Android App Analysis
University of Paderborn
Warburger Str. 100
33102 Paderborn

PAndA^2
Final Documentation

Paderborn, March 31, 2016

Authors:

Abhinav Solanki
Anand Devarajan
Arjya Shankar Mishra
Fabian Witter
Felix Pauck
Monika Wedel
Pham Thuy Sy Nguyen
Ram Kumar Karuppusamy
Sriram Parthasarathi

Contents

1 Introduction

The operating system Android becomes more and more widespread in these days. The number of Android applications available in the Google Play Store grows continuously. But with increasing number of Apps the danger of malicious Apps that try to cheat the user grows, too. Therefore, it becomes more and more necessary to investigate the applications and to find out whether they are trustworthy or not.

To do so we developed PAndA^2 . The name PAndA^2 reflects the meaning **P**aderborn **And**roid **A**pp **A**nalysis. Our tool offers a framework for analyzing Android applications in which different analyses can be integrated. Together with the tool we already deliver three different analyses and provide the possibility to easily integrate more of these. Moreover, our tool offers a graphical user interface for a good usability.

In the following document we give some details about PAndA^2 and the three analyses. Therefore, in Section 2 we shortly remain the reader of the architectural structure of our tool which was described in more detail in the Architecture Document. The next section describes details about the way we implemented PAndA^2 . Here the main focus rests on concepts and algorithms used. It consists of six subsections of which Section 3.1 describes the `Client` component and Section 3.2 the `CoreServices`. Afterwards, the `Enhancer`, which is used in the three analyses, is described in Section 3.3. And finally, the Sections 3.4 to 3.6 deal with the three analyses itself. We offer two analyses that deal with the application's usage of Android permissions. The first analyzes the permission usage inside an application (Section 3.4) and the second between different applications (Section 3.5). The third analysis tracks information flow between Android permissions (Section 3.6).

As already mentioned PAndA^2 provides the opportunity to integrate further analyses. How this can be done as well as how the user interface can be changed is described in Section 4. Our tool currently supports the Android API version 22. The question how to adapt the API level PAndA^2 supports, is also treated in Section 4. Section 5 then deals with the actions we took to assure a certain level of quality. We also compared our tool and in particular the three analyses with some other tools to evaluate how PAndA^2 performs in comparison to other Android analysis tools. The concrete setup as well as the results are described in Section 6. A description of possible options for future refinement of PAndA^2 follows in 7. Finally, Section 8 gives a short conclusion derived from the previous chapters.

2 Architecture Overview

This chapter is used to sum up the underling architecture of PAndA^2 . A fully detailed description of the architecture can be found in our Architecture Document. PAndA^2 is not only a tool but also a framework. On one hand we deliver a tool that supports three different analyses right away:

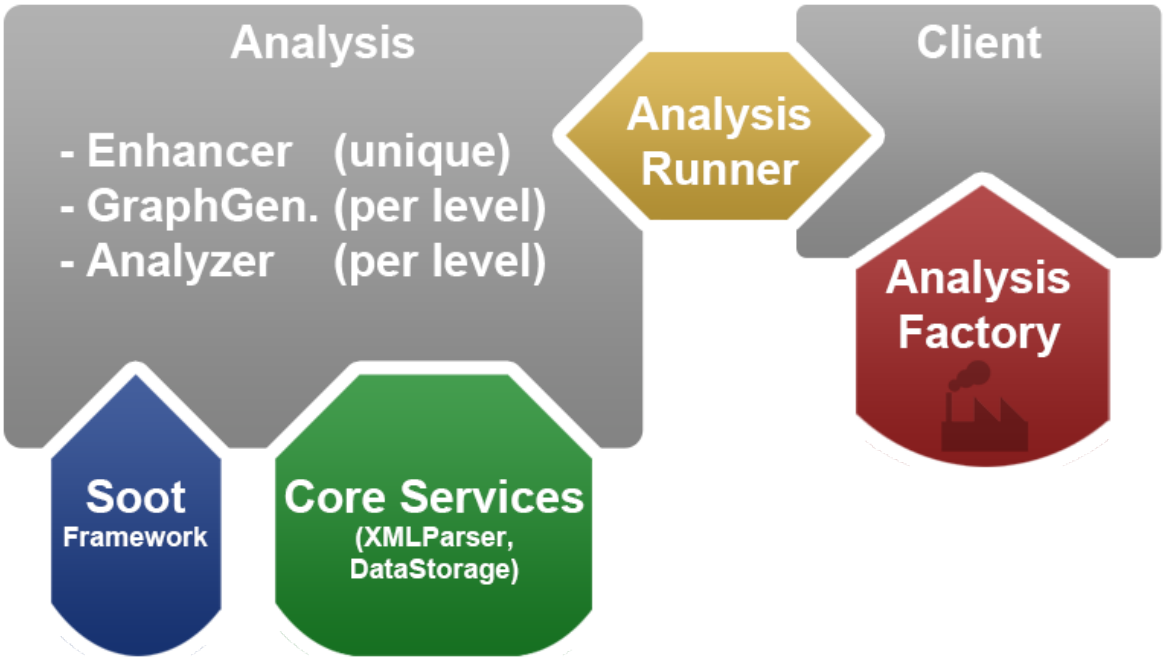

Figure 1: Architecture Overview

- Intra-App Permission Usage Analysis (see Section 3.4)
- Inter-App Permission Usage Analysis (see Section 3.5)
- Intra-App Information Flow Analysis (see Section 3.6)

On the other hand we deliver a framework that can easily be extended to support any other analysis or to use another user interface (see Section 4).

In this chapter we will provide an overview over the architecture of our tool and by that explain the general workflow of any analysis executed with PAndA^2 .

Figure 1 provides a complete summary of the tool's architecture. The biggest component illustrated in that figure is the `Analysis` component, it contains the analysis itself. The second largest component shown in Figure 1 is the `Client`. This component contains the whole interactive part of PAndA^2 , e.g. the Commandline Interface and the Graphical User Interface. All other components visualized in Figure 1 belong to the `Core` package. This package contains all interfaces which loosely couple the `Client` component with the `Analysis` component, namely the `AnalysisFactory` and the `AnalysisRunner`. The component that accesses the Soot framework which can be used in different steps of any analysis belongs to the `Core` package as well. Last but not least the `CoreServices` belong to that package, too. These services can be used by several other components of the tool and provide a large variety of functionality (see Section 3.2).

In order to start an analysis the `Client` has to call the `AnalysisFactory` which will create the `Analysis` itself. Once the `Analysis` is created the `Client` can use the `AnalysisRunner` to execute this `Analysis`. At the end of any analysis the `AnalysisRunner` will reply an `AnalysisResult` to the `Client`.
All in all this makes sure that the `Client` and the `Analysis` components are independent from each other. To start an analysis the `Client` has to hand over only a minimum of input information to the factory. And any `AnalysisResult` provides a textual or graphical result representation in form of an HTML-Document.

Via a closer look at the three sub-components of the `Analysis` component, namely the `Enhancer` (see Section 3.3), the `GraphGenerator` and the `Analyzer`, the workflow of any analysis that follows our implementation principle will be described in the following. While the `Enhancer` is unique, the `GraphGenerator` and the `Analyzer` can be different from `Analysis` to `Analysis`. The `Enhancer`'s task is to enhance the source code representation which is extracted from an App's .apk file. This representation contains all information about the structure of an App. The extraction process itself is performed by the underlying Soot framework, which directly provides a tree like data structure for the source code representation. In order to enhance this representation the `Enhancer` will use several `CoreServices` and by that add more information to it.

All of the built-in analyses and most of all analyses work on graphs. Therefore, the next step during any analysis is done by the `GraphGenerator`, which will build a graph that suits the specific type of analysis. The `Analyzer` will then perform the analysis itself on the previously constructed graph. Then it will compute the `AnalysisResult` and thereby finish the analysis process.

The design of our architecture ensures the possibility of extension. Thus, it provides a head-start to the development of a new analysis. Furthermore, it allowed us as a project group to work on different parts in parallel.

3 Implementation Details

This section describes important concept and algorithms realized during the implementation of PAndA^2 . Section 3.1 starts with the `Client` component. This is then followed by the description of the `CoreServices` in Section 3.2. Afterwards, Section 3.3 deals with the `Enhancer`, which is used in the three analyses. These analyses follow right after the `Enhancer` in the Sections 3.4 to 3.6. The first analysis is the Intra-App Permission Usage Analysis in Section 3.4 followed by the Inter-App Permission Usage Analysis in Section 3.5. Finally, Section 3.6 describes the Intra-App Information Flow Analysis.

3.1 Client

This paragraph provides a short introduction about the *Client* and the way the related class was described in the architecture document. It is an abstract class which acts as a stub between the Graphical User Interface (*GUI*), Command Line Interface (*CLI*) and Analysis Factory. A `Client` class contains all the methods to provide input, trigger analysis and provide results back. Client class has two subclasses, one for GUI and another for CLI. These subclasses provide implementations of all the abstract methods present in `Client` (Refer Figure 2).

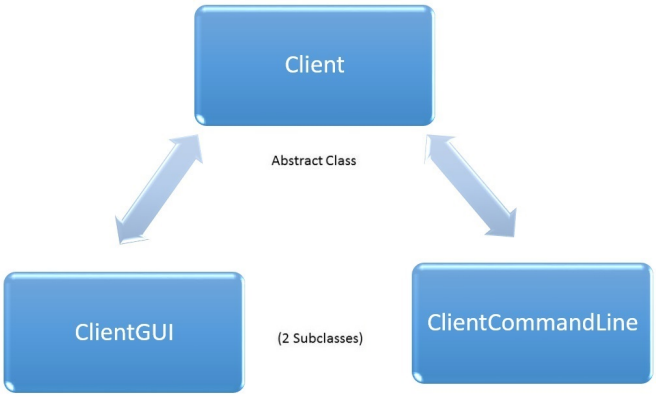

Figure 2: Client Model

The two subclasses of `Client` are

1. `ClientGUI.`
2. `ClientCommandLine.`

Before explaining in the next subsections about CLI and GUI of our tool, this section provides a basic idea with respect to

- How the user can able to start the application.
- What changes we made with respect to the way we are calling the business logic classes now.

Application can be started either by using GUI or CLI based on the user preference (Refer Figure 3). The user can start the tool either by clicking the application jar file for using the GUI or by using this command "java -jar PAndA2.jar <inputs>" in the command prompt for using CLI.

Certain changes had been made in the codebase compared to the decisions which we had taken in the architecture document with the way CLI or GUI interfaces getting triggered using the application main method.

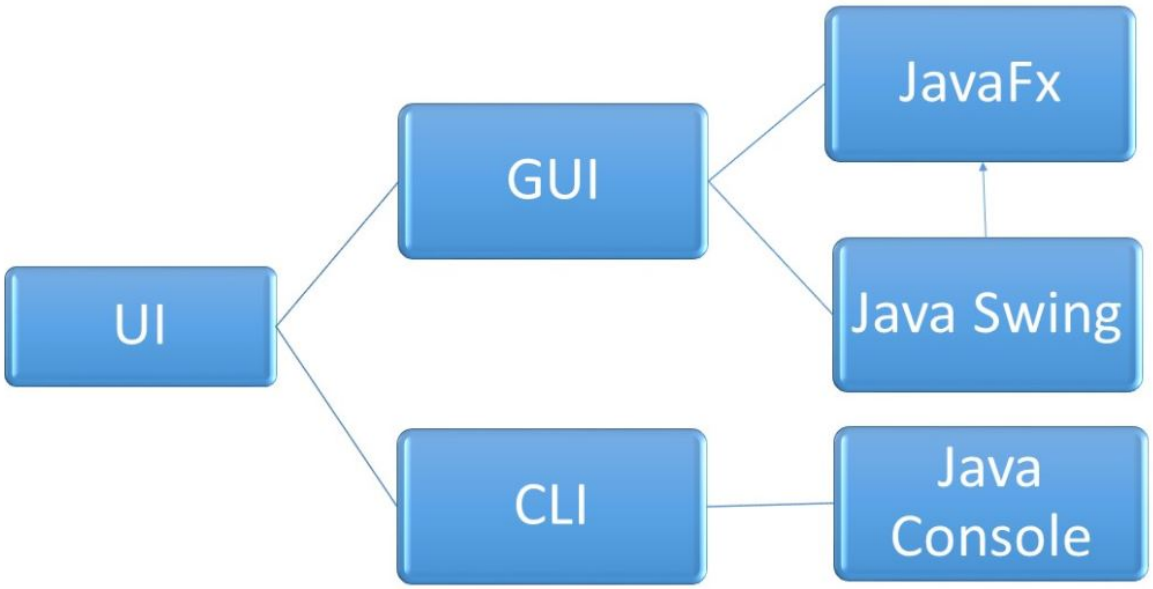

Figure 3: Triggering User Interface

Previously, based on the architecture document the application's CLI or GUI will be triggered based on the business logic classes (`ClientGUI` and `ClientCommandLine`). That is in simple terms, the views (`CommandLine` or `GUI` classes) and its related functionalities are triggered initially based on the business logic classes (Models). But in general, business logic classes should not be used for deciding which interface (`CommandLine` or `GUI`) should be shown to the user. It has to be the other way around. So we have made the necessary changes with respect to the way in which the initial calls are made for selecting the user interface (*CLI*/*GUI*) based on the input parameter provided by the user.

Now the selection of the user interfaces will be decided based on the following criteria (Refer Figure 4)

- If the input parameter is none, then GUI will be triggered.
- If there are input parameters, then CLI will be triggered.

This decision is now taken in the main method which is present in the `ApplicationEntryPoint` class of our tool itself, so that now the business logic classes are totally de-coupled with the user interface selection.

3.1.1 Graphical User Interface (GUI)

As our tool is developed in Java, so we have to come up with a technology supporting Java to interact with the User. Out of bundle of options available initially we decide to go with *Java Swing* which is quite robust, consists of lots of supporting functions, and later adapting to our

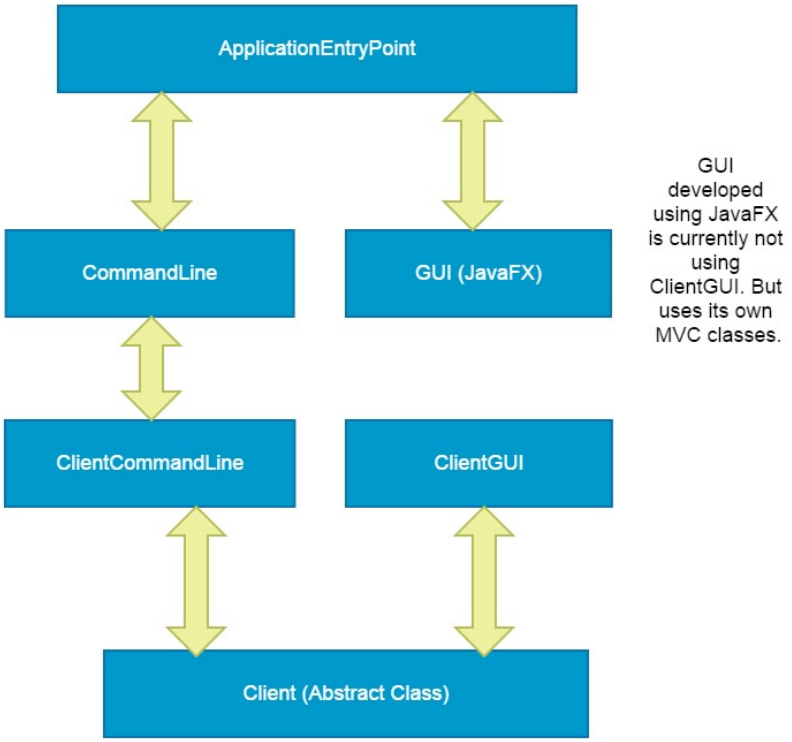

Figure 4: Client Architecture

growing needs to handle the output display in a simple, fruitful manner we shifted to *JavaFX*. The basic motto or reason behind shifting to *JavaFX* was the increasing demand of tool to display graphical results specifically in COMPARISON mode of the Intra-App Information Flow Analysis. A brief description of *JavaFX* has been given in the subsequent section.

JavaFX It provides us with a set of graphics and media packages that enables to design rich client applications. *JavaFx* applications can use *JavaAPI* libraries to access native system capabilities which is an advantage over others. *JavaFX* APIs are available as a fully integrated feature of the Java SE Runtime Environment and the Java Development Kit. On the top of it, some of the *JavaFx* advantages which make us migrate are as follows

1. Swing interoperability - This means the existing *Java Swing* applications can be updated to *JavaFX* to access rich graphic features and other contents.
2. Built-in UI controls and *CSS* - It provides almost all the major UI controls required to develop a full featured application.

3. Components can be skinned with standard Web technologies such as *CSS*.
4. *HTML* Content - *JavaFX* brought the ability to render *HTML* content in Java applications by providing a user interface component that has web view and full browsing functionality.

Starting Analysis Using `GUI` As discussed earlier in this section, for the ease of the user interaction with the tool we have a graphical user interface built in *JavaFX*. Here in this section, a brief description of the functions involved, triggering of `GUI`, and basic functioning of `GUI` is covered.

Tool Initialisation The tool can be started in two ways, either by passing the `CommandLine` arguments which trigger command line interface or when we leave it blank will trigger the `GUI` as shown in the Figure 5. Both the interfaces provides certain set of functionalities that has been discussed in the previous section and more elaboration is given in subsequent sections. Here in this section, the process of analysing Android applications via `GUI` is explained.

Parameters Input Now we can start a new analysis by clicking on *New* icon on the toolbar, a dialog box appears in front as shown in Figure 6 and Figure 7 which requires some Input from the user, e.g. *Level*, *Mode*, *Path* of `.apk` files, etc. The Input parameters as displayed in Figure 7 depends on the input user had given in previous page shown in 6. On the basis of the input, the respective Analysis will be carried out. Some of the basic validations on the Input will be carried in this phase such as checking fingerprints of the `.apk` file for the Version number, etc. and user can see them right after entering at run time.

Start Analysis When done with all the required input field, clicking on *Finish* button will transfer the control to the `Client` which triggers the *Analysis*. *Analysis* will run in background, user can see the rotating icon on the bottom left of the screen which states that the tool is processing. Once the tool is finished with Analysis, icon state will be changed and the result generated during analysis will be displayed to the user.

