity and recently used in (Fallah and Bidgoly, 2019; Lashkari et al., 2017; 2018; Lee and Park, 2019; Taheri et al., 2019).

Table 1 shows the results of *RGSS* on all the datasets in terms of transitions present in the original dataset (Column 5) and transitions present after the *RGSS* (Column 6). The total number of transitions as shown by last row (computed for 267 apps having reflection) reduced from 4171 to 2499 which means 59.91% transitions are reduced because of *RGSS*. The decrease directly cor-

[2] https://github.com/secure-software-engineering/DroidBench/tree/develop#reflection_icc-

Table 1
Performance evaluation of proposed approach *RGSS*.

Dataset Name	#App	Avg App Size (in MB)	#Apps having Reflection	#Transitions (Original)	#Transitions (Sliced)	#Explicit Intents	# Final States	Avg Time/App (in Sec.)
F-Droid	13	0.44	13	95	36	2	39	10.43
Ripple	17	55.23	17	512	394	5	266	68.12
Virustotal	200	3.35	71	1152	788	50	617	21.98
MalGenome	17	0.69	11	313	193	10	80	11.42
SMSMalware	107	1.84	52	425	196	8	141	34.5
Adware	97	7.76	52	623	440	10	130	60.45
Scareware	108	5.72	34	727	292	15	142	55.3
Ransomware	101	1.11	17	324	160	12	102	34.2
Total	660	59.71	267	4171	2499	112	1517	296.4

Table 2
Evaluation of dynamic analysis.

App Name	Dataset	Original		Sliced	
		#RLog	Time (Sec)	#RLog	Time (Sec)
0dcc91b6960a86537ae79a59b9fb8b824cd8089548ec1ee5f20f9dc63d051600	VT	15	72	15	41
1c2fda7cf9b7c33f0436ac058f6f02e292148d579f90f106a5959a364d23d8b8	VT	53	38	58	13
0e7a4e557ed2fc7828f8261abfd40478f0972bf24e67bebfc1fbaa8fa3834a5c	VT	14	81	14	20
0aaf731eba4a4a74a515e32970cfea73743d73eb114a21dff62340c3e9eb4d7f	VT	15	90	17	78
0a3e346cce78c22a1e8a6ee92d97e65050bcd92339eda1ee738759446a8b4f4e	VT	0	60	6	13
AirPush Detector	F-Droid	5	71	5	14
External IP	F-Droid	3	57	3	22
Android Time Tracker	F-Droid	3	62	3	39
santander	Ripple	4	90	4	60
Result for 267 apps having reflection		2684	15,356	2950	7486

VT - VirusTotal, #RLog - Number of reflection calls logged.

responds to a huge reduction in the exploration space of dynamic analysis with same accuracy. This reduction is without loss of any information as our approach does not remove any path leading to desired state. As shown in the last column of Table 1, exploration of sliced apps takes around 1 minute for the average app size of approximately 55 MB while the original apps have quite large exploration time. The automated exploration follows the approach mentioned in (Gajrani et al., 2017b). Table 1 - column 7 depicts the number of explicit transitions added by EspyDroid+ for different datasets for improving the coverage of reflective calls. We manually verify the results of Table 1 for F-Droid dataset (opensource) and found that EspyDroid+ is able to log all reflection calls successfully.

We used Android emulator having API level 19 for experiments. The apps are instrumented with APIMonitor to report the reflection APIs executed in Logcat.[3] Table 2 shows the results of number of distinct reflection calls logged and exploration time for both original (#RLog (Original)) and sliced app (#RLog (Sliced)) after dynamic analysis module of EspyDroid+. Due to paucity of space, details of nine randomly selected apps have been added. We have added the results of all the 267 apps as the last row of Table 2.