Result Representation Once the Analysis is finished, the result will be shown to the user with all the predefined Legend and Statistics of the analysis for the better understanding of the User. Analysis result will be shown to the user in Textual manner as in Figure 8 and Graphical manner as in Figure 9 in TabPane. TabPane contains three sub TabPanes for *textual*, *graphical* and *Messages* respectively. User will be having the option to save the result with predefined extension `.pa2`. User need to install *Graphviz* which is available free of source. More about it, is mentioned in *InstallationRequirements*.

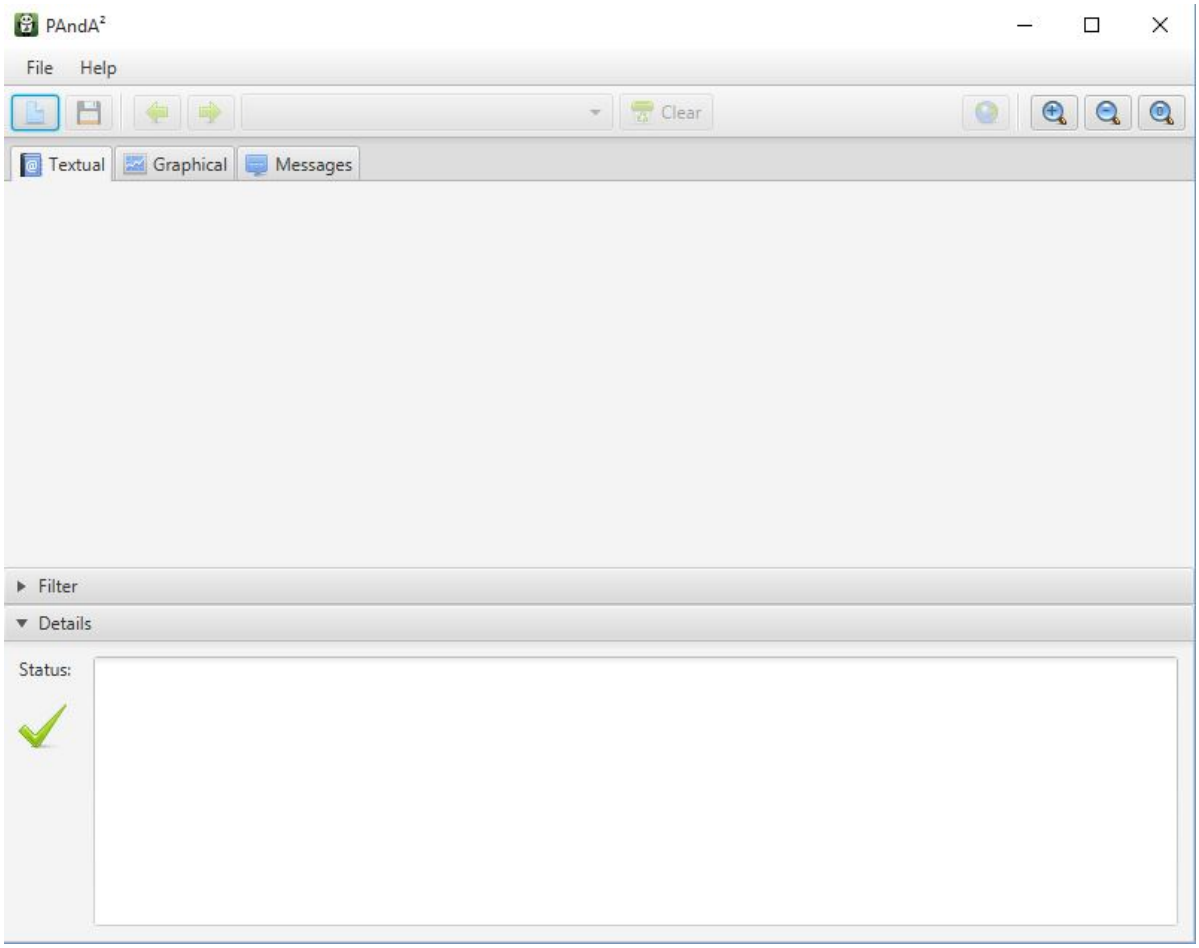

Figure 5: GUI: Home

Result Filters Result field can be huge at times, therefore filters functionality is provided to filter the result.User can filter the result in certain categories depending on the type of analysis chosen while setting up the parameters. Result can be filtered on *Detaillevel* (*APP*, *COMPONENT* etc.) or Permission filters (REQUIRED, MAYBE_MISSING etc.). Filters for $PermissionUsage(IntraApp-Level1)$ is shown in Figure 10.

Status can be seen at the bottom of the window throughout the *Analysis*. *UImessages* will be displayed on the field present on the bottom right of the window and also the exceptions in case if encountered any.

Saving and Comparing Result After the analysis is finished, user will see either the analysis result of current `.apk` file or its comparison with previous saved result `.pa2` file providing that a previous saved result was provided to the tool while setting input parameters of analysis. User can also opt for saving the current comparison result.

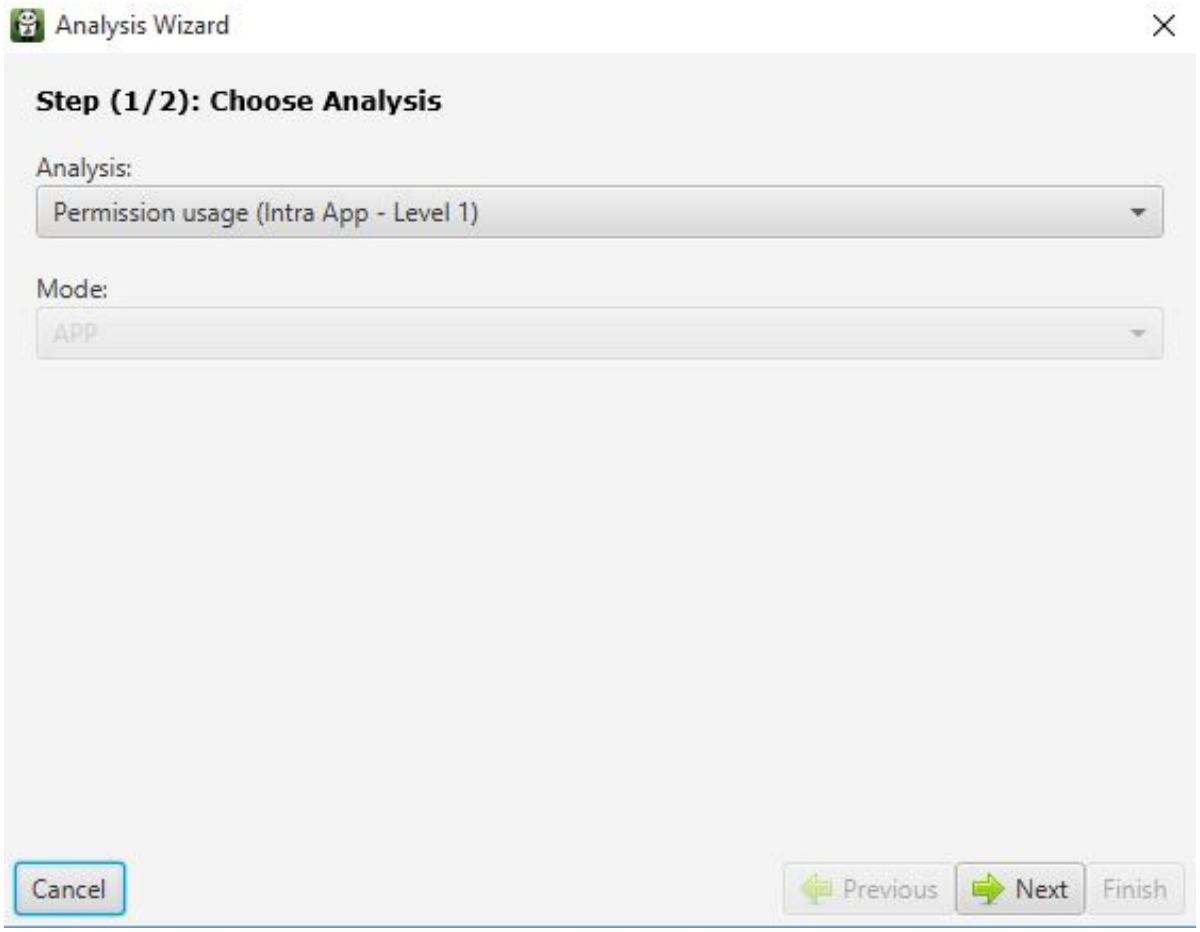

Figure 6: Analysis Wizard: Page1

Open Previous Result Apart from starting a fresh analysis, user can also opt to open a previous saved result `.pa2` file present in the system. The saved result will contain both the *textual* and *graphical* representations in *SUMMARY* as well as *COMPARISON* mode.

3.1.2 Command Line Interface

As mentioned in the previous section as well as in the architecture document, the Command Line Interface (CLI) functionality of our application can be triggered when the user provides input parameters (refer Table 1) in the command prompt (refer Figure 11). This section covers the various functionalities which are possible using this interface and the third party libraries which we have used for implementing those functionalities. The `CommandLine` class which is used for the following functionalities,

- Handles the user input parameters.

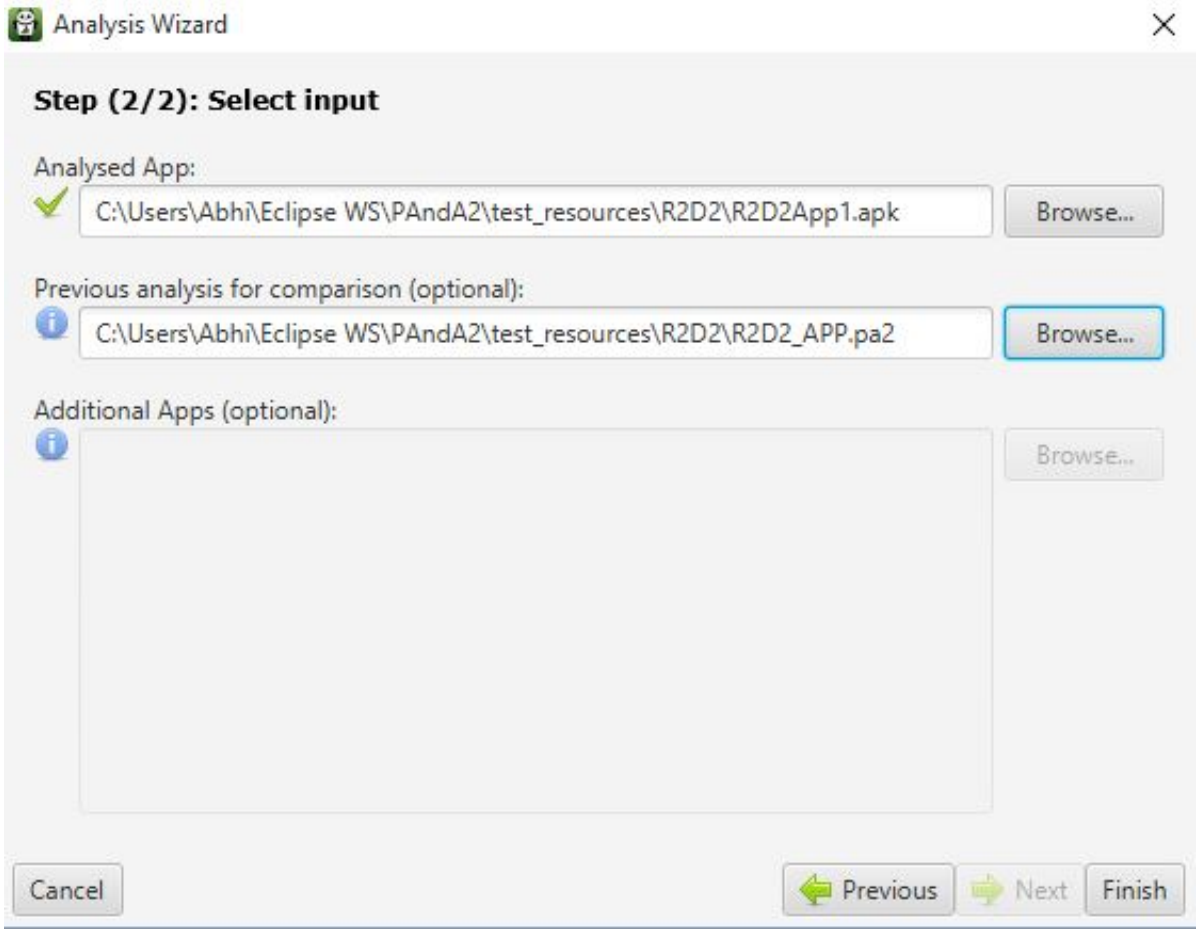

Figure 7: Analysis Wizard: Page2

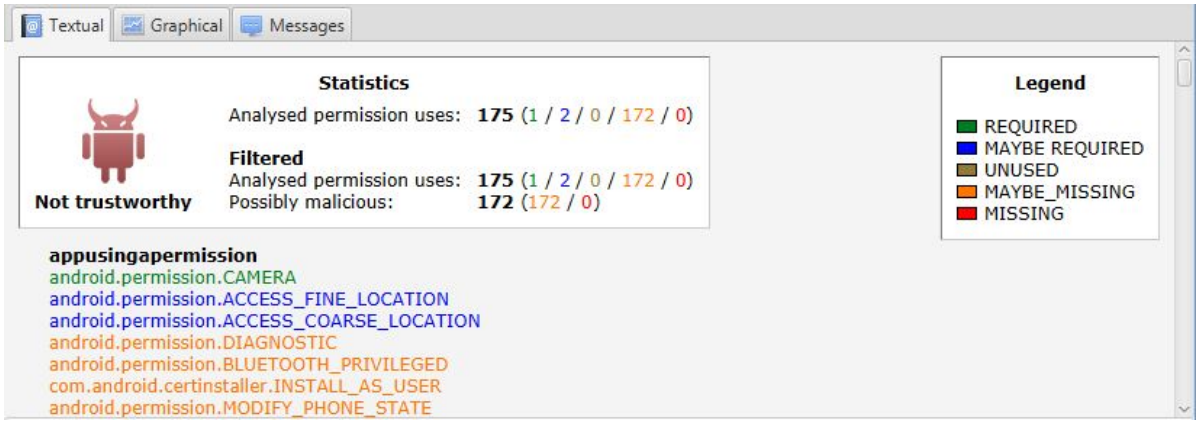

Figure 8: GUI: Result

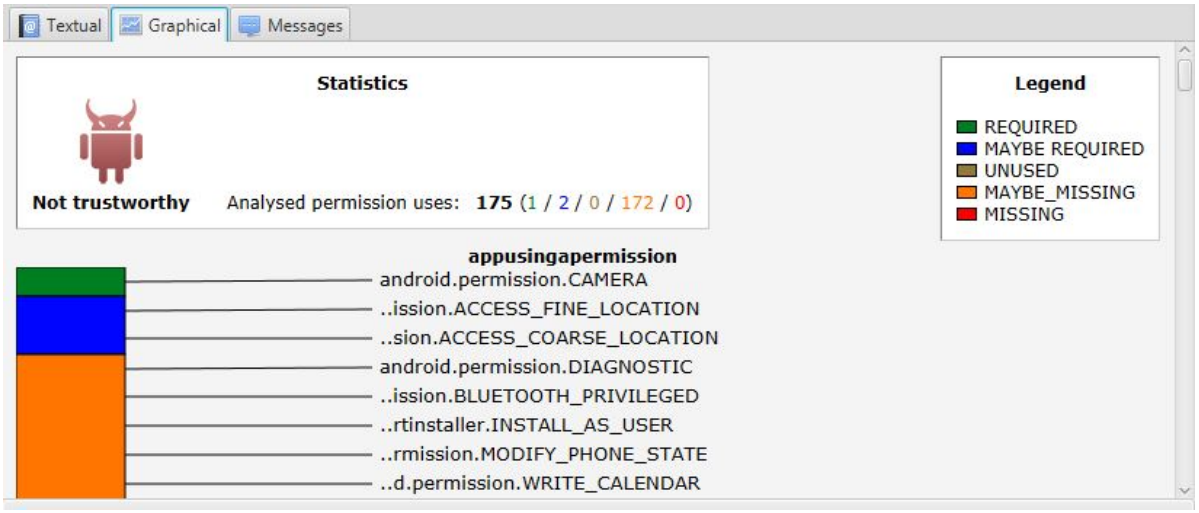

Figure 9: GUI: Result2

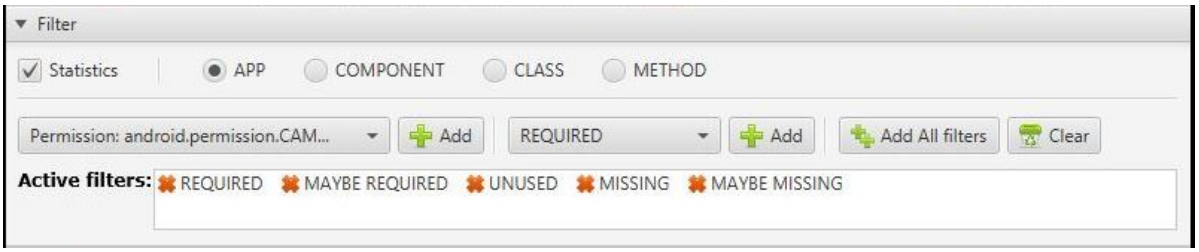

Figure 10: GUI: Filter

  - Parsing the user input parameters.
  - Validating the user input using Business logic class.
- Start Analysis.
- Handles the analysis result.
  - Displaying the analyzed result.
  - Filtering the analyzed result.
  - Saving the analyzed result in the user selected file path.
  - Comparing the previous result for analysis.

Available user input parameters for CLI are shown in Table 1.

Parsing the user input parameters Parsing the user input parameters which was mentioned in the previous section is handled using the *JCommander* [1]library. The high level purpose of

[1]`http://jcommander.org/`

```
C:\Personal\Development\PG-A3-Workspace\PAndA2_Latest\build>Java -jar PandA2.jar -opt help
Usage: <main class> [options]
  Options:
    -ci, -compareinput          Previous analysis file path for 'Comparison
                                Mode'. COMPULSORY if you want to use this mode.
    -dl, -detaillevel           Provide the 'Detail Level'. Available
                                options,DetailLevelLvl1 ==>(APP/COMPONENT/CLASS/METHOD),DetailLevelLvl2a
                                ==>
                                (RES_TO_RES/COMPONENT/STATEMENT),DetailLevelLvl2b ==> (APP/COMPONENT)
    -fp, -filepath              File path for Viewing previous result/Saving
                                current analysis result. COMPULSORY for both these
                                scenarios
    -f, -filter                 Provide the 'Result Filters'. Can give more than
                                one value by giving space between them. Some filter
                                options are:Please choose any one of these
                                options,Filters
                                ==>(REQUIRED/MAYBE_REQUIRED/UNUSED/MISSING/MAYBE_MISSING)
    -i, -input                  Initial APK file path. COMPULSORY for all the
                                levels.
    -l, -level                  Analysis Level: COMPULSORY for all the levels.
                                Please choose any one of these options,Level1
                                ==>(1/LEVEL
                                1/LEVEL1/INTRA-APP-PERMISSION/INTRA-PERM/INTRA-APP-PERM),Level2a ==> (2A/LEVEL
                                2A/LEVEL2A/INTRA-APP-INFO-FLOW/INTRA-INFO-FLOW/INTRA-IF),Level2b ==> (2B/LEVEL
                                2B/LEVEL2B/INTER-APP-PERMISSION/INTER-APP-PERM/INTER-PERM)
    -lm, -levelmode             Analysis Level Specific Mode: COMPULSORY for
                                LEVEL2B Scenario: Please choose any one of these
                                options,(APP/ALL).
    -nn, -nonnative             Non Native APK file path. Optional field to give
                                for LEVEL2B scenario.
    -r, -result                 Result Representation Option. Available
                                options,Save ==>(S/SAVE/SA),View ==> (V/VIEW/VI)
                                Default: VIEW
    -x, -exit, -t, -terminate   Terminate the program after the analysis
                                execution.
                                Default: true
    -ts, -testscenario          Whether you want the testing scenario to be
                                enabled?If so use this option so that result won't be
                                printed in the console.
                                Default: false
    -tstat, -teststats          Showing test statistics in the result.
                                Default: true
    -toolopt, -opt, -option     Tool options: Please choose any one of these
                                options,Analysis ==>(ANALYZE/ANA/ANALYSIS/A),LOAD ==>
                                (LOAD/L),HELP ==> (HELP, H, -HELP, -H, MAN, -MAN).
                                Default: ANALYSIS
    -v, -view                   Result View Option. Available options,Textual
                                ==>(T/TEXT/TEXTUAL),Graphical ==> (G, GRAPH, GRAPHICAL),Message ==> (M, MSG,
                                MESSAGE)
                                Default: TEXTUAL
```

Figure 11: Client CommandLine Interface

this library is used for creating the instance of the `UserInput` class which actually used for persisting the user input values such as levels, apk input path, etc. So the tool is configured by using the *JCommander* parameter annotation for the variables present in that class. For example, the level mode variable is binded using the parameter annotation as follows,

```
@Parameter(names = { "-l", "-level" },
description = "Analysis Level", required = true,
converter = LevelConverter.class)
private Level level;
```

So from the above lines of the code, we can understand that this parameter is required and should be given by the user using the notation such as *−l* or *−level*. The value will be String value and it will be converted to instances of `Level` enum using the custom converter classes.

With respect to the custom converter classes which is used for converting the value of the String class to the instance of our custom classes or enums such as `Level`, `Mode` etc. *JCommander* will throw an exception if the user input doesn't match one of the following scenarios,

- Input parameter is incorrect or not available.

Parameter	Description	Compulsory
-opt (or) -option	Tool Options	N
-l (or) -level	Analysis Level	Y
-i (or) -input	Initial APK File Input	Y
-lm (or) -levelmode	Level Specific Mode	Y(-l is 2B)
-nn (or) -nonnative	Non native APK Files	N
-ci (or) -compareinput	Previous analysis file path	N
-r (or) -result	Result option (Save (or) View)	N
-v (or) -view	Result View option	N
-fp (or) -filepath	Path for saving the result	N
-f (or) -filter	Initial filter values	N
-dl (or) -detaillevel	Initial detail level values	N
-x (or) -exit (or) -t (or) -terminate	Option for program termination	N
-ts (or) -testscenario	Field for printing the output or not	N
-tstat (or) -teststat	Statistics should be present in the output	N

Table 1: List of CLI parameters.

- Required parameter is not given in the input.
- The input value is not correct for the respective parameter.

Once the input parameters are parsed using the *Jcommander* and the initial validation was successful as mentioned in the above paragraphs, then the instance of the `User Input` class will be updated with the possible values in their instance variables with the initial set of values given by the user. This instance variable of the `User Input` class will be further used for analysis.

Validating the user input using Business Logic class From a command line interface perspective, we will do the second step of validation through our common business logic class (*Client*) for handling the following scenario's validation,

- In Inter App resource usage (level2b) scenario, the level specific mode value should be given by the user. Else validation will be failed.
- In comparison scenario, the previous analysis result parameter value should be given. Else the user input won't be considered as valid.
- For saving the result, the respective file path along with the valid file name should be given by the user. Else validation is failed for this scenario.

Start Analysis Once the input parameters given by the user are valid, then the analysis will be performed with the help of the methods present in the business logic class (`ClientCommandLine`) which was specific to the CLI. The input of the analysis was passed to the different instance of the `AnalysisFactoryLvl` classes based on the level selected by the user. By using the method presented in that `AnalysisFactoryLvl` instance, the `analysislist` is generated which will be used for getting the result after analysis by calling the method using the `AnalysisRunner` instance. If the analysis executed successfully without any exception, then we will get the instance of the level based `AnalysisResult` classes which will be further used for displaying or saving the result based on the user selection. If we got an exception during the analysis or the error message present in the instance of the level based `AnalysisResult` classes, then the error message will be displayed in the CLI. So in short the functionality described in this paragraph can be described as follows,

- Generate the analysis based on the user configuration and trigger the exceution of the same.
- If analysis was successful, then show the result based on the user configuration.
- Else show the error message.

Displaying the analyzed result Based on the selected user input parameter, there are three ways the result can be displayed with the help of the command line interface. Three different types of result supported by our tool are,

- Textual.
- Graphical.
- Message.

Textual Textual analysis result will be displayed in the CLI similar to the way the statements are getting printed in the console for the Java applications. Depends on the selected level, and their corresponding Detail Levels (APP, COMPONENT etc) and the Permission filters (REQUIRED, MAYBE_REQUIRED etc) the result contains information about the same. The display selection of this result is selected by the CLI by default if the user doesn't give any specific input parameter for the same. This result displayed to the user contains all the necessary information in HTML and CSS format which will be used for displaying it in the GUI. So from CLI perspective, we are parsing the textual result to remove the HTML contents using the third party library called "*jsoup*" [2].

Using this library ("*jsoup*") we are removing the HTML tags and attributes and other CSS information, so that currently we are printing the output of the result in the console. The class named `HtmlParser` is used for parsing the generated textual result and then for displaying the same in CLI. The following Figure 12 illustrates how the textual result using CLI.

[2] `http://jsoup.org/`

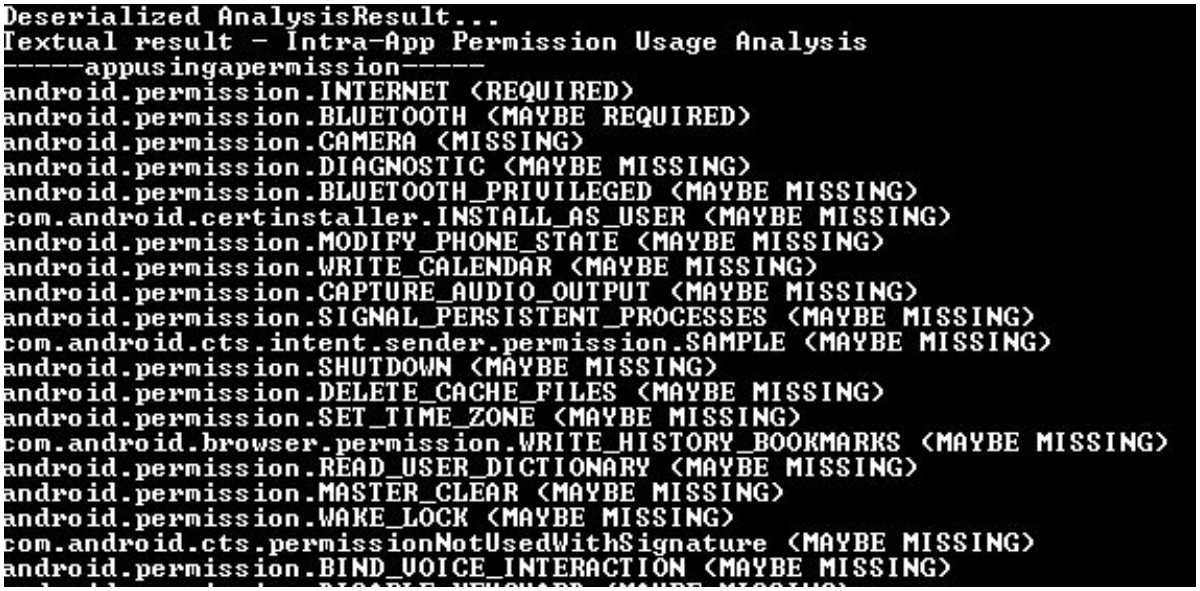

Figure 12: CLI Textual Result

Graphical In general, GUI will be launched to show the graphical result. So basically once the user gives the parameter(-v) as "GRAPHICAL" for viewing the result, the validation methods in the business logic class (`ClientCommandLine`) will decide the user selected view option as a boolean value which will be used in the view class (`CommandLine`) to decide what type of result which needs to be displayed.

So in this scenario, once the analysis is successful, the instance of the user selected level based `AnalysisResultLvl` class is passed as the initial analysis result to the `GUI` class which will be triggered for launching the application in the GUI format. Once the GUI application is launched and the initial result is loaded, then by default the graphical result will be displayed (Refer Figure 9).

Message Usually, the instance of the `AnalysisResultLvl` class contains the list of messages which will be of different types such as *ERROR*, *INFO*, *SUGGESTION* and *WARNING*. So all these messages will be printed in the console output of the CLI by going through all the messages, once the given application was analyzed successfully. The Figure 13 represents this type of analysis result in the CLI.

Filtering the analyzed result Similar to GUI, filtering the result for viewing the same (Textual or Graphical) is possible through CLI. User can filter the result as much as they want using CLI and view the result. User can use this filtering option in two ways,

- Filter and view the result with the help of the initial user input parameters available for result filters and detail levels. Even before analysis starts, the filter configuration is set

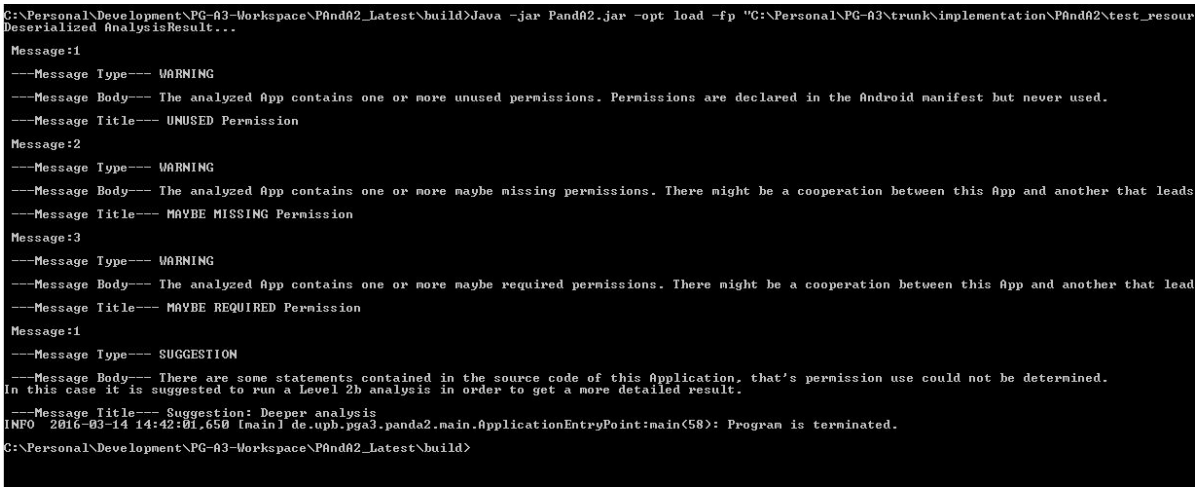

Figure 13: CLI Message Result

and once we get the analysis result, the result will be shown to the user in the console based on the initial filter configuration.

- Filtering the result once it is shown. This can be done by not terminating the program by giving the related parameter in the initial user parameter configuration. In general, result will be shown to the user and application will be terminated. In this case, by the result will be shown based on default detail level and result filter and then asks the user the option to filter the result.

Saving the analyzed result in the user selected file path Similar to GUI, the analysis result can be saved in CLI with the help `CustomSerializer` class which was explained in this section. The scenario is explained as follows,

- If the given filepath location exists, and if we get the successful analysis result, then result will be saved. Else error message will be shown to the user.

Loading the previous analysis result Once the result is saved, it can be loaded for viewing the same to CLI similar to GUI. All the options related to filtering the result is available for this functionality. For further information about how the result is getting loaded, please refer to Section 3.1.3).

Comparing the previous analysis result Based on the initial parameter configuration settings, comparing the application with previous analysis result or with the another Android application can be performed similar to GUI.

Initially, the information will be shown to the user in the console, regarding the difference between the applications as well as their version difference. If user wants to continue doing the analysis, then he has to enter "Y" in the console.

Once the analysis is performed successfully, then the result will be displayed or saved based on the initial user selected configuration.

3.1.3 Loading and Storing the Result

As mentioned in the Architecture document, we have implemented the functionality for loading and saving the instance of the `AnalysisResult` class using the serialization mechanism with the help of third party serializer library called *Kryo* [3]. Even though, Java supports serialization out of the box with the classes (`Serializable`, `Externalizable`), we have used the custom serializer library named *Kryo* for implementing this serialization mechanism for saving (serialization) and loading (de-serialization) the `AnalysisResult` object. Even though the implementation of default Java serialization was written in our project, currently we are using the *Kryo* serialization for performing this functionality.

The decision for using the custom serialization library over the standard Java serialization is because of following reasons,

- No need to implement `Serializable` interface in the classes if we use Kryo for doing the serialization.
- No need to update the *serialVersionUID* value if we modify the variables in the respective classes which we are serializing. The value of this varaible needs to be updated, whenever we make changes in the serialized classes. This must be required if we use `Serializable` interface for serialization.
- Custom options are available in *Kryo* for serializing the objects in the classes in other libraries such as *Soot*. Using Java serialization, it is not pretty staright forward to implement this feature.
- Custom options are available in *Kryo* for speeding up the de-serialization process by registering the respective classes which are getting serialized. This can be done in future if we want to improve the performance of reading and writing the object using *Kryo*.

In general, the way we have implemented the serialization mechanism in our project is by using the available serialization and dependent classes offered by the *Kryo* library. Serialization and de-serialization will be fully taken care by *Kryo*. It depends on the way we want certain classes to get serialized and de-serialized by providing specific parameters for them.

For storing the result, the user selected folder path and the file name will be used along with the instance of the generated instance of the `AnalysisResult` class. *Kryo* will serialize this object in the bytecode format and the contents will be saved in the respective file with the

[3] `https://github.com/EsotericSoftware/kryo`

file extension (*.pa2*). The file extension (*.pa2*) is important in certain cases such as during comparison mode as tool might validate the file extension of the previously saved analysis result. From *Kryo* or de-serialization perspective, this file extension is not a pre-requisite.

For loading the result based on the user selected file of *.pa2* format, the result will be de-serialized using the Kryo and will be converted to the object. Incase of any problems occured during the serialization or de-serialization scenarios, related exceptions will be captured and the information will be displayed in the tool.

This custom serializer (*Kryo*) dependent on the libraries [4] such as,

- minlog-1.3.0.jar.
- objenesis-2.2.jar.
- reflectasm-1.0.1-shaded.jar.

In the architecture document, we haven't mentioned in-depth about which serialization mechanism we are going to use for loading and storing the analysis result. So with respect to changes and additions for the serialization mechanism, the classes (`CustomSerializer` and `DefaultJavaSerializer` is added for implementing the functionality of the same.

`CustomSerializer` class represents the functionality which we have discussed earlier in this section (Serialization mechanism using *Kryo* library). Currently we are using this class for loading and storing the result. `DefaultJavaSerializer` represents the serialization functionality using Java serialization. We are currently not using the functionality of this class as we have mentioned the reasons earlier in this section.

3.2 CoreServices

The `CoreServices` describe elements that provide their functionality to be used in the analyses. Currently we provide the three services `XMLParser`, `DataStorage` and `StatementAnalyzer`. Details about these services can be found in the following sections, starting with the `XMLParser` in Section 3.2.1. This is followed by the `DataStorage` (Section 3.2.2). Finally, the `StatementAnalyzer` is described in Section 3.2.3.

3.2.1 XMLParser

XML Parser is one of the core features of our project and we are going to see a little bit in detail about it in this section. Our analysis tool gets the apks as the input, these apks are unzipped to get the XML file in the apk and we use XML Parser in our project to get the information from the XML file. The XMLParser we have implemented performs various functions which are described in detail below,

[4] `https://github.com/EsotericSoftware/kryo/tree/master/lib`

Cert Parser Cert Parser is a part of our XMLParser, which performs the action of getting the signature and fingerprint of the apk that is passed as the input. Cert Parser is an extension of Binary Parser which uses a xmlPull parser to understand the non-human-readable XML file, which we get for unzipping the apks.

XMLLayout Parser The function of XMLLayout Parser is to parse through the XML layout files of the Android application to find out the call back function that is used. For this first the unzipped layout file is parsed by LayoutBinary Parser using the XmlHandler object. Then `DummyMain` class uses XMLParser to find the call back methods and adds it to a list. For using LayoutBinary Parser, we have included axml2jar file to our project folder.