As seen from the Table 2, the number of logged reflection APIs is sufficiently higher on sliced apps. Moreover, the exploration time on sliced apps is reduced significantly. **The results in Table 2 show that EspyDroid+ has 9.91% more coverage (Number of reflection calls logged increased from 2684 to 2950) on sliced apps because of conditional rewriting through instrumentation, improved UI exploration, and adding of explicit intents.** Table 2 **further shows 51.25% of decrease in analysis time (exploration time reduced from 15,356 seconds to 7486 seconds) due to removal of irrelevant paths.** We came across various sensitive run-time operations, such as termination of incoming calls, connecting to Command & Control server, etc., which were hidden through reflection deeply inside app components.

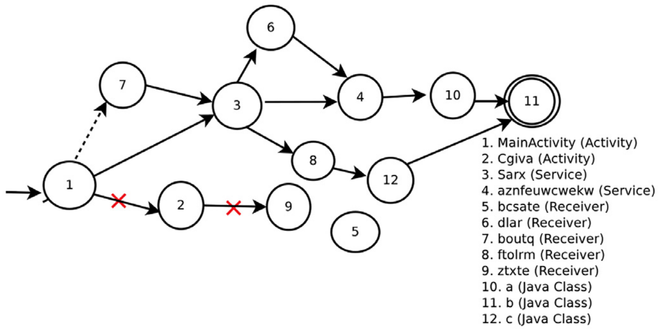


Fig. 9. *RGSS* for representative example of Virustotal dataset.

Fig. 9 shows the output of *RGSS* for an app from Virustotal dataset.[4] The app checks whether Internet is available or not and turn on the mobile data in case Internet not available. The app has declared permission of *CHANGE_NETWORK_STATE* along with other dangerous permissions. The app uses field, method, and class reflection for enabling mobile data and thereafter leaks various information of device like IMEI, Build, release, etc., through Internet. The original app is having 12 states (2 activities, 2 services, 5 receivers, and 3 non-component classes), 12 transitions, and one final states (State 11). Reflective calls were present mainly in one Java class.

RGSS reduced the number of transitions to 10 where one of the valuable pruning is the transition between *MainActivity* to *Cgiva* Activity (State 1 to State 2 in Fig. 9). As shown in Table 2, the number of reflection calls logged in original app is 0. The dynamic analysis is triggering *Cgiva* activity and then stopping the analysis. There is receiver *boutq* which has paths to final state. However, its calling is dependent on system event *BOOT_COMPLETED*, and

[3] https://developer.android.com/studio/command-line/logcat.

[4] 0a3e346cce78c22a1e8a6ee92d97e65050bcd92339eda1ee738759446a8b4f4e

therefore, receiver's call is added from Launcher activity (dotted line between State 1 and 7). Moreover, the invocation of service *aznfeuwcwekw* in state 3 is controlled through IF condition which checks the specific value of incoming Intent Extra. EspyDroid+ has successfully rewritten that constraint to make forced invocation of service and able to log all reflection APIs.

5. Limitations of EspyDroid+

EspyDroid+ cannot handle hybrid apps that include native code in the app itself not from the Android OS, and JavaScript code. If the ICC links are not statically resolvable by Dialdroid, then component transition analysis would not calculate paths completely. Nonetheless, the success rate, identified links/actual links, of DialDroid is 95% as measured and verified manually on open source apps downloaded from F-Droid Free and Open Source App Repository. DialDroid misses the links to component of fragment Activity type so the transition in state diagram is missed. However, EspyDroid+ adds a direct intent to such missed components. EspyDroid+ instruments the apps for optimization as well for appending non-reflection calls corresponding to reflection calls. Therefore, EspyDroid+ can not analyze the apps which have implemented the mechanism to prevent themseleves from being modified or function improperly on being modified. EspyDroid+ will be inefficient in situations where the app contains behavior dependent on specific data inputs which are not removed in *RGSS* and the desired inputs are not provided by dynamic analysis.

The effectiveness of static reduction depends on whether a large part of the app is irrelevant with respect to the reflection code. However, our approach never prunes a relevant component as it does an over-approximation. We keep all paths towards a relevant component, so if at least, one path leading to target code will be executed, it give at least 1 value of obfuscated parameters of reflection APIs. The over-approximation approach implies that EspyDroid+ never results in loss of information due to slicing with respect to analysis targets.

6. Related work

We broadly categorize approaches for reflection analysis of mobile apps in static analysis and dynamic analysis. This section elaborates upon various state-of-the-art solutions under each approach.

6.1. Static analysis

Li et al., proposed DroidRA (Li et al., 2016), which uses string inference for resolving reflection targets in Android apps. DroidRA addresses reflection by solving a constant string propagation problem. However, the approach can resolve class and method names from reflective calls only if these are constants. This limits the approach for targets which are non-constant.

Ripple (Zhang et al., 2018) further supplemented string inference with type inference for getting reflection targets, which are non-constant and non-null strings but whose variable/object type can be inferred like values read from configuration files or provided from command lines. For a known target method whose receiver object is unknown, it infers the object based on the type of method's class. However, type inference is limited due to a large number of false positives as many objects/variable may have the same type in a method. This leads to low precision. Second, static analysis based on type inference like Ripple generates many false positives, leading to low precision. Ripple is unable to resolve many reflection targets. The primary reason is attributed to the lack of both string and type information for the reflective targets that are accessed with statically unknown ways like encryption, polymorphism, and reflection. EspyDroid+ prevents false alarms in contrast to Ripple as targets are reported only after actually executing the app. EspyDroid+ has high recall as it can effectively handle obfuscation with the use of dynamic analysis.

Authors in MIRROR (Liu et al., 2017) propose hybrid reflection analysis approach for Java applications. The reflection oriented slicing ahead of dynamic analysis increases the code coverage of dynamic analysis significantly with low false reflective targets. However, the approach does not apply to Android due to various reasons like no single entry point, event-oriented execution, dependency between components using different ways as mentioned in Section 2.1.

6.2. Dynamic/hybrid analysis

Harvester (Rasthofer et al., 2016) as described in Section 2.1 computes highly optimized slices (intra-component slices) and thus will not be able to identify the target of reflection APIs accurately during ICC dependency. EspyDroid+ approach of slicing is peculiar to component level slicing which improves the overall recall. Standard methods of computing the backward slices (Chen and Xu, 2001; Jayaraman et al., 2005) are not practically feasible for platforms like Android.

DROIDPF (Bai et al., 2018) is a framework that proposes model checking for identifying sensitive leaks in Android apps. The authors developed executable mock-up Android OS for enabling JPF (Visser et al., 2003) to dynamically explore the concrete state spaces of Android apps. Mock-up OS includes programs for generation of user interactions and environmental inputs, which drive the dynamic execution of the apps. However, DROIDPF identifies obfuscated reflection targets only and does not include reformation and instrumentation of equivalent non-reflection APIs in the original app. Also, it is limited by underlying JPF so it can only explore the app if the underlying JPF can explore. The approach is too complex due to the design of mock-up OS. Adding features of new Android releases and associated libraries require the efforts of continuous development of the precise model of mock-up OS for verifying latest apps, which limits the scalability of DROIDPF. DROIDPF is not fully automatic as one of the case is Pincer which is controlled by C&C server. For Pincer, DROIDPF can find malicious behaviors but with manual efforts of analysts by providing data inputs, which satisfy branching conditions. In contrast, EspyDroid+ includes taming conditional behaviors that make it independent of such external events.

StaDyna (Zhauniarovich et al., 2015) focuses on addressing dynamic code loading and reflection using the hybrid approach. The static analysis first constructs call graph of application, then run the app and afterward expands callgraph with additional information captured at run-time. However, evaluation is done only in terms of an increased number of nodes and edges without any focus on privacy leakage. It needs a modified Android OS, which makes its installation and usage difficult. It does not provide a way to directly benefit existing static analyzers, i.e., to support them in performing reflection-aware analyses.