ARSC parser The resource files are represented with Integer ID While generating the Jimple code, which we get from passing the apk through Soot that are passed as the input. So the ARSC Parser job is to find out which resource files ID particularly layout ID, that is used by a specific class. To get the resourceID from Android application, the Integer ID from the Jimple code is passed to the ARSC Parser. ARSC Parser uses the Integer ID and goes through the Android application to get the name of the resource(layout) that has been used. `DummyMain` class uses ARSC parser to find layout files for Android application's activity class

ManifestXML Parser Similar to the Cert Parser, ManifestXML Parser is also an extension of Binary Parser that implements XmlPull Parser to get a human readable XML file. Then the ManifestXML parser goes through the Manifest file to get the information that is defined in it. Informations that are got from Manifest file is listed below

- Manifest Information
- App Name
- Uses Permission
- Intent Filter

3.2.2 The Interface DataStorage

Since the tool PAndA^2 tries to analyze a particular type of applications - Android application, it requires some defined input information from a specific version of Android Library to perform analysis. Such input information for example is a list of available permissions associated with API calls in the specific version. Hence in the PAndA^2 tool there is an interface that supplies those information, called `DataStorage`.

Being a part in the `Core services` of the PAndA^2 , the interface plays a role as a loader that loads and collects required data from files and stores them for later use. With the data, the interface provides a mapping of all APIs and their permissions, lists of classified library method calls such as sources, sinks or callback methods. Those APIs, permissions and method calls are supported in the Android Library API level 22 as agreed in the target level agreement. The class that implements the interface is `A3DataStorage`. It processes raw text in files

to obtain significant data needed for analysis (see Table 2 and 3.2.2 for description of files). Currently those files are results obtained from other tools. They are PScout [6] and SuSi [12]. The PScout extracts permissions supported in an Android version by using static analysis. The SuSi is an automated machine-learning tool that analyzes directly an Android Library source code to identify sources of sensitive data and sinks of maybe data-leaked methods. For the extensibility of the PAndA^2 tool to support newer versions of Android API, those files can be replaced by another ones which of course are generated by PScout and SuSi for the newer versions.

The interface is mainly used in the `Enhancer` component which builds data models for analysis within the PAndA^2 tool. In particular, the class `A3DataStorage` is used by the `Enhancer` to link all information obtained after Soot Framework disassembles an input `.apk` file. Such the information is permission, normal Java or Android component class, method and statement. In addition the interface is also used in some other specific classes for the Intra-App Information Flow Analysis.

3.2.3 Statement Analyzer

In Android applications, intent plays a very important role for launching components (such as activities, services, broadcast receivers and content providers) as well as for communication between them. In addition intents can also be used to transfer data from one component to another one. That is the reason why the PAndA^2 tool takes intents into account for the Intra-App Permission Usage Analysis, Intra-App Information Flow Analysis and the Inter-App Permission Usage Analysis. The PAndA^2 processes intents by the interface `StatementAnalyzer`.

The interface is considered as a service in the PAndA^2 tool and is implemented by the class `A3StatementAnalyzer`. It provides APIs which allow users to get intents in all types or in specific types available in Android applications. Currently there are two types of intent considered in the PAndA^2 tool. They are explicit and implicit intent. In the PAndA^2 tool, they are considered as valid if they are used to launch other components. By the way the PAndA^2 tool also considers a case of unknown-type intent which is mainly used for transferring data from a target component back to the caller one without launching any component.

The PAndA^2 tool takes body of each method in each class to analyze and collect intents if any available. All information of an input Android application is manipulated in the object `EnhancedInput` (see the Section 3.3.1 for more information). It means that while creating the object `EnhancedInput`, the interface `StatementAnalyzer` is also used to get all explicit and implicit intents existing in the application. Then they are stored in the object `EnhancedInput` for later use. Indeed, intents can be analyzed and obtained whenever an analysis needs them. However because all of analyses in the PAndA^2 tool require them, intents can be processed at the same time when the information of the application is manipulated. As a result of this, the performance of the PAndA^2 tool can be improved much. The relevant information of an intent includes it's type (explicit or implicit), the target component corresponding

PScout's Result Files	
File Name	Description

`allmappings`	This file is a text file containing all mapping between statements and their required permissions. The format of content in this file is:
	`Permission:<name of permission>`
	`<number of> Callers:`
	`<ClassName: ReturnType Method-Name()>`
	`...`
	`<ClassName: ReturnType Method-Name()>`

`publishedapimapping`	An extra file that supplies more permissions and statements besides the `allmappings` file. This file has same format as the `allmappings` file.

`contentproviderfield-permission`	This file lists out the classes or sub classes in the package `android.provider` with specific actions and the permission for those actions. The content of this file is in format:
	`PERMISSION:<name of permission>`
	`ClassName or SubClassName: android.net.Uri <Constant Variable>`

`contentprovider-permission`	Each content provider need a specific Uri to be accessed and each Uri is in format `content://uri` . Therefore this files maps each Uri with a permission that requires for Android application to access to the content provider.

`intentpermission`	Since implicit intent is specified by an action defined by the Android system, the `intentpermission` file will shows what action uses what permission. The detail of content looks like
	`<Constants Action Name> <Permission Name>`
	`<Abstract Class Name> <Permission Name>`

`listofallpermissions`	The file contains all supported permission in a specific version of Android Library. In details, this files contains 124 defined permissions in API 22 of Android System.
`androidcallbacks`	consists of all callback interfaces supported in the Android API version 22.

Table 2: Defined Input Information from PScout

SuSi's Result Files	
File Name	Description
output_Sources.txt	generated by the tool SuSi, this file contains all APIs that are seen as sources. Each line in the file represents for a source which is in format: <class: returnedType methodName(parameters,...)> -> _SOURCE_
output_Sinks.txt	same as the file for sources, but this file consists of APIs which are considered as sinks. Those APIs are represented in format: <class: returnedType methodName(parameters,...)> -> _SINK_

Table 3: Defined Input Information from SuSi

to the specified class or intended action string and finally the statement which launches the intent. For example, in the application SimToSms (14), there are 3 three intents. In the class SimReaderActivity, one is explicit intent with target class SmsSenderActivity and it is launched by the method call startActivityForResult. The second one is implicit and is specified with an action string de.upb.pga3.sendData. This intent is started by a method call startActivity. The last one is unknown type in the class SmsSenderActivity because it is used to transfer data from itself - the target component back to the caller - the SimReaderActivity.

As promised in the target level agreement, the PAndA^2 tool only considers intents which are declared, specified and launched in a same method. It skips intents which are defined as global variables or referenced variables. One important remark is that all syntaxes of classes, methods, statements and so on of an input Android application are not as same as normal Java language. The PAndA^2 tool therefore mainly work on statements in format of Jimple Code. For example the method onCreate(...) of the class SimReaderActivity (see 14) will be disassembled into Jimple Code as below

```
protected void onCreate(android.os.Bundle)
{
    de.upb.pga3.simtosms.SimReaderActivity $r0;
    android.os.Bundle $r1;
    android.content.Intent $r2;
    java.lang.Object $r3;
    android.telephony.TelephonyManager $r4;
    java.lang.String $r5;

    $r0 := @this: de.upb.pga3.simtosms.SimReaderActivity;

    $r1 := @parameter0: android.os.Bundle;
```

```
    $r3 = virtualinvoke $r0.<de.upb.pga3.simtosms.SimReaderActivity:
       java.lang.Object getSystemService(java.lang.String)>("phone");

    $r4 = (android.telephony.TelephonyManager) $r3;

    $r5 = virtualinvoke $r4.<android.telephony.TelephonyManager:
       java.lang.String getSimSerialNumber()>();

    $r2 = new android.content.Intent;

    specialinvoke $r2.<android.content.Intent: void
       <init>(android.content.Context, java.lang.Class)>
       ($r0, class "de/upb/pga3/simtosms/SmsSenderActivity");

    virtualinvoke $r2.<android.content.Intent: android.content.Intent
       putExtra(java.lang.String,java.lang.String)>("SIM_DATA", $r5);

    virtualinvoke $r0.<de.upb.pga3.simtosms.SimReaderActivity: void
       startActivityForResult(android.content.Intent,int)>($r2, 1);

    return;
}
```

Listing 1: Jimple code of method onCreate()

To extract intents, the analysis deals with three specific statements in format of `Jimple Code`. They are the declaring intent statements, the specifying property statements and the launching intent statements.

The first one - declaring intent statement - is an assignment. It declares a variable in type of `android.content.Intent`. The assignment can propagate to the variable a new instance of `android.content.Intent`, an other existing variable or a returned value from a method call. For example in the `Jimple Code` snippet above, the statement `$r2 = new android.content.Intent;` is considered as a declared intent statement. Because it defines the variable `$r2` in type of `android.content.Intent` which can be used later on. By dealing with this type of statement, the intent analyzer can get a lists of declared variables which perhaps then are enriched with further properties and are started.

The second type is the specifying property statements which always follow the declaring ones. This type of statement adds more information to already defined intents. An intent has many properties however to decide if it is explicit or implicit, the PAndA^2 tool will take into account the target classes or the action strings of it. Both of them are specified through method calls of the object `android.content.Intent`. There are two groups for those method calls. One is constructor and the another one is `set` methods. For example, after being declared, the intent `$r2` is initialized by a constructor. The constructor of it (`specialinvoke $r2.<android.content.Intent: void <init>(android.content.Con - text,java.lang.Class)>($r0, class "de/upb/pga3/simtosms/SmsSen-`

derActivity");) has two parameters. The important parameter is the second one which specifies the target class SmsSenderActivity. Based on it, the intent $r2 is decided as explicit one. In case a constructor does not have input parameters then the intent is considered as unknown. If an intent calls set methods such as setAction(...) or setClass(...) etc. type of it is also recognized. The PAndA^2 tool will covers several cases for constructors and set methods listed in the 3.2.3. Processing this type of statement helps the analysis to decide the type of an intent - explicit or implicit.

The last type of statements is used to start component specified in intents. After being declared and specified with properties, intent can be started. Then depending on target class or action string, the components corresponding to such those properties will run. For example, in the Jimple Code (1), the statement virtualinvoke $r0.<de.upb.pga3.simtosms. - SimReaderActivity: void startActivityForResult(android.con - tent.Intent,int)>($r2, 1); is a launching intent statement. It requires the intent $r2 defined above as input parameter. Currently the PAndA^2 tool processes many method calls for launching intents supported in the Android APIs 22. Table 3.2.3 lists out all of them. With this type of statement, the analysis can decide what intent is launched and what is not.

Briefly the analysis traverses forward a control flow graph which is built for body of each method. The graph has type of UnitGraph - an object supported by Soot Framework. Indeed, the analysis can processes forward through a set of statements of a body. However the set does not represent exactly the order of statements as well as the control flow through each of them. That is why the UnitGraph is more promised for the result of analysis intents. While traversing forward the graph, the analysis records all defined variables and their referenced objects or values in a list not only for intents but also for another types such as string, int or object class. Because the variables may be used later on as an instance of a class, as a property or as input parameters for method calls. In case a statement is a method call from a defined variable, the analysis can get the referenced object according to the defined variable. If a statement is a method call which requires some variables as input parameters, then basing on the list of defined variable, the analysis can extract the values corresponding to the required variables. By this way, the analysis can obtain exactly information needed in each statement to decide the property of intents. For example in the snippet Jimple Code above, the statement virtualinvoke $r2.<android.content.Intent: android.content.Intent putExtra - (java.lang.String,java.lang.String)>("SIM_DATA", $r5); requires the variable $r2 as a referenced object and the variables $r5 as input parameters. Then the analysis will check in the list of defined variables to get the Intent referenced by the variable $r2 as well as the value of the parameter $r5. Hence, it is known that the already defined intent $r2 is manipulating the data hold by $r5. For intents, if a statement is the specifying property statement (the second type), the analysis can extract the values of the input parameters in the statement, and then use the EnhancedInput to get the target class corresponding to the values. If a statement is in type of launching intent statement, the analysis will get the referenced object according to the input parameter of the statement to know what intent is launched.

Methods	Description
`Intent()`	constructor of intent without any specification (unknown type)
`Intent(java.lang.String)`	initialize an intent with an action (implicit)
`Intent(java.lang.String, android.net.Uri)`	construct an intent with an action and a URI (implicit)
`Intent(android.content.Intent)`	construct an intent with an existing one. Type of the form on is depending on the type of the later one.
`Intent(android.content.Context, java.lang.Class)`	initialize intent with a specific class or component (explicit)
`Intent(java.lang.String, android.net.Uri, android.content.Context, java.lang.Class)`	constructor of intent which is specified with an action associated with a URI as well as a class or a component. (can be both types)
`setAction(java.lang.String)`	set an action to existing intent (implicit)
`setClass(android.content-.Context, java.lang.Class)`	specify a class or component within the App to intent (explicit)
`setClassName(android.content-.Context, java.lang.String)`	specify a string class name to an intent (explicit)
`setClassName(java.lang.String, java.lang.String)`	set a package name as well as the class name or component name to an intent (explicit)

Table 4: Specifying Intent Methods

Process control flow graph by using the `UnitGraph` of `Soot Framework`, the intent analysis has a control flow graph which helps processing statements in body of a method more precisely. However the analysis also must deal with some problems of if statement and loop statement. Because `Soot Framework` just supports a graph and it is not a compiler to show exactly the result of condition in if statement or in loop statement, the analysis will assume that all flows or branches generated by these statement will be taken into account. Hence there would be a case where there is only one defined intents, but through if statement (or loop statement) the intent is specified in two different ways corresponding to true and false of condition in the if statement (or loop statement). Then the analysis returns two intents. There would be a case when an intent is already defined and specified, but through the control flow, the intent is defined or specified again, then the analysis will consider only the latest definition or specification of the intent.

No.	Methods
1	`void startActivity(android.content.Intent)`
2	`void startActivity(android.content.Intent, android.os.Bundle)`
3	`void startActivityForResult(android.content.Intent,int)`
4	`void startActivities(android.content.Intent[])`
5	`void startActivities(android.content.Intent[], android.os.Bundle)`
6	`void startActivityForResult(android.content.Intent,int, android.os.Bundle)`
7	`void startActivityFromChild(android.app.Activity, android.content.Intent,int)`
8	`void startActivityFromChild(android.app.Activity, android.content.Intent,int,android.os.Bundle)`
9	`void startActivityFromFragment(android.app.Fragment, android.content.Intent,int)`
10	`void startActivityFromFragment(android.app.Fragment, android.content.Intent,int,android.os.Bundle)`
11	`boolean startActivityIfNeeded(android.content.Intent, int)`
12	`boolean startActivityIfNeeded(android.content.Intent, int,android.os.Bundle)`
13	`boolean startNextMatchingActivity(android.content-.Intent)`
14	`boolean startNextMatchingActivity(android.content-.Intent,android.os.Bundle)`
15	`android.content.ComponentName startService( android.content.Intent)`
16	`boolean bindService(android.content.Intent, android.content.ServiceConnection,int)`
17	`void sendBroadcast(android.content.Intent, java.lang.String)`
18	`void sendBroadcast(android.content.Intent)`
19	`void sendOrderedBroadcast(android.content.Intent, java.lang.String)`
20	`void sendOrderedBroadcast(android.content.Intent, java.lang.String,android.content.BroadcastReceiver, android.os.Handler,int,java.lang.String, android.os.Bundle)`
21	`void sendStickyBroadcast(android.content.Intent)`
22	`void sendStickyOrderedBroadcast(android.content.Intent, android.content.BroadcastReceiver,android.os.Handler, int,java.lang.String,android.os.Bundle)`

Table 5: Launching Intent Methods

3.3 Enhancer

The Enhancer is the first step in all three analysis levels our PAndA^2 tool implements.

It decompiles the given Android application and generates an `EnhancedInput` for this application (see Section 3.3.1). For this task the Enhancer uses the Soot framework which performs the decompilation of the application. The result given by Soot then has to be wrapped by the Enhancer into our structure of the `EnhancedInput`.

Furthermore, the `Enhancer` enriches the datastructure given by Soot. This is done by means of the `PermissionMapper` as well as the Core Service `StatementAnalyzer` (see Section 3.2.3 for more details).

The `PermissionMapper` adds permissions to statements as well as to Android components and to the whole application. This is done by investigating the statement and dealing with the different cases of statements. Possible cases are:

- content provider
- API call
- implicit intent

Depending on the case which was found the datastorage (Section 3.2.2) is called to map the permission to the statement.

3.3.1 EnhancedInput

The `EnhancedInput` is a tree like datastructure which describes the source code of the application to be analysed. This structure consists of a set of nodes and links between them. A node can represent one of the following parts of the application:

- App
- Class (Android component)
- Method
- Statement
- Permission

For an application there exists one App node which is the root of the `EnhancedInput`. This node then has links to all its classes. An Android component is a special type of a class. Each class is connected to its methods, the methods have links to their statements and finally each statement which is protected by a permission has a connection to a corresponding permission.

In the Architecture Document we planned to use a datastructure that represents the source code of an Android application with own classes for e.g. the method, statement, etc. nodes

but while we started developing, acquired deeper knowledge about Soot and elaborated our requirements for the analyses, we decided to use the datastructure which is already provided by Soot instead of creating a completely independent one.

The main reason for this design decision was that the way we planned our datastructure in the architecture phase would lead to some problems in the analyses, e.g it did not take the Soot datastructure into account. This would lead to problems in the Intra-App Information Flow Analysis (Section 3.6), where we would have to map the datastructure given by Soot to our datastructure. Furthermore, the statements in our own datastructure were only given in a string representation, so any additional information, like for example the fact whether it is an invocation statement and which method it invokes would be lost and it would be an overhead to regain the information. In addition to that, the Soot datastructure directly provides the usage links of variables, which was another advantage of that datastructure. So we decided to use the classes `SootClass, SootMethod` and `Unit` replacing our own datastructure for classes, methods and statements.

But there is also a disadvantage that came along with this decision. The Soot datastructure does not support the possibility to navigate from a unit to the method it belongs to. Therefore we had to enrich our `EnhancedInput` to add this possibility since we wanted to be able to navigate through the complete `EnhancedInput`without problems. By adding a map from each unit to its method body we solved this problem. The map is created while the `EnhancedInput` is created and can then be used in the following analyses.

Furthermore we wanted to connect statements with their corresponding permissions but since the Soot datastructure has no such connections we had to add another mapping to the `EnhancedInput` which then gives us the possibility to store the links between statements and permissions.