The first version of EspyDroid+ named EspyDroid was presented for uncovering reflection employed with run-time dependency in Android apps in our earlier work (Gajrani et al., 2017a). EspyDroid+ has a number of considerable enhancements. First, EspyDroid+ proposes *RGSS*, a novel slicing approach that strengthens dynamic analysis by pruning irrelevant paths with respect to analysis targets. *RGSS* models the original app in form of Deterministic Finite Automaton by capturing transitions using intra-component, inter-component, and remote call analysis techniques. As detailed in Section 3.1, EspyDroid+ proposes an approach for converting *RGSD (Reflection Guided State Diagram)* from the original state diagram. *RGSD* prunes all the irrelevant transitions without any loss of accuracy. The results show that dynamic analysis of *RGSD* instead of original state diagram significantly reduces analysis time and improves the recall for large sized apps. Second, EspyDroid+ handles the anti-analysis characteristics found in recent malware by proposing constraint rewriting and control-flow analysis using Soot as detailed in Section 3.1-Step 4. Third, along with dynamically analyzing GUI components (activities), we propose to analyze leaks hidden in non-GUI components (receivers and services) by instrumenting explicit calls to these components. Forth, intelligent UI exploration instead of simple Monkey in dynamic analysis helps in handling exceptions. Further, we performed testing on more sophisticated and diverse dataset, and added various experimental results. In summary, EspyDroid+ primarily helps in achieving scalability by solving path-explosion problem of dynamic analysis, improves the recall by handling anti-analysis constraints and non-GUI components.

6.3. Machine learning

In Android, number of vulnerable and malicious apps are increasing at an alarming rate and due to this, analysis time per app should be as small as possible without any loss of accuracy. Therefore, many authors have proposed machine learning based approaches, especially, deep learning based approaches which may be able to identify vulnerabilities not yet found.

Wang Wei et al. (Wang et al., 2018) propose a framework which employs an ensemble of multiple classifiers for detecting malicious apps and also categorizing benign apps. They extract 2,374,340 features from each app and apply Support Vector Machine (SVM) model to assign weights to features based on their effectiveness in distinguishing benign and malicious features. Finally, they used 34,630 top ranked features to perform the detection of malicious apps and the categorization of benign apps. The app undergoes classification by 5 classifiers i.e., SVM, Classification and Regression Tree (CART), Random Forest, Naive Bayes, and KNN algorithms. The final classification is based on the majority of these five classifiers. The authors shown effectiveness of proposed approach by performing evaluation on a large app set of 107,327 benign apps and 8701 malicious apps.

Liu Xing et al. (Liu et al., 2019) designed and implemented framework called “Alde”, which discovers privacy leaks caused by analytics libraries. The authors show that even some Google playstore and Chinese market apps leak private information to analytics libraries without users consent allows the owners of these companies to profile and characterize the users. The authors proposed and developed an app “ALManager” which leverages Xposed framework (Xposed Module Repository, 2017) to hook tracking APIs used by analytics libraries and further control the information to be collected by these.

Android apps have high dimensional feature set which require high training time. Wang et al. (Wang et al., 2019) propose to combine deep autoencoder (DAE) with convolutional neural network (CNN) to improve the detection accuracy and reduce the training time. Authors show that with inclusion of DAE, essential features of Android apps are captured efficiently. Further, CNN-S and CNN-P structures are employed and experimental results demonstrate that CNN-S model improves accuracy by 5% compared with Support Vector Machine (SVM) on the dataset of 23,000 apps. The time required for training is reduced by 83% by using DAE-CNN when compared to CNN-S model.