We developed the SimToSms application as a running example. It will occur wherever we want to explain some contents on an example application. The application consists of the two classes `SimReaderActivity` and `SmsSenderActivity`, which behave as the names already suggest. The `simReaderActivity` reads the sim serial number and the `SmsSenderActivity` sends it via sms. The source code of this application can be found in the Appendix in Listing 14. Figure 14 shows a part of the `EnhancedInput` of the SimToSms application. The root node is the App node, which then is linked to the Android components `SimReaderActivity` and `SmsReaderActivity`. The activities are connected to their methods and the methods to their statements. Due to space reasons not all methods and statements are shown in the figure.

Each statement that is protected by one or more permissions has a link to the corresponding permissions, like for example node 10 which represents the statement `sendTextMessage(...)`. This statement is protected by the two permissions *android.permission.SEND_SMS* and *android.permission.WRITE_SMS* and is therefore linked in the `EnhancedInput` to both of these permissions.

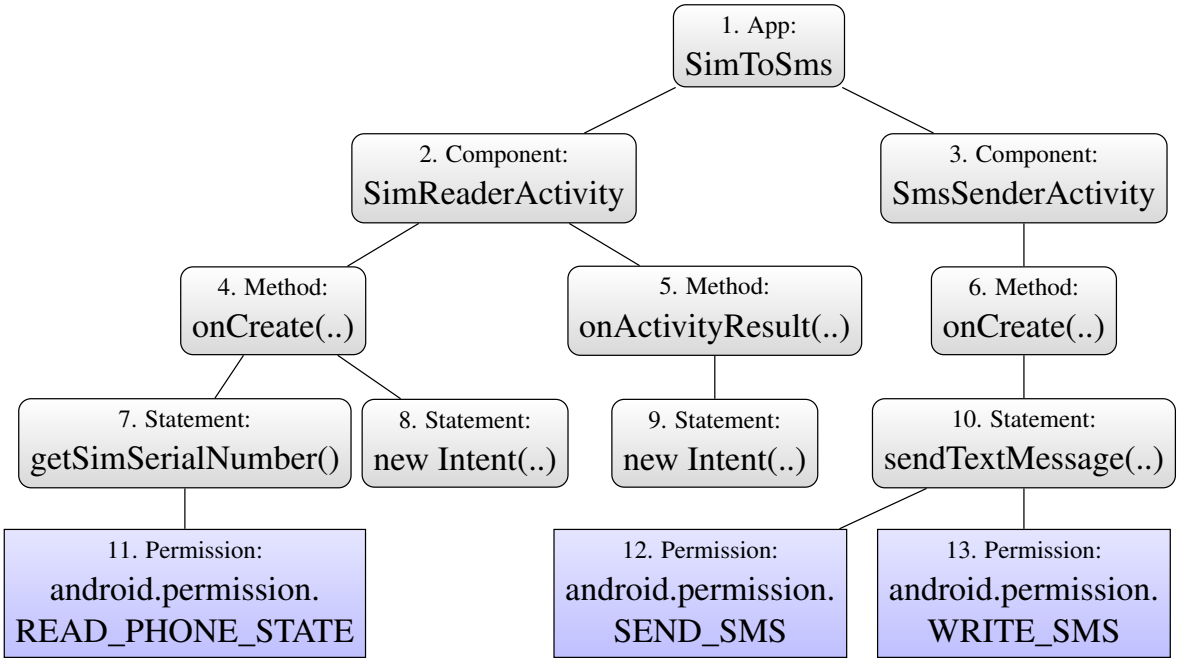

Figure 14: `EnhancedInput` of the SimToSms App

3.4 Intra-App Permission Usage Analysis

The first type of analysis which comes with our tool is the Intra-App Permission Usage Analysis. This analysis uses the capabilities of our tool in order to check if an App is or might be using any permissions. If it uses permissions this analysis will tell the user if the App is allowed to use these permissions and if the App is trustworthy or not. Some violations appear to influence the trustworthiness, but in reality they could only do so with the help of another App. In this case the analysis will determine if such a cooperation between the analyzed App and another App is possible.

The analysis itself works as follows. At first all permissions used in the App are collected. Then each and every collected permission will be categorized into one of the following groups:

- REQUIRED: All permissions which are required are needed in order to use the application and their use is intended. The user will be informed that these permissions are used.
- UNUSED: While installing an application the user will always be informed which permission are used by the App. But it might be that some of these permissions are not used at all. These will be marked as unused.
- MISSING: Missing permissions are the most security critical ones, because this group describes all permissions which are used by the App without informing the user.

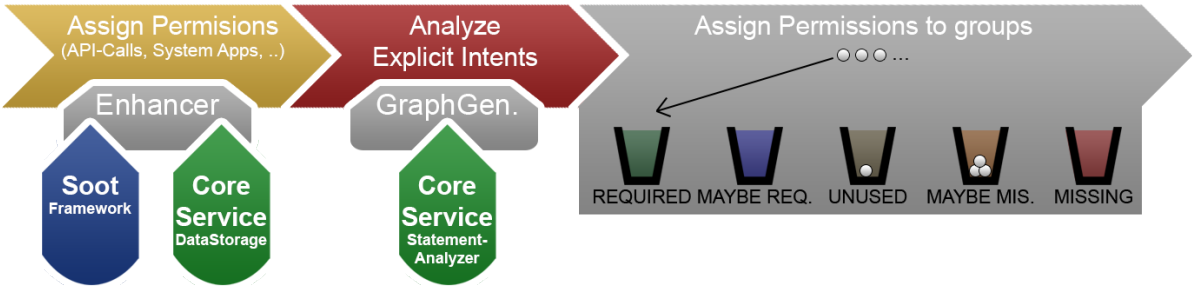

Figure 15: Intra-App Permission Usage Analysis: Overview

And there are two more groups considering the case that the App itself does not require (MAYBE_REQUIRED) or miss (MAYBE_MISSING) a permission, but an cooperating App might use these permissions through the analyzed App. In the following these five groups will be called **Permission-Groups (PGs)**.

After the analysis has finished the result provides the gathered information on METHOD, CLASS, COMPONENT and APP detail level. This means the PGs will be assigned to all permission involvements in each and every method, class (component) and the App itself. This gives the user of our tool the possibility to tell which permissions e.g. are missing. In addition it allows to investigate further where exactly these permissions are missing. In other words in which class, component or method the permissions are missing.

In the next sections it will be described how the Intra-App Permission Usage Analysis works. Figure 15 provides an overview over its three different steps. The SimToSms example, introduced in Section 3.3, is continued and will be used to explain the different steps.

3.4.1 Enhancer Support

As well as in any other analysis the first step is the execution of the `Enhancer` (see Section 3.3). By that it will provide us an `EnhancedInput` object which is the basis for this analysis. This basis includes a tree like data structure that represents the whole App. In the following $G_{tree} = (V, E)$ will describe this data structure. The nodes (V) of such a tree stand for the App's classes (components), methods and statements and the information which statement requires which set of permissions. E is the set of all edges which connect the nodes and build the tree. In more detail the edges are connecting the App node with all class (component) nodes, the class nodes with their associated method nodes and the method nodes in turn with their associated statement nodes.
Figure 16 illustrates G_{tree} for the SimToSms App. It only shows a subgraph of the complete graph, but it is sufficient for all following explanations. All nodes are consecutively numbered

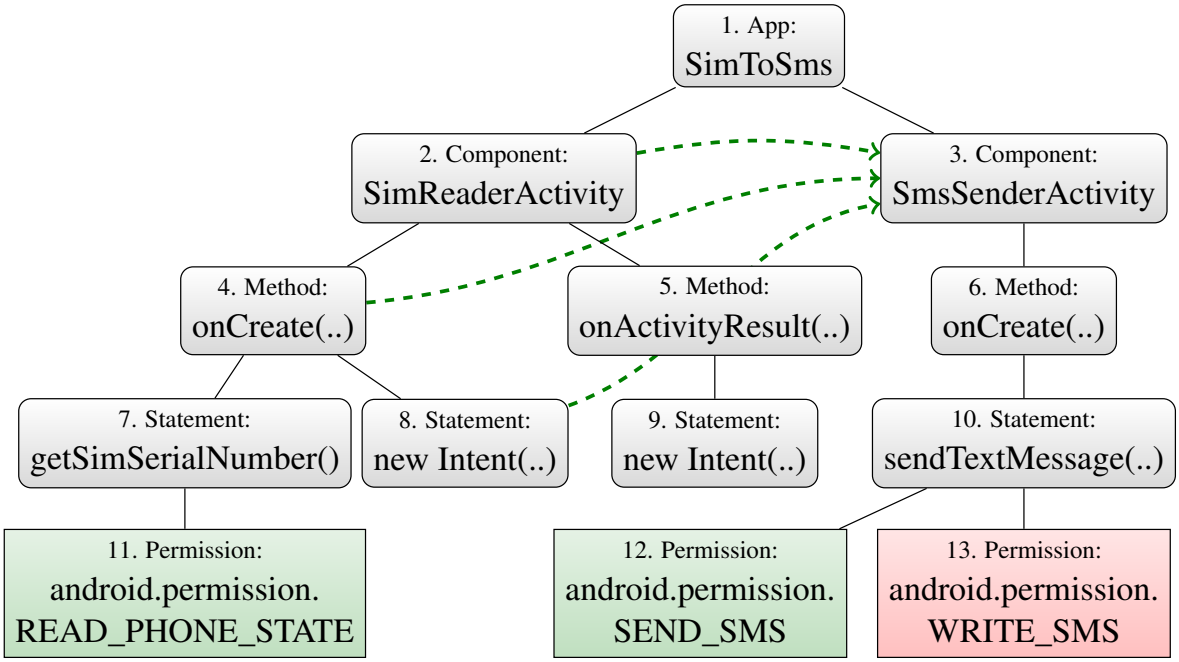

Figure 16: Analysis basis for the SimToSms App

and classified by the first line of any node's label. Additionally the labels reference the part of the code or the permission which is represented by this node. Let L be the set of all labels and let $v_l \in V$ stand for the node with label $l \in L$. For example the node $v_{4.\text{ Method: onCreate(..)}}$ has number 4 and is classified as a Method node which represents the method `onCreate(..)` of the component `SimReaderActivity` (see Line 6-17 in Listings 14).

3.4.2 GraphGenerator: Analyze Explicit Intents

Creating a graph with the analysis specific `GraphGenerator` is the second step. The graph will represent the inter-component communication of the analyzed App. The `GraphGenerator` will use another `CoreService`, namely the `StatementAnalyzer` (see Section 3.2.3) in oder to find out if a statement is an explicit Intent definition. In that case the `StatementAnalyzer` will provide a list of targets. Targets are other Android components which could be identified as receivers of the defined Intent. The target list will be used for the graph generation. A transition will be added from the Intent statement and its belonging method and class to all the targets.
Let T be the set of all transitions. Then the analysis basis will be extended by these transitions and by that build the graph which is used in the next step. G_{tree} will be transformed into the graph $G = (V, E \cup T)$. In the current example three edges will be added as visualized by the green, dashed arrows in Figure 16. The start nodes of these edges are

Table 6: Decision table (Intra-App Permission Usage Analysis)

Permission-Group assigned	Question 1.	2.	3.
REQUIRED	✓	✓	
MAYBE_REQUIRED	✓	✗	✓
UNUSED	✓	✗	✗
MISSING	✗	✓	
MAYBE_MISSING	✗	✗	✓

$v_{8.\text{ Statement: new Intent(..)}}$, $v_{4.\text{ Method: onCreate(..)}}$ and $v_{2.\text{ Component: SimReaderActivity}}$. The end node always is $v_{3.\text{ Component: SmsSenderActivity}}$.

3.4.3 Analyzer: Assign Permission-Groups

Based on the graph's information and by the analysis specific `Analyzer` one PG will be assigned to every permission at each node. Let P be the set of all available permissions in the Android API and $MaybeMore \subseteq V$ represent a set of nodes which have an implicit Intent as descendant and by that may call another cooperating App. At first any permissions assigned to a node $v \in V$ will be assigned to all ancestors of v except the root node. The root node represents the whole App in one node and only the permissions described as used in the Android manifest are assigned to it. In addition for all nodes $u \in MaybeMore$ all ancestors will be added to $MaybeMore$. By that in the SimToSms example the permission *android.permission.READ_PHONE_STATE* is assigned to the nodes $v_{4.\text{ Method: onCreate(..)}}$ and $v_{2.\text{ Component: SimReaderActivity}}$. As well as the permissions *android.permission.SEND_SMS* and *android.permission.WRITE_SMS* are assigned to the following nodes: $v_{6.\text{ Method: onCreate(..)}}$ and $v_{3.\text{ Component: SmsSenderActivity}}$. In addition the nodes $v_{5.\text{ Method: onActivityResult(..)}}$, $v_{2.\text{ Component: SimReaderActivity}}$ and $v_{1.\text{ App: SimToSms}}$ are added to the $MaybeMore$ set, because of the implicit Intent represented by node $v_{9.\text{ Statement: new Intent(..)}}$.

After transferring all permissions and filling up the $MaybeMore$ set the `Analyzer` will answer 3 questions for each node $v \in V$ and each permission $p \in P$:

1. Is permission p assigned to the visited node v?
2. Is permission p assigned to at least one child or descendant of v?
3. Is v in the $MaybeMore$ set?

Question 1 will receive a positive answer if permission p has been assigned to node v. Question 2 checks if p is really required by a statement which is a descendant of v. A positive answer to question 3 allows the possibility that p might be required or missing.

Based on all the answers the PG will be assigned to p at v. The Table 6 displays in which case which PG will be assigned. The ✓-Symbol symbolizes a positive answer and the ✗-Symbol

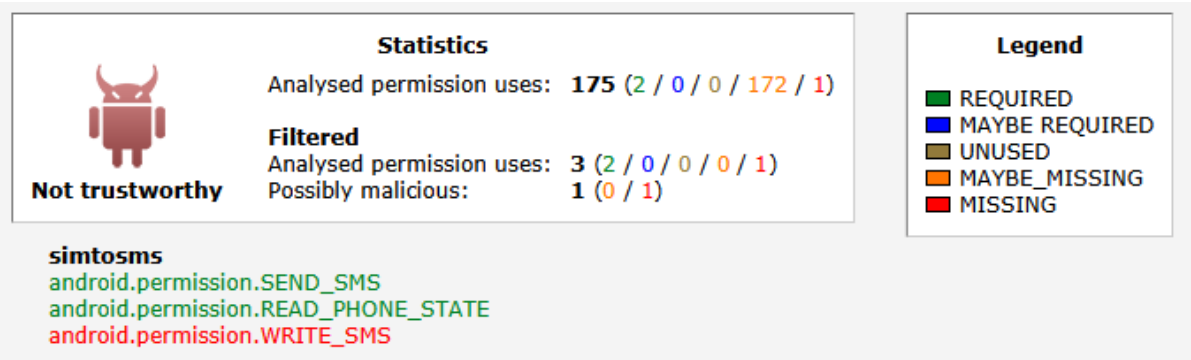

Figure 17: Intra-App Permission Usage Analysis: Textual result

a negative answer. If a cell is empty the answer to this question does not influence the outcome. For example if $v = v_{2.Component:SimReaderActivity}$ and $p = android.permission.READ\_PHONE\_STATE$ then the answers will be:

1. $\rightarrow$ ✓
2. $\rightarrow$ ✓
3. $\rightarrow$ ✓

Accordingly the PG that will be assigned to p at v is the REQUIRED PG. After assigning the PGs to all permissions at each node the analysis ends.

3.4.4 Result Representation

In the following, based on the analysis result of the SimToSms App, the result representation is explained. The source code of the SimToSms App contains two statements which are using permissions (see Line 10 and 53-54 in Listing 14). These statements are Android API calls. In the first statement the method `getSerialNumber()` from the class `android.telephony.TelephonyManager` is being called. The execution of this statement requires the *android.permission.READ_PHONE_STATE* permission. The second statement consists of the method call `sendTextMessage(...)` from class `android.telephony.SmsManager`. This call in turn requires two permissions *android.permission.SEND_SMS* and *android.permission.WRITE_SMS*.

The Figure 17 shows the textual result representation of the example generated and displayed by the PAndA^2 tool. A short analysis summary is given in the statistics-box. Since there is a missing permission the App is categorized as not trustworthy. Only Apps without any missing or maybe missing permissions are considered as trustworthy. In the main part below the statistics-box we can see that the permissions *android.permission.SEND_SMS* and *android.permission.READ_PHONE_STATE* are required and *android.permission.WRITE_SMS* is missing indicated by the colors which are defined in the Legend at the top right corner. The

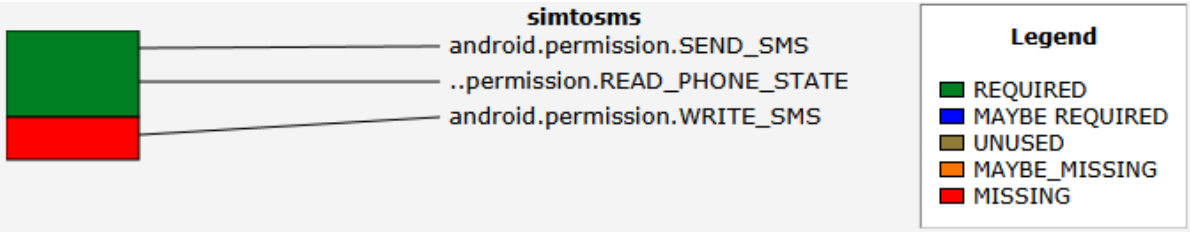

Figure 18: Intra-App Permission Usage Analysis: Graphical result

shown result is displayed on APP detail level and the filters are set up to show all PGs except the MAYBE_MISSING one. On the other hand Figure 18 shows the graphical result representation. It figuratively shows how trustworthy the App is: The ”greener” the bar on the left is the trustworthier the App is. This pictorially description considers the PGs MAYBE_REQUIRED and UNUSED as green and the MAYBE_MISSING and MISSING PG as red. It also represents which permission belongs to which PG by the edges between the bar and the permission labels. Not visible in any figure are the generated messages. In this example case there are three messages. The first one is a suggestion that tells us to execute a deeper analysis (see Section 3.5) in order to find out if the MAYBE_REQUIRED and MAYBE_MISSING permissions are really required/missing. The second one is a error message which informs the user especially that the analyzed App is missing a permission. The last message is a warning which is provided because there are maybe missing permissions that may cause a data leak. Based on all these information the user can reconsider the decision to use this App or investigate further where and for what these permissions are used.

The result itself will be saved in another tree like data structure. Figure 19 partly illustrates the analysis result of the SimToSms example. The root of such a tree is the result itself which always has 4 children, one for each detail level (see APP, COMPONENT, CLASS and METHOD nodes). Each child in turn contains one child per associated object e.g. the `SimReaderActivity`. All objects have 5 children representing the PGs. And these children finally hold the assigned permissions as leafs of the tree. This type of data structure makes filtering and changing the detail level quite easy and fast, because the only thing we have to do is selecting another subtree. On the other hand when it comes to memory efficiency another data structure would perform better.

All in all the Intra-App Permission Usage Analysis will detect the possibility of data leaks based on Android’s permission system and allows to start a detailed causal research based on the analysis result. In comparison to other state-of-the-art taint analyses which mostly focus on information flow analyses this analysis is less precise but in most cases a lot faster and often sufficient (see Section 6.1). The next section will describe an extension of this analysis, which makes it more precise and capable of taking more than one App as input.

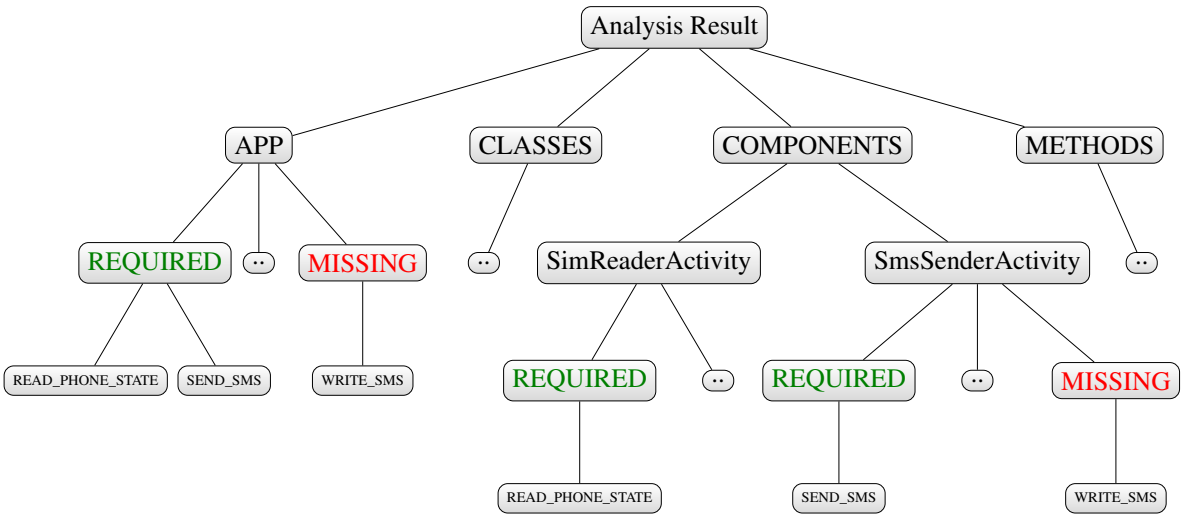

Figure 19: Analysis result for the SimToSms App (Intra-App Permission Usage)

3.5 Inter-App Permission Usage Analysis

This type of analysis is an extension to the previously described Intra-App Permission Usage Analysis (see Section 3.4). It uses the capabilities of our tool in order to check if an App is or might be using any permissions. In contrast to the Intra-App Permission Usage Analysis this analysis will identify cooperations between different Apps instead of only detecting the possibility of such a cooperation. By that it will include multiple Apps into the analysis instead of just one.

The analysis itself works almost exactly the same way the Intra-App Permission Usage Analysis works. Each permission that is detected as being used in the analyzed App will be categorized into one of the previously introduced Permission-Groups (PGs - see Section 3.4) and in addition a differentiation will be made between direct and indirect use of these permissions. A permission use is considered as **direct**, if there exists a statement in the source code of the analyzed App that directly requires a permission. On the other hand **indirect** permission uses are detected if one permission is accessed through another cooperating App. This fact leads to the higher precision of this analysis. Instead of just determining the existence of a cooperation itself this analysis will find the cooperating App and analyze exactly which permissions can be accessed. Once the result is computed two detail levels are supported, namely COMPONENT and APP.

The example presented before will be continued in this Chapter as well. By explaining the analysis result of this example the benefit of this analysis will come out. But first of all a second App that is cooperating with the SimToSms App is needed. This App is called **PhoneNumberToInternet** and will be introduced now. It does exactly what the name suggests: Uploading a phone number to the Internet (see Listing 2). In this case it is not any phone number but the number of the SMS receiver which is defined in the SimToSms App. The

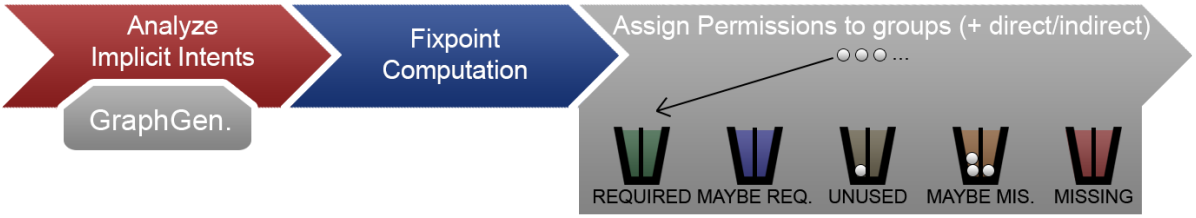

Figure 20: Inter-App Permission Usage Analysis: Overview

only permission required by this Application is the *android.permission.INTERNET* permission which is used by calling the `connect()` method of the `HttpURLConnection` class (see Line 12).

```
1  public class PhoneNumberUploaderActivity extends Activity {
2      public static final String SERVICE_NUMBER_DATA = "RECEIVER_NUMBER";
3
4      @Override
5      protected void onCreate(final Bundle savedInstanceState) {
6          String phoneNumber = getIntent().getStringExtra(SERVICE_NUMBER_DATA);
7
8          try {
9              URL url = new URL("http://website.net/upload.php?phonenumber="
10                                 + phoneNumber);
11             HttpURLConnection conn = (HttpURLConnection) url.openConnection();
12             conn.connect();
13         } catch (IOException e) {
14             Log.e("Error", e.getMessage());
15         }
16     }
17 }
```

Listing 2: Source code of PhoneNumberToInternet App

The next sections will show how this level of analysis is working in detail. Figure 20 figuratively summarizes the whole workflow of this analysis.

3.5.1 Enhancer Support: Collect Previous Results

As in every other analysis as a first step the `Enhancer` will be called. But since this analysis is an aggregation analysis the `Enhancer` will work differently this time (This is the only aggregation analysis that comes with our tool). The `AnalysisRunner` will run the Intra-App

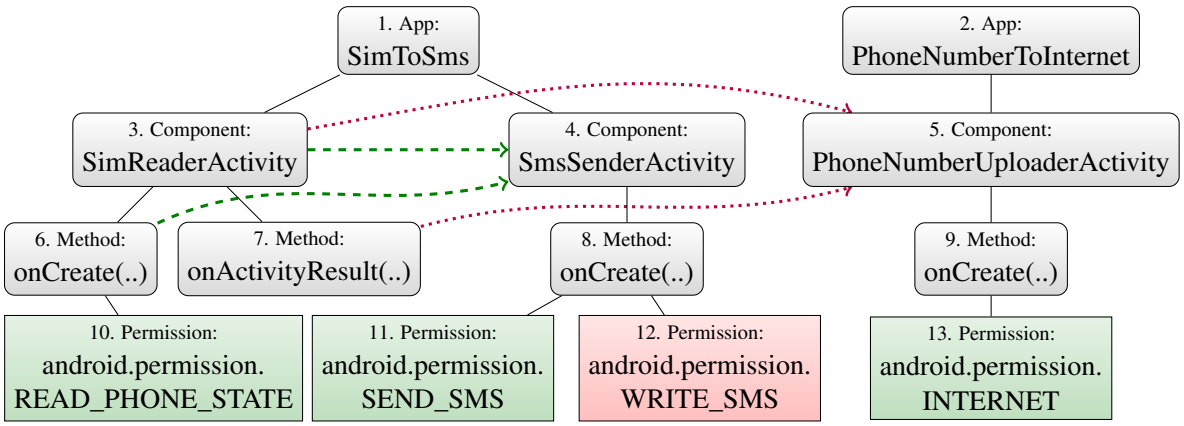

Figure 21: Analysis basis for the SimToSms and PhoneNumberToInternet App

Permission Usage Analysis on all input .apks, including the non-native .apks if the user has provided any. Once all of these analyses are finished the `Enhancer` will only collect and forward the results of these analyses. The collection of all these results represents the analysis basis in this case. Figure 21 shows a cutout of this analysis basis with respect to the current example. Basically it contains two trees representing the two different Apps (SimToSms and PhoneNumberToInternet). The cutout does not display any statement nodes because the result can only be displayed on APP or COMPONENT detail level and by that every detail of the example can be explained based on this cutout. Nevertheless the full basis also contains nodes for all statements in the Apps. The green, dashed arrows still represent the transitions based on the explicit Intents. Let $G_{SimToSms}$ describe the part of the analysis basis with $v_{1.App:SimToSms}$ as root node and $G_{PhoneNumberToInternet}$ the other part. Then the analysis basis $G = (V, E)$ can be defined as $G = G_{SimToSms} \cup G_{PhoneNumberToInternet}$. Currently G is a disconnected graph that consists of multiple connected subgraphs. These subgraphs might be connected with each other in the next step.

3.5.2 GraphGenerator: Analyze Implicit Intents

The analysis specific `GraphGenerator` will now add transitions for the implicit Intents. These are illustrated by the purple, dotted arrows in Figure 21. In order to compute these edges the `StatementAnalyzer` is called again. It will provide a list of implicit Intent definition statements and for each of these, a list of targets. Just as it did before for explicit Intents but this time the targets are not required to be part of the analyzed App. They can be part of another non-native App that has been provided as input. The exact way of finding targets is described together with the `StatementAnalyzer` (see Section 3.2.3). As in the Intra-App Permission Usage Analysis a transition will be added from the Intent statement and its belonging method and class to all the targets. So lets assume T' represents all these transitions. Then the analysis graph G' can be build by transforming the analysis basis G into $G' = (V, E \cup T')$.

3.5.3 Analyzer: Assign Permission-Groups

The first thing the `Analyzer` of this analysis does, is transferring permissions. As before any permissions assigned to a node $v \in V$ will be assigned to it's ancestors except any root node. In the current example this will also transfer permissions from $G_{PhoneNumberToInternet}$ to $G_{SimToSms}$ since both graphs are connected by the transitions in T'. For example the *android.permission.INTERNET* is represented by the node $v_{13.Permission:android.permission.INTERNET}$. All valid ancestors that can be found are: $v_{9.Method:\ onCreate(..)}$, $v_{5.Component:PhoneNumberUploader-Activity}$, $v_{7.Method:onActivityResult(..)}$ and $v_{3.Component:SimReaderActivity}$. The permission will be assigned to all these nodes. The same happens vice versa. So e.g. the permission *android.permission.READ_PHONE_STATE* will be transferred to $v_{5.Component:\ PhoneNumberUploaderActivity}$. During the analysis this transferring of permissions will be executed multiple times until a fixpoint is reached. This is necessary because there could be an App that is getting access to a permission through another App, which in turn got access to this permission through a third App. The fixpoint computation can get quiet complex if a lot of Apps are involved. Because of that we implemented a worklist algorithm. The worklist WL is initialized with the whole set of nodes $WL = V$. Once a node v is visited it will be removed from the worklist $WL = WL \setminus \{v\}$ and all permissions assigned to it are transferred. If any node $u \notin WL$ gets a new permission while transferring, it will be added to the worklist $WL = WL \cup \{u\}$. This process continues until the worklist is empty $WL = \emptyset$.
In addition while transferring permissions all indirect permission usages will be marked. So if one permission is transfered via a transition $t' \in T'$ this permission will get marked as indirect for the target node of t'. Accordingly let function $indirect : V \times P \rightarrow \{true, false\}$ describe a function that returns $true$ if the permission $p \in P$ at node $v \in V$ is accessed indirectly and $false$ in any other case.
As in the Intra-App Permission Usage Analysis *MaybeMore* will specify a set of nodes that have an implicit Intent as descendant, whose targets could not be found in the provided input .apks.

Then the `Analyzer` will start assigning PGs to every permission $p \in P$ at each node $v \in V_0$. V_0 correlates to V but it does not contain any node that stands for a statement or a method. It is sufficient to use V_0 for this computation since the available detail levels for this analysis are APP and COMPONENT. In order to assign the PGs four questions will be answered this time:

1. Is permission p assigned to the visited node v?
2. Is permission p assigned to at least one child or descendant (u) of v..
   2.1. and is $indirect(u, p) = true$?
   2.2. and is $indirect(u, p) = false$?
3. Is v in the $MaybeMore$ set?

Table 7 shows in which case which PG will be assigned. It also contains the differentiation between direct and indirect access. Back to the current example this means that e.g. for

Table 7: Decision table (Inter-App Permission Usage Analysis)

Permission-Group assigned	**Question 1.**	**2.1.**	**2.2.**	**3.**
REQUIRED (direct)	✓	✓	✗	
REQUIRED (indirect)	✓	✗	✓	
REQUIRED (direct & indirect)	✓	✓	✓	
MAYBE_REQUIRED	✓	✗	✗	✓
UNUSED	✓	✗	✗	✗
MISSING (direct)	✗	✓	✗	
MISSING (indirect)	✗	✗	✓	
MISSING (direct & indirect)	✗	✓	✓	
MAYBE_MISSING	✗	✗	✗	✓

$v = v_{3.Component:SimReaderActivity}$ and $p = android.permission.INTERNET$ the PG MISSING (indirect) will be assigned, because the questions are answered as follows:

1. → ✗
2.1. → ✗
2.2. → ✓
3. → ✗

After assigning the PGs to all permissions at each node the analysis ends.

3.5.4 Result Representation

The result of the SimToSms and PhoneNumberToInternet example is displayed in Figure 22 (textual) and Figure 23 (graphical). The differences between this result and the result of the Intra-App Permission Usage Analysis will be pointed out now. The most obvious and interesting difference is that another missing permission could be found. Namely the permission *android.permission.INTERNET* was previously assigned to the MAYBE_MISSING PG and is assigned to the MISSING PG now. The reason is that the previous analysis could only find out that another unknown App might be called but this analysis in turn could tell exactly that PhoneNumberToInternet is the App which might be called. And since the PhoneNumberToInternet App is using the *android.permission.INTERNET* permission, the SimToSms App could also get access to the Internet through this cooperating App. In fact, we know it does. The SimToSms App will send the phone number to the PhoneNumberToInternet App which will upload it. This security breach could be found and identified by this analysis. But there is another difference in these two results. The permission *android.permission.READ_PHONE_STATE* is still considered as required but it could be accessed direct and indirect now. This again is related to the fact, that the called App has been determined to be the PhoneNumberToInternet App. Both differences can be recognized in the graphical result as well. The legend has been

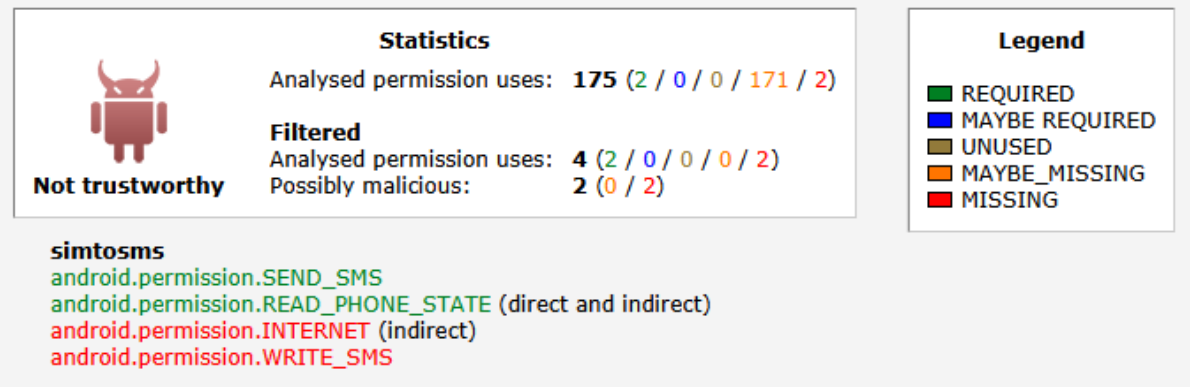

Figure 22: Inter-App Permission Usage Analysis: Textual result

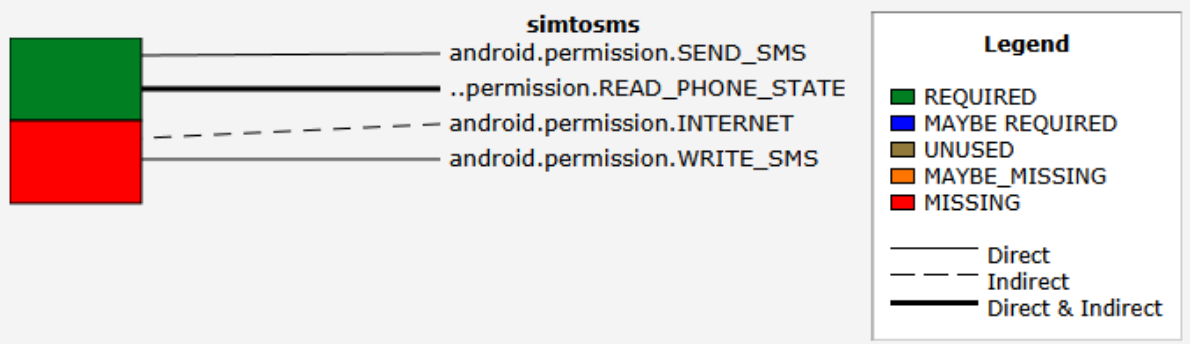

Figure 23: Inter-App Permission Usage Analysis: Graphical result

extended to show which permissions are indirectly accessed. With this more precise analysis result it might be easier for the user to decide whether he wants to use this App or not.