7. Conclusion and future work

In this work, we propose EspyDroid+, an automatic reflection analysis approach for Android apps. EspyDroid+ tackles the limitation of static analysis in verifying true targets of reflection by dynamically executing the app. Further, we propose *RGSS* for reducing the state space of dynamic analysis. We propose an approach to model standard state diagram from an Android app and modeling state diagram into an efficient Reflection Guided State Diagram (*RGSD*), which is suitable for dynamic analysis of reflection APIs. Instead of exploration of a complete app, dynamic analysis executes the sliced app having paths leading to reflection code. EspyDroid+ is also effective in handling anti-analysis, logic bombs, time bombs, C&C controlled execution, etc. Although, the current focus is on reflection code, it could be easily extended to log parameters of other sensitive sinks similarly. Limiting the number of paths to be explored based on analysis target makes EspyDroid+ an efficient solution for analyzing large real-world Android applications. We applied EspyDroid+ on a collection of 660 apps from diverse sources and showed that EspyDroid+ can significantly reduce the exploration time as the links not relevant to reflection code are broken.

Machine learning based solutions detect any possible malicious activity based on the structure of the app and depend on how good training dataset is. Considering that many zero day vulnerabilities are discovered only after an attack, modeling this vulnerability requires an analysis of effect of attack on vulnerable app and hybrid of static, dynamic, and machine based methods may be more effective than any single analysis technique. Therefore, in future, we further work on hybrid solution combining static, dynamic, and machine learning based methods.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The work is also partially supported by DST-CNRS project IFC/DST-CNRS/2015-01/332 at MNIT Jaipur.

References

APIMonitor, https://github.com/pjlantz/droidbox/tree/master/APIMonitor.
Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., 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.
Azim, T., Neamtiu, I., 2013. Targeted and depth-first exploration for systematic testing of android apps. In: Acm Sigplan Notices, 48. ACM, pp. 641–660.
Bai, G., Ye, Q., Wu, Y., Botha, H., Sun, J., Liu, Y., Dong, J.S., Visser, W., 2018. Towards model checking android applications. IEEE Trans. Softw. Eng. 44 (6), 595–612.
Bodden, E., Sewe, A., Sinschek, J., Oueslati, H., Mezini, M., 2011. Taming reflection: aiding static analysis in the presence of reflection and custom class loaders. In: Proceedings of the 33rd International Conference on Software Engineering. ACM, pp. 241–250.
Borges Jr, N.P., 2017. Data flow oriented ui testing: exploiting data flows and ui elements to test android applications. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, pp. 432–435.
Bosu, A., Liu, F., Yao, D.D., Wang, G., 2017. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security. ACM, pp. 71–85.
Canadian Institute for Cybersecurity (CIC), University of New Brunswick, Fredericton, NB, Canada. https://www.unb.ca/cic/.
Chen, Z., Xu, B., 2001. Slicing object-oriented java programs. ACM SIGPLAN Notices 36 (4), 33–40.
CIDRE, E., 2016. Kharon dataset: android malware under a microscope. Learn. from Authoritative Secur. Exp. Results 1.
EMMA. a free Java code coverage tool, http://emma.sourceforge.net/.
Fallah, S., Bidgoly, A.J., 2019. Benchmarking machine learning algorithms for android malware detection. Jordanian J. Comput. Inf. Technol. (JJCIT) 5 (03).
Fratantonio, Y., Bianchi, A., Robertson, W., Kirda, E., Kruegel, C., Vigna, G., 2016. Triggerscope: towards detecting logic bombs in android applications. In: Security and Privacy (SP), 2016 IEEE Symposium on. IEEE, pp. 377–396.
Free and Open Source App Repository, https://f-droid.org/.
Gajrani, J., Laxmi, V., Tripathi, M., Gaur, M.S., Sharma, D.R., Zemmari, A., Mosbah, M., Conti, M., 2017. Unraveling reflection induced sensitive leaks in android apps.