As before the result itself will be saved in another tree like data structure. This structure is very similar to the structure from the Intra-App Permission Usage Analysis. The only two differences are, that on level 1 of the tree due to the restricted detail levels only two nodes can be found and on the other hand that the direct/indirect information is saved along with the permissions. A subtree of this result structure associated with the running example is illustrated in Figure 24.

Finally it must be said that the PAndA^2 tool's framework made this extension very easy to implement, because it could be done by simply extending the Intra-App Permission Usage Analysis. And even if it generates an overhead by computing all the Intra-App Permission Usage Analysis results first it is faster then most of the state-of-the-art information flow based taint analyses (see Section 6.1). But of course it is less precise. However in many cases the result of the Inter-App Permission Usage Analysis is sufficient e.g. if you want to detect possible cooperations between Apps.

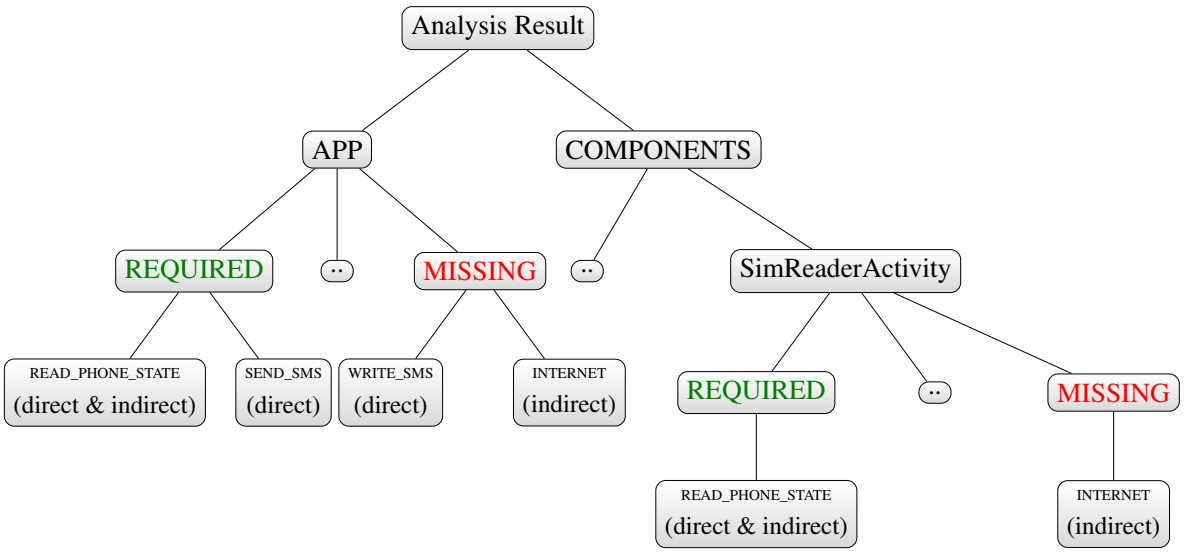

Figure 24: Analysis result for the SimToSms App (Inter-App Permission Usage)

3.6 Intra-App Information Flow Analysis

In this chapter we give some details about the way we implemented the Intra-App Information Flow Analysis (IFA). Special attention is payed to the algorithms and concepts used and to the changes regarding the description in the Architecture Document.

The Intra-App Information Flow Analysis analyzes an application to detect whether there is critical information flow inside that application.

Consider for example again the SimToSms application from the previous sections which is shown in Listing 14. Information is flowing from the statement `getSimSerialNumber()` in the `SimReaderActivity` to `sendTextMessage(...,simSerialNumber,...)` in the `SmsSenderActivity`.

Our IFA detects this critical flow and provides the possibility to display the result. This analysis is structured in the three components `Enhancer`, `GraphGenerator` and `Analyzer` to meet the requirements given by the overall framework. In the following we focus on the `GraphGenerator` and `Analyzer`.

Based on the input given by the `Enhancer`, the `GraphGenerator` builds the Program Dependence Graph (PDG). The PDG then is used in the `Analyzer`, which executes the three steps:

- Compute sources and sinks
- Perform a backward slicing procedure on the Program Dependence Graph
- Find paths between sources and sinks

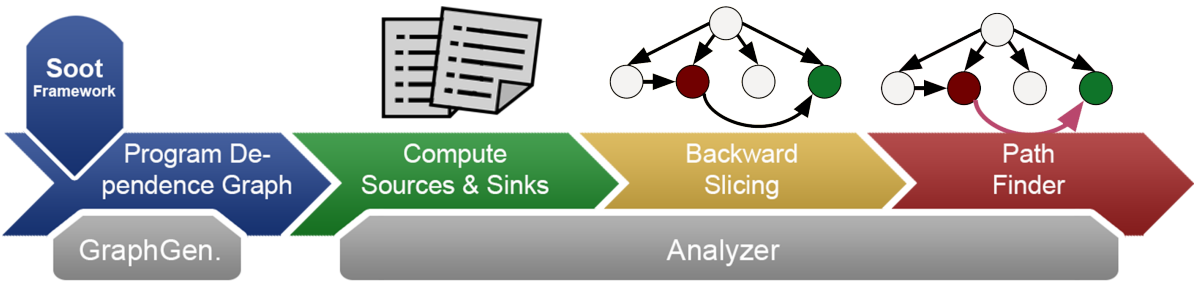

Figure 25: Workflow of the Intra-App Information Flow Analysis

The just described workflow of this Intra-App Information Flow Analysis in shown in Figure 25.

In the following we explain in Section 3.6.1 how we integrated the Soot framework into the Intra-App Information Flow Analysis. In Section 3.6.2 we describe the `GraphGenerator`. This is done by explaining which parts the Program Dependence Graph consists of and how it is computed. Afterwards we describe the `Analyzer` in Section 3.6.3. In this section we give some details about the concept we used to identify sources and sinks. Furthermore, the backward slicing algorithm used for this analysis is described as well as the concept of our path finder. Finally, the Section 3.6.4 deals with the textual and graphical representation of the analysis result for our Intra-App Information Flow Analysis.

3.6.1 Soot framework support

In addition to the support provided by Soot to the Enhancer, the Intra-App Information Flow Analysis requires more information regarding information flow through the Android application under analysis. The Program Dependence Graph(PDG) is built upon the analyzed application's call graph and unit graphs. Unit graph in Soot represents control flow between statements within a method. More details about PDG construction is provided in Section 3.6.2.

To build the PDG, we need the classes, methods and statements of the Android application. In addition we will also need formal actual parameters for methods. We also have to analyze statements that define local variables and statements that use the local variables. To get these information from Soot to build the PDG, we define our own analysis and inject into Soot framework. This is done by a component extending Soot's `BodyTransformer` class. The analysis graph will need unit graphs for each method in the analyzed application. A map of each `SootMethod` and its unit graph is created. This map will be used to add control flow transitions from each method to the statements in that method.

We will be using `SmartLocalDefs` utility offered by Soot to identify statements that define local variables. `SmartLocalDefs` implements Reaching Definition Algorithm to identify statements that define or modify a variable at a particular point of execution. `SmartLocalDefs` implements a forward flow analysis technique. We initialize the constructor

of `SmartLocalDefs` class with the unit graph of a method. The method `getDefsOfAt(...)` will return a list of all statements that define or modify the value of a local variable at a particular point of program execution. Then we obtain the list statements that define each variable. We create a map of statement where a variable is used and statements that define the variable. This map will be used to add Data flow transitions to the analysis graph.

Dummy Main Soot's call graph algorithm is designed to start at a program's single entry point, look for other method calls and the call graph will be constructed. In case of normal Java programs, the entry point will be the `public static void main(...)` method. Soot constructs the call graph with the main() method as a starting node. Then all method calls inside the main() method are obtained and added to the call graph. However, for Android programs, there is no single point of entry. In Android applications, the classes extend pre-defined Android Operating System classes such as `Activity` and over-write life cycle methods such as `void onCreate(...)`. During the application's execution, the Android Operating System instantiates these classes and calls the life cycle methods at pre-defined stages. The call to the method `Activity.onCreate(...)` will never be known to Soot as the call to this method is hidden inside the Android Operating System implementation. Therefore, the generated call graph of the Android application will be empty.

In order to overcome this, we create our own entry point method. This entry point method will model all the calls the Android Operating System will make during the execution of the application. In the following paragraphs, we explain how the dummy main method is created and how invoke statements are added to the dummy main method body. The dummy main method has two invoke the constructor methods of Android component classes, Android life cycle methods and call back methods. We will describe how these methods are identified and how invoke statements are added to the dummy main method.

Creating Dummy Main method The first step is to create a new SootClass with the name `dummyMainClass`. This class will then be added to the Scene and will be set as an Application Class. A new method with the name `dummyMain` is created and is added to the `dummyMainClass`. At the end of first step, we have a SootClass `dummyMainClass` with one empty SootMethod `dummyMain()`.

As stated earlier, to have a complete call graph, the dummy main method should call or invoke all methods which will be potential entry points for the Android application. This is done by adding `Jimple` invoke statements to the `dummyMain()` method body. First we start with creating invoke statements for constructors of Android component classes namely Activity, Broadcast Receiver, Service and Content Provider. In addition, Android applications might also extend the `Application` class to maintain global application state. Class local variables are created using Soot's `LocalGenerator` functionality. Then invoke statements are constructed using this class local variable. We create invoke statement for the constructor of the Android component class. Then we analyze the application to identify Android life

Android Components	Life cycle and Call back methods

Activity	Life cycle methods:
	`onCreate(), onStart(), onResume(), onStop(), onRestart(), onDestroy(), onPause()`
	Call back methods:
	`onActivityStarted(), onActivityStopped(), onActivitySaveInstanceState(), onActivityResumed(), onActivityPaused(), onActivityDestroyed(), onActivityCreated()`

Service	Life cycle methods:
	`onCreate(), onStart(), onStartCommand(),onBind(),onRebind(), onUnbind(), onDestroy()`
	Call back methods:
	`onDeletedMessages(), onError(), onMessage(), onRecoverableError(), onRegistered(), onUnregistered(), onDeletedMessages(), onMessageReceived(), onMessageSent(), onSendError()`

Broadcast Receiver	Life cycle method: `onReceive()`
Content Provider	Life cycle method: `onCreate()`

Table 8: Android components,life cycle methods and call back methods

cycle methods and call back methods. A comprehensive list of life cycle methods and call back methods associated with the Android components is provided in Table 3.6.1. If one or many of these methods are present in the Android application, we create and add invoke statements for these methods to the dummy main method.

For example, consider an Android application having an Activity class `ActivityA` with two life cycle methods `onCreate(...)` and `onStart()`. Then, the dummy main method generated by our tool for this application will look similar to the code in Listing 3.

```
1
2 public static void dummyMain(java.lang.String[])
3 {
4     ActivityA $r0;
5     android.os.Bundle $r1;
```

```
6     $r0 = new ActivityA;
7     specialinvoke $r0.<ActivityA: void <init>()>();
8     $r1 = new android.os.Bundle;
9     specialinvoke $r1.<android.os.Bundle: void <init>()>();
10    virtualinvoke $r0.<ActivityA: void onCreate(android.os.Bundle)>($r1);
11    virtualinvoke $r0.<ActivityA: void onStart()>;
12 }
```

Listing 3: Dummy Main with invoke statements added

Identifying Call Back Methods The last step is to add invoke statements for the call back methods that the application developer might have implemented. In general, there are two ways of implementing call back methods. The developer can implement listeners in the code, which when triggered will invoke call back methods. Call back functions can also be declared in the XML layout files when defining user controls.

Consider the following example in Listing 4. We have a layout XML file, which has a Button control declared. For the Button control, a call back method `onContact(...)` is declared. The XML Layout Parser will identify this method as a call back method. We will then add invoke statements for this method to the dummy main method.

```
1   <RelativeLayout ...
2   .....
3      <Button
4      android:id="@+id/button_contact"
5      android:layout_height="wrap_content"
6      .....
7      android:onClick="onContact"
8
9      />
10  </RelativeLayout>
```

Listing 4: Call back method in Layout

From the `XMLLayoutParser` (see Section 3.2), we obtain a mapping of Layout XML file names and the call back method names defined in each Layout XML file. Even though we can obtain the names of call back methods by parsing layout XML files, we have no information about the activity classes to which the layout files are linked. In the Android source code, an `Activity` class is linked to its XML layout file by the Android library method `setContentView(...)`. The method takes the Resource ID of the layout file as a parameter.

Consider the following example in Listing 5, the layout for the Activity class `MainActivity` is set by calling the function `setContentView(...)` within the `onCreate()` method. The parameter for the method `setContentView()` is an Integer value `R.layout.activity_main`.

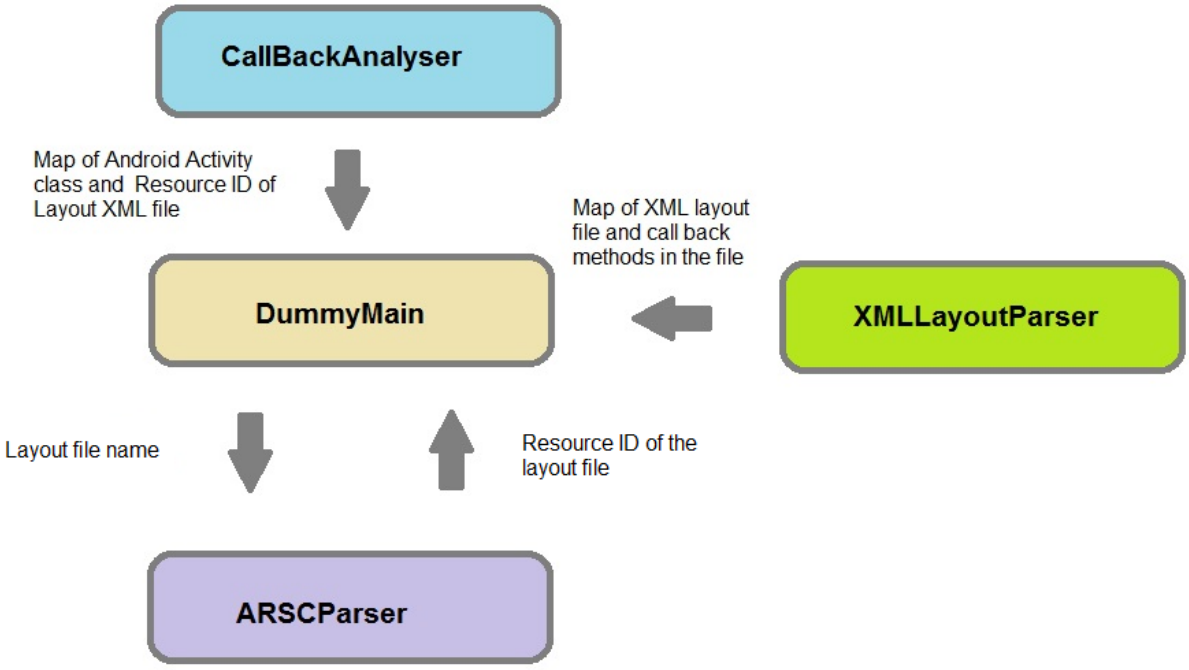

Figure 26: Identifying Android call back methods from XML Layout file

```
1 public class MainActivity extends ActionBarActivity {
2 protected void onCreate(Bundle savedInstanceState) {
3 .........
4 setContentView(R.layout.activity_main);
5     }
6 }
```

Listing 5: Example MainActivity

We have to look up the `ARSC` file to obtain layout file name corresponding to the Resource ID used in the source code. From the `CallBackAnalyser`, we obtain a mapping of Android component class names and the resource Ids of the layout file. Using the `ARSCParser`, we will identify the class names of the Android components.

We analyze the Android application's source code for any call back methods declared. Android provides a set of interfaces, which the developer can use to implement call backs. For example, the developer may want to call a method when a Button is pressed in the application. One way the developer can achieve this is by using the interface `View.OnClickListener` and overriding the method `onClick(...)`. Our aim is to look for these overridden call back methods in the source code. The example in Listing 6 gives an overview of the above described scenario.

```
1  MainActivity extends ActionBarActivity
2    implements View.OnClickListener {
3
4  @Override
5  protected void onCreate(Bundle savedInstanceState) {
6  ..........
7  Button1..setOnClickListener(this);
8  ..........
9  }
10
11 @Override
12 public void onClick(View v){
13 ..........
14 }
15
16 }
```

Listing 6: Activity with call back regitrations

This analysis is done using the class `CallBackAnalyser`. We extend Soot's `SceneTransformer` class. We need a list of Android interfaces that can be used to implement call back methods. This list can be found in the file `/data/androidcallbacks`. We load the Android component classes in Soot and get all the reachable methods. Then we analyze all the invoke statements that implement the call back interfaces. If there is any call back interface implemented, we obtain methods for this interface, check if any method implemented by the interface is present in the analyzed application. In addition, we treat methods created by developers by overriding Android library methods, as call back methods.

Finally, after adding all the invoke statements for class constructors, life cycle methods and call back methods, the dummy main method is complete. This method will then be set as entry point for Soot's call graph construction algorithm.

3.6.2 GraphGenerator: Building the Program Dependence Graph

The Program Dependence Graph is built by analyzing the applications source code using Soot. The first step is to obtain the application's call graph and unit graphs from Soot. A dummy main method is created and set as the entry point for the call graph. Soot builds the call graph with the dummy main method as the root node. More information can be found in Section 3.6.1. The next step would be to add various transitions between the nodes. This will be explained in the following sections.

Control Flow The root node (dummy main method) of call graph is obtained and transitions of type `CONTROLFLOW` are added between the dummy main method and all the methods called by dummy main method. Then all the methods in each of the application's class are

obtained. CONTROLFLOW transitions are added between the methods and the other methods called. Then using the unit graph of each method, we add CONTROLFLOW transitions between the statements in the methods. CONTROLFLOW transitions are added between each method and the first statement of that method.

Data Flow The transitions DATAFLOW are added between statements that define local variables and statements where the local variables are used. We will refer to the statement that defines a variable as `definition unit` and the statement that uses the variable as `use unit`. From Soot, we will obtain a mapping of each use unit in the Android application and a list of definition units that define or modify the local variables that are used in the use unit. We will then add data flow transitions with the definition unit as the source node and the use unit as destination node .