In: International Conference on Risks and Security of Internet and Systems. Springer, pp. 49–65.
Gajrani, J., Sarswat, J., Tripathi, M., Laxmi, V., Gaur, M.S., Conti, M., 2015. A robust dynamic analysis system preventing sandbox detection by android malware. In: Proceedings of the 8th International Conference on Security of Information and Networks. ACM, pp. 290–295.
Gajrani, J., Tripathi, M., Laxmi, V., Gaur, M.S., Conti, M., Rajarajan, M., 2017. spectra: a precise framework for analyzing cryptographic vulnerabilities in android apps. In: 2017 14th IEEE Annual Consumer Communications & Networking Conference (CCNC). IEEE, pp. 854–860.
Hoffmann, J., Ussath, M., Holz, T., Spreitzenbarth, M., 2013. Slicing droids: program slicing for smali code. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing. ACM, pp. 1844–1851.
Hu, Z., Ma, Y., Huang, Y., 2017. Droidwalker: Generating reproducible test cases via automatic exploration of android apps. arXiv: 1710.08562.
Jayaraman, G., Ranganath, V.P., Hatcliff, J., 2005. Kaveri: Delivering the indus java program slicer to eclipse. In: International Conference on Fundamental Approaches to Software Engineering. Springer, pp. 269–272.
Jiang, X., Zhou, Y., 2012. Dissecting android malware: characterization and evolution. In: 2012 IEEE symposium on security and privacy. IEEE, pp. 95–109.
Lam, P., Bodden, E., Lhoták, O., Hendren, L., 2011. The soot framework for java program analysis: a retrospective. In: Cetus Users and Compiler Infastructure Workshop (CETUS 2011), 15, p. 35.
Lashkari, A.H., Kadir, A.F.A., Gonzalez, H., Mbah, K.F., Ghorbani, A.A., 2017. Towards a network-based framework for android malware detection and characterization. In: 2017 15th Annual Conference on Privacy, Security and Trust (PST). IEEE, pp. 233–23309.
Lashkari, A.H., Kadir, A.F.A., Taheri, L., Ghorbani, A.A., 2018. Toward developing a systematic approach to generate benchmark android malware datasets and classification. In: 2018 International Carnahan Conference on Security Technology (ICCST). IEEE, pp. 1–7.
Lee, K., Park, H., 2019. Malicious adware detection on android platform using dynamic random forest. In: International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing. Springer, pp. 609–617.
Li, L., Bartel, A., Bissyandé, T.F., Klein, J., Le Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., McDaniel, P., 2015. Iccta: Detecting inter-component privacy leaks in android apps. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1. IEEE Press, pp. 280–291.
Li, L., Bissyandé, T.F., Octeau, D., Klein, J., 2016. Droidra: Taming reflection to support whole-program analysis of android apps. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. ACM, pp. 318–329.
Liang, C.-J.M., Lane, N., Brouwers, N., Zhang, L., Karlsson, B., Chandra, R., Zhao, F., 2013. Contextual fuzzing: automated mobile app testing under dynamic device and environment conditions. Microsoft. Microsoft, nd Web.
Lindorfer, M., Neugschwandtner, M., Weichselbaum, L., Fratantonio, Y., Van Der Veen, V., Platzer, C., 2014. Andrubis–1,000,000 apps later: A view on current android malware behaviors. In: 2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS). IEEE, pp. 3–17.
Liu, J., Li, Y., Tan, T., Xue, J., 2017. Reflection analysis for java: Uncovering more reflective targets precisely. In: Software Reliability Engineering (ISSRE), 2017 IEEE 28th International Symposium on. IEEE, pp. 12–23.
Liu, X., Liu, J., Zhu, S., Wang, W., Zhang, X., 2019. Privacy risk analysis and mitigation of analytics libraries in the android ecosystem. IEEE Trans. Mobile Comput.
Machiry, A., Tahiliani, R., Naik, M., 2013. Dynodroid: an input generation system for android apps. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, pp. 224–234.
LoopFinder, https://www.sable.mcgill.ca/soot/doc/soot/jimple/toolkits/annotation/logic/LoopFinder.html.
Mahmood, R., Esfahani, N., Kacem, T., Mirzaei, N., Malek, S., Stavrou, A., 2012. A whitebox approach for automated security testing of android applications on the cloud. In: Proceedings of the 7th International Workshop on Automation of Software Test. IEEE press, pp. 22–28.
Mao, K., Harman, M., Jia, Y., 2016. Sapienz: multi-objective automated testing for android applications. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. ACM, pp. 94–105.
Mirzaei, N., Garcia, J., Bagheri, H., Sadeghi, A., Malek, S., 2016. Reducing combinatorics in gui testing of android applications. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). IEEE, pp. 559–570.
Monkey. http://developer.android.com/tools/help/monkey.html.
Nath, S., Lin, F.X., Ravindranath, L., Padhye, J., 2013. Smartads: bringing contextual ads to mobile apps. In: Proceeding of the 11th annual international conference on Mobile systems, applications, and services. ACM, pp. 111–124.
Octeau, D., Luchaup, D., Dering, M., Jha, S., McDaniel, P., 2015. Composite constant propagation: application to android inter-component communication analysis. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1. IEEE Press, pp. 77–88.
Rasthofer, S., Arzt, S., Miltenberger, M., Bodden, E., 2016. Harvesting runtime values in android applications that feature anti-analysis techniques. In: Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS).
Rasthofer, S., Arzt, S., Triller, S., Pradel, M., 2017. Making malory behave maliciously: targeted fuzzing of android execution environments. In: Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on. IEEE, pp. 300–311.
Rastogi, V., Chen, Y., Enck, W., 2013. Appsplayground: automatic security analysis of smartphone applications. In: Proceedings of the third ACM conference on Data and application security and privacy. ACM, pp. 209–220.
Ruiz, F., Oct 2012. Fakeinstaller leads the attack on android phones. website of mcafee labs. https://blogs.mcafee.com/mcafee-labs/fakeinstaller-leads-the-attack-on-android-phones.
Shah, G., Shah, P., Muchhala, R., 2014. Software testing automation using appium. Int. J. Curr. Eng. Technol. 4 (5), 3528–3531.
Taheri, L., Kadir, A.F.A., Lashkari, A.H., 2019. Extensible android malware detection and family classification using network-flows and api-calls. In: 2019 International Carnahan Conference on Security Technology (ICCST). IEEE, pp. 1–8.
UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.html.
Vidas, T., Christin, N., 2014. Evading android runtime analysis via sandbox detection. In: Proceedings of the 9th ACM symposium on Information, computer and communications security. ACM, pp. 447–458.
VirusShare. https://virusshare.com/.
VirusTotal. www.virustotal.com.
Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F., 2003. Model checking programs. Automated Softw. Eng. 10 (2), 203–232.
Wang, W., Li, Y., Wang, X., Liu, J., Zhang, X., 2018. Detecting android malicious apps and categorizing benign apps with ensemble of classifiers. Future Gener. Comput. Syst. 78, 987–994.
Wang, W., Zhao, M., Wang, J., 2019. Effective android malware detection with a hybrid model based on deep autoencoder and convolutional neural network. J. Ambient Intell. Humanized Comput. 10 (8), 3035–3043.
Wang, X., Zhu, S., Zhou, D., Yang, Y., 2017. Droid-antirm: taming control flow anti-analysis to support automated dynamic analysis of android malware. In: Proceedings of the 33rd Annual Computer Security Applications Conference. ACM, pp. 350–361.
Wei, F., Roy, S., Ou, X., et al., 2014. Amandroid: a precise and general inter-component data flow analysis framework for security vetting of android apps. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, pp. 1329–1341.
Weiser, M., 1981. Program slicing. In: Proceedings of the 5th international conference on Software engineering. IEEE Press, pp. 439–449.
Xposed Module Repository, 2017. http://repo.xposed.info/.
Zadgaonkar, H., 2013. Robotium Automated Testing for Android. Packt Publishing Ltd.
Zhang, Y., Li, Y., Tan, T., Xue, J., 2018. Ripple: reflection analysis for android apps in incomplete information environments. Software: Pract. Exp. 48 (8), 1419–1437.
Zhauniarovich, Y., Ahmad, M., Gadyatskaya, O., Crispo, B., Massacci, F., 2015. Stadyna: addressing the problem of dynamic code updates in the security analysis of android applications. In: Proceedings of the 5th ACM Conference on Data and Application Security and Privacy. ACM, pp. 37–48.
Zhou, Y., Jiang, X., 2012. Dissecting android malware: characterization and evolution. In: 2012 IEEE symposium on security and privacy. IEEE, pp. 95–109.

Jyoti Gajrani received the M.Tech. degree in Computer Engineering from the Indian Institute of Technology, Bombay in 2013. She is currently pursuing the Ph.D. degree (Part-time) in computer science with Malaviya National Institute of Technology, Jaipur, under the supervision of Prof. Manoj Singh Gaur from Indian Institute of Technolgy, Jammu and Dr. Meenakshi Tripathi from Malaviya National Institute of Technology, Jaipur. She is working as Assistant Professor (Full-time) in Computer Engineering department at Govt. Engineering College, Ajmer since 2006. Her research interests include the area of security and privacy, with a special emphasis on security in Android.

Umang Agarwal is undergraduate student at Govt. Engineering College Ajmer. Along with programming, his research interest includes cyber security, malware analysis. He has participated in various hackathons and programming contests conducted at national level. He has also developed Android apps for cultural and sports events in his college.

Dr. Vijay Laxmi received her Master's in Computer Science and Engg. from Indian Institute of Technology Delhi and Ph.D. from University of Southampton, UK. She has been a faculty in Department of Computer Science and Engineering, Malaviya National Institute of Technology Jaipur, India. Her research interests include Information security, Malware analysis, Security and QoS provisioning in wireless Networks.

Prof. Bruhadeshwar Bezawada has received his B.E. (ECE, Distinction) from Osmania University, India; M.S. (Electrical and Computer Engineering) and PhD from the Dept. of Computer Science and Engineering at Michigan State University, USA. He is currently Professor of Computer Science at Mahindra Ecole Centrale, India. Previously he was an Assistant Professor at the International Institute of Information Technology, India, Visiting Scholar at Nanjing University, China, University Professor (visiting) at K.L. University, India, and Senior Research Associate at Colorado State University, USA. His research interests are in the areas of information security, Internet-of-Things, Cyber-physical systems, cloud computing security and malware analysis. His paper has won the Best Student Paper award at DBSec 2019.

Prof. Manoj Singh Gaur completed his Master's degree in Computer Science and Engineering from Indian Institute of Science Bangalore, India and Ph.D. from University of Southampton, UK. Prof. Gaur has been a faculty in Department of Computer Science and Engineering, Malaviya National Institute of Technology Jaipur, India and currently Director, IIT Jammu. His research areas include Networks-on-Chip, Computer and network security, Multimedia streaming in wireless networks.

Dr. Meenakshi Tripathi is currently an associate professor at Computer Science and Engineering Department, Malaviya National Institute of Technology Jaipur, India. She has been teaching UG and PG courses in the area of mobile computing and wireless communications. Her research interests are in the areas of security, wireless sensor networks and software defined networks. She is also a member of IEEE and ACM.

Akka Zemmari has received his Ph.D. degree from the University of Bordeaux, France, in 2000. He is an associate professor in computer science since 2001 at University of Bordeaux, France. His research interests include machine and deep learning, randomized algorithms, security and distributed algorithms and systems.