Consider the following example. The method `onCreate(...)` has three units labelled A,B and C. Unit C uses two variables `simNo` and `imeiNo` which are defined by previous units A and B. Our analysis using Soot will return a mapping of unit C and a list containing A and B. DATAFLOW transitions are then added between units A and C and between units B and C.

```
1  public class MainActivity extends ActionBarActivity  {
2  protected void onCreate(Bundle savedInstanceState) {
3  ..........
4
5  String simNo = manager.getSimSerialNumber();  ---A
6  String imeiNo = manager.getDeviceId();        ---B
7  ..........
8
9  explicitIntent.putExtra(SIM_DATA,simNo + "" + imeiNo) ; ---C
10 }
11 }
```

Listing 7: Adding Data Flow Transitions

Control Dependency This flow describes whether a statement is control dependent on another statement. Statement B is control dependent on statement A if the decision whether B is executed or not is dependent on the way statement A is evaluated. Typical statements on which others are control dependent are those, in which conditions are evaluated, e.g. if-statements or while-conditions.

When looking at Listing 8, which reflects the Lines 12 and 35-40 of the SimToSms application (see Listing 14),

```
1 protected void onCreate(Bundle savedInstanceState) {
2     ...
3     simSerialNumber = shortenSim(simSerialNumber, 5);
4     ...
5 }
```

```
6
7  private String shortenSim(String simSerialNumber, int length) {
8      if(simSerialNumber.length() > length) {
9          simSerialNumber = simSerialNumber.substring(0, 4);
10     }
11     return simSerialNumber;
12 }
```

Listing 8: Snippet from SimToSms App

it is obvious that the statement `simSerialNumber.substring(0,4)` in Line 9 is control dependent on the statement `if (simSerialNumber.length() > length)` in Line 8 because its execution depends on whether the if-condition is evaluated successfully or not. Control dependency is needed to take indirect information flow into account. Assuming the integer value `length` contains critical data. Indirect information flow then means that even though there is no information about the value of length flowing directly to the statement `simSerialNumber.substring(0,4)`, we know that if Line 9 is executed then the value of the variable `length` is smaller than the length of the `simSerialNumber`.

In the Program Dependence Graph of our analysis we consider the control dependency to take possible indirect information flow from a source to a sink into account. The control dependency is computed in several steps according to Ferrante, Ottenstein and Warren [2] .

As basis for the computation the control flow graph is needed, since it describes possible execution structures of the program. In most cases the control flow graph is not one connected graph but consists of many independent parts describing the flow of the different methods. Therefore, a common start and end node had to be defined which then leads to the starting resp. ending points of the subgraphs.

Given the control flow graph with common start and end node the main challenge in computing control dependency was to generate the postdominator tree. Statement A is postdominated by statement B if every path from A to the end of the program contains B, so every time A is executed afterward B will be executed, too. The computation is done in our tool with the algorithm by Lengauer and Tarjan [9], which computes a dominator tree for a given control flow graph. The algorithm gets a control flow graph as input an then executes three steps. At first a depth-first search on the input graph is performed to number all the nodes and generate a spanning tree. Afterwards the semidominator of every node except for the root node is computed. A semidominator of a node n describes the smallest node for which a path to n exists such that all other nodes on the path were visited during the depth first search after node n [9].

The last step in the algorithm then computes the dominator for each node. If the semidominator of a node and the spanning tree are known, it is quite easy to compute the dominator out of it. This is the case because the way the semidominators are computed makes sure that if only the subgraph consisting of the spanning tree and edges from each semidominator to its dominated node are taken into account then the dominators are the same as for the original

graph [9]. After the three steps are executed the algorithm terminates and returns the dominator tree.

To compute the postdominator tree one can simply take the reverse control flow graph as input and compute the dominator tree of it which then is exactly the postdominator tree for the original control flow graph.

After we created the postdominator tree, it is evaluated such that for every control flow edge where the target does not postdominate the source we go upwards in the postdominator tree beginning from the target and mark all nodes as control dependent on the source that are visited before we reach the postdominator of that source according to Hammer [4].

At that point in the computation we have added control dependency edges for all nodes which are dependent on a specific statement. The last step that had to be done to complete the control dependency computation is then to add control dependency from all statements that have no dependency edge yet to the corresponding method. This again is needed to model the implicit information flow from the invocation of a method to its different statements.

Parameters The information flow between methods is displayed in our PDG via parameter nodes and special parameter edges. There exist four types of parameter nodes and two types of parameter edges connecting the nodes.

The four types of parameter nodes are

- ACTUAL_IN
- ACTUAL_OUT
- FORMAL_IN
- FORMAL_OUT

and the parameter edges are

- PARAM_IN
- PARAM_OUT

If there is information flowing through a parameter into a method or out of a method then tracking the information through the parameter nodes will make the flow visible. Furthermore, the special parameter edge type is used in the backward slicing procedure to create a context-sensitive slice (see Section 3.6.3 for more details).

Each method with parameters gets one FORMAL_IN node per input parameter. For each invokation statement which invokes one of these methods one ACTUAL_IN node per parameter is created. The corresponding FORMAL_IN and ACTUAL_IN nodes then are connected via the edge type PARAM_IN.

In addition to that, if a method has a return parameter, a FORMAL_OUT node is created for that method and similar as for the input parameters for every invokation statement of that method an ACTUAL_OUT node is created. These nodes then are connected via a PARAM_OUT edge.

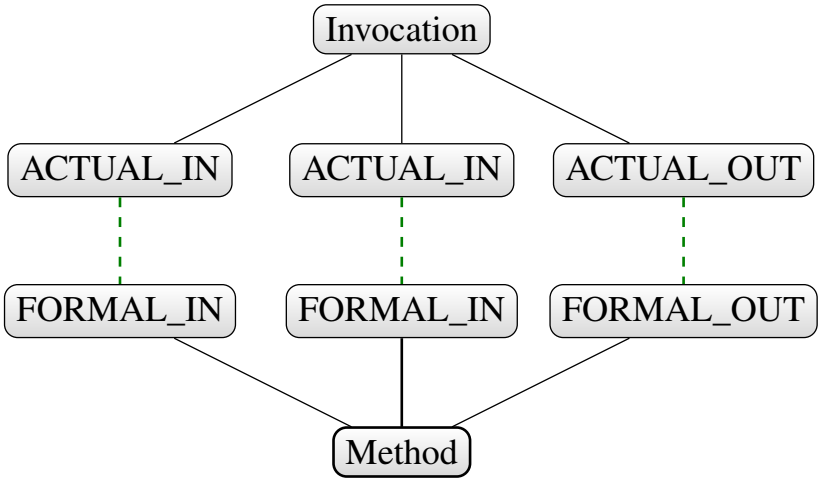

Figure 27: Relations between Parameter Nodes

Figure 27 shows the relations between the parameter nodes. The invocation is the top node which has ACTUAL_IN and ACTUAL_OUT nodes. These nodes are connected via the green lines to their corresponding FORMAL nodes which belong to the method which was invoked by the top node. The green lines therefore indicate the PARAM_IN and PARAM_OUT edges.

The following example visualizes the parameter edge and node creation. The method `shortenSim(...)` from Listing 8 has two input parameters and a return value. In Line 3 it is called with the input values `simSerialNumber` and `5` and the result value is saved in the variable `simSerialNumber`. Our Intra-App Information Flow Analysis would add to the PDG of this example two parameter nodes of type FORMAL_IN representing the input parameters of the method. These nodes are connected with the method via control dependency edges. To the invokation statement in Line 3 two parameter nodes of type ACTUAL_IN are added representing the input values `simSerialNumber` and `5`.

The ACTUAL_IN node representing the value `simSerialNumber` is then connected via a PARAM_IN edge to the FORMAL_IN node representing the parameter `String simSerialNumber`. Similar the ACTUAL_IN node representing the value `5` is connected to the FORMAL_IN node representing the parameter `int length` via a PARAM_IN edge.

In addition to these parameter nodes representing the input a FORMAL_OUT node is added to the method via a control dependency edge. This node displays the return value of the method.

For the invocation statement in Line 3 a node of type ACTUAL_OUT is created and connected to the FORMAL_OUT node via a PARAM_OUT edge.

Summary Edges Summary edges connect ACTUAL_IN and ACTUAL_OUT nodes (see previous chapter) of an invocation if there is information flowing from the ACTUAL_IN node to the ACTUAL_OUT node. These edges summarize interprocedural information flow if such

exists and are needed to reduce computation time to get a context-sensitive backward slice in a later step (see Section 3.6.3 for more details).

The computation of the summary edges is done according to the algorithm presented in [4], which is an optimization of an algorithm presented in [10].

The algorithm uses a worklist in which edges are stored. It starts at the FORMAL_OUT nodes and searches backwards to find a FORMAL_IN node of the same method. If such a FORMAL_IN node is found, a summary edge is added for each invocation from each corresponding ACTUAL_IN node to the ACTUAL_OUT node of the same invocation. For more information on how the algorithm searches for such a FORMAL_IN node see [10].

Consider for example the method `shortenSim(String simSerialNumber, int length)` from the SimToSms application (Lines 35-40 in Listing 14). In this method there is information flowing from the input parameter `simSerialNumber` to the return value. Therefore, for each invocation of this method a summary edge will be added from the ACTUAL_IN node representing the input parameter `simSerialNumber` of the invocation statement to the ACTUAL_OUT node. For the SimToSms application this will be the invocation statement in Line 12.

Call Edges Call edges connect a caller and its call site. These edges were added to our PDG straightforward by searching for callers and connecting them to their callee. The corresponding callee we could get easily through the datastructure given by soot.

Consider for example Line 10 in Listing 14 which is the statement

```
String simSerialNumber = manager.getSimSerialNumber().
```

Since this statement calls the method `getSimSerialNumber` a call edge from this invoke statement to the method `getSimSerialNumber` is added to the Program Dependence Graph.

Call dependencies had to be modelled with this special type of edges to perform a context-sensitive backward slicing procedure in a later step of the analysis (see Section 3.6.3 for more details).

In addition to connecting callers and callees we use call edges to describe the flow that might appear when using explicit intents (see next paragraph for more details).

Intents Since our Intra-App Information Flow Analysis was created especially for Android applications some special cases have to be taken into account. Besides the dummy main method, which was described in Section 3.6.1 one important concept is the way how intents are handled.

The PDG has to be enriched with some additional edges to be able to track whether information is flowing via an intent from one activity to another. We decided to describe this behavior by adding call edges from where an intent is started to the activity the intent starts.

Since we consider only explicit intents in this Intra-App Information Flow Analysis this can be done straightforward. Furthermore, we explicitly had to add dataflow edges representing the flow of information going into that intent up to the point where the intent is started. This had to be done to make sure that the information put into the intent via methods like `putExtra(...)` is tracked, too. This was not covered by the algorithms described in the previous paragraphs because the methods called on the intent are not part of the source code of the application and therefore, the information flow insode that method is traced. But by adding the just mentioned dataflow edges we circumvented this problem.

Possible results of an intent have to be taken into account as well. Therefore, if an intent is started with the method `startActivityForResult(...)`, the statement in which the result is set (`setResult(...)`) is enriched with an edge pointing to the method `onReceiveResult` of the activity that started the intent.

Consider again the SimToSms application. Here the `SmsSenderActivity` is started via intent in Lines 14-16 of Listing 14. While creating the PDG for this application PAndA[2] will add an dataflow edge from the statement in Line 14 to the statement in Line 15, since there information is added to the intent. Line 16 then is connected to Line 15 and finally a call edge from the start of the intent in Line 16 to the activity `SmsSenderActivity` is added. Since the intent is started and a result is expected (`startActivityForResult`) the `setResult` statement (Line 57 in Listing 14) of the `SmsSenderActivity` gets connected to the method `onReceiveResult` of the `SimReaderActivity`.

3.6.3 Analyzer: Finding Information Flow Paths

After the GraphGenerator finished building the Program Dependence Graph the Analyzer starts its work. For the Intra-App Information Flow Analysis this means that based on the PDG as a first step sources and sinks have to be identified. Afterwards, a backward slicing procedure has to be executed to find out from which source to which sink information flow paths exist. Finally, information flow paths have to be extracted. The following paragraphs describe these steps.

Source and Sink Computer The next phase of the Intra-App Information Flow Analysis is identifying source and sink. In Android application, the calls to library methods that get sensitive data of current device's system are considered as sources. On the other hand library method calls which manipulate the sensitive data of current device's system are seen as sinks. In the analysis, since users want to know if there exists any flows of data protected by permissions within a specific application, then only sources and sinks that require those permissions will be taken into account. This phase is implemented by the class `SourceAndSinkComputer` in the component `Analyzer`. It supports APIs that can allow the whole analysis to get a map of source or sink statements associated with their permissions. Those APIs process all statements

(in format of `Jimple Code`) existing in the input Android application to collect sources and sinks.

As mentioned in the target level agreement, the PAndA^2 tool attempts to support two types of sources and one type of sinks. The first type of sources are method or library calls which require permissions or execute on variables protected by permissions. Indeed, only the calls that require permissions for running or getting secured data will be considered. The case execution on variables protected by permissions will be omitted. Because the permissions protecting for such those variables are the ones needed for execution of the method calls which return values to those variables. If any calls, running on those variables, return values and assign them to other variables, then the new ones again become protected by the permissions. The chain of propagation therefore may exist and as a result of it there are many sources. However all of those calls and those variables require the same permissions. That is the reason why only the original calls which need permissions will be considered as source. The second type of source is the return result of a component which requires permissions for starting up. In particular, the method call `setResult` inside a component will be a source with permissions that are required for launching the component. For sinks, only method or library calls that use protected information propagated to their input parameters are considered. With respect to such that types of sources and sinks, the PAndA^2 tool uses a list of defined sources and sinks got from another tool named SuSi [12] (see Table 3.2.2 for more information).

In addition to meet the requirement of the PAndA^2 tool, the list is generated by `SuSi` [12] for Android APIs version 22. The lists of sources and sinks contain both protected and unprotected method calls. However as a remark at the beginning of this phase, the PAndA^2 tool only processes sources and sinks which require permissions.

Based on the lists of defined sources and sinks, the phase checks all statements of an input application and collects those which are considered as sources (or sinks). The phase carries out the process to identify statement by comparing string objects. In particular, the basic information such as name (of package, class and method), as well as parameter's type of a method call in a statement will be extracted. Then based on those information, the phase checks if that method call exists in the list of defined sources or sinks. After identifying a statement as a source or a sink, the `EnhancedInput` object is used to get the corresponding permissions to that statement. If the permission exists, then it and the statement will be added to the result map, otherwise the statement will be skipped. For example in the snippet code `SimToSms` 14, there are two method calls which after being checked by using the list of defined sources and sinks are a source and a sink. The first one is `getSimSerialNumber(...)`. It is a source and requires permission *android.permission.READ_PHONE_STATE*. The other is `sendTextMessage(...)` which is a sink. It needs *android.permission.SEND_SMS* and *android.permission.WRITE_SMS* for execution.

Backward Slicing Computing After building the Program Dependence Graph as well as collecting a list of source and sink statements, the third phase - in the Intra-App Information

Flow Analysis - mainly takes responsibility for extracting a set of source statements which have flow path(s) to a specific sink statement in the graph. Instead of traversing the graph forward from each source in the list of source statements until reaching sinks, this phase will go backward the graph from a sink by applying the backward slicing algorithms [4].

The class implementing this phase is the `BackwardSlicer`. It requires the Program Dependence Graph, and a set of source statements as input parameters to process for each sink. However source and sink statements can be either in a same method (procedure) or even in different methods or classes, this phase therefore not only deals with the intra- but the inter-procedure analysis also. In particular, the class applies the algorithm on the `PDG` which consists of both intra- and inter-procedure information flow (see Section 3.6.2 for more details).

The process starts traversing the graph from the slicing criterion - the specified sink . It is carried out in two steps to deal with context-sensitivity and context-insensitivity. In the first step, the process considers a specific node and then gets a list of its predecessors. The list is filtered out with transition only of type of `DATAFLOW, CONTROLDEPENDENCY, CALL, SUMMARY` and `PARAMIN`. In addition, the first step also ignores parameter nodes which have transitions of type of `FORMAL_OUT`. The purpose of the first step is to make sure that all flows within a method as well as all flows from other callers to the method are checked. A list of all skipped parameter nodes having transitions of type `FORMAL_OUT` in the first step will be processed in the second one. In this step, the process again gets a list predecessors for each of these parameter nodes and only considers transition of type of `DATAFLOW, CONTROLDEPENDENCY, SUMMARY` and `PARAMOUT`. The second step does not take into account the `CALL` and the `PARAMIN` transitions because its purpose is just to deal with the current checked method not the callers. During traversing backward the graph, if a node is an instance of `Unit` and is existing in the list of source statements (the input parameter got from previous phase) then that node is identified as a source which has path(s) to the specified sink. Traversing just stops when no further predecessor of current checked node is found or when the total number of sources found for a sink is equal to the number of sources existing in the input list of source statements. The two steps might have two different sets of sources for the specified sink therefore the final result will be a union of both sets of sources.

Computing Paths Between Source And Sink With the help of the `BackwardSlicer`, we discovered pairs of source and sink between which an information flow exists. For the final step of our analysis, we compute all paths of direct and indirect information flow between the source and sink pairs. We define a path as a sequence of connected transitions in the Program Dependence Graph (PDG). For computing the paths, we introduce the functional class `PathFinder` which we did not mention in our Architecture Document.

Given a pair of source and sink, the `PathFinder` stepwise traverses all transitions outgoing from the source until the sink is reached. Transitions of all types are traversed with two exceptions:

1. We do not consider transitions of type CONTROLFLOW since they are not part of the official PDG introduced by Hammer in [4].
2. We do not consider transitions of type CONTROLDEPENDENCY that point to a parameter node. Those transitions to fixate parameter nodes to the corresponding methods.
3. We do not consider transitions of type SUMMARY because that type of transition is only used to speed up the backward slicing.

By traversing the transitions in a depth-first search behavior, the `PathFinder` creates a path according to the given rules above. We call two paths distinct if and only if they differ in at least one transition. The computation of the `PathFinder` terminates if all distinct paths between the pair of source and sink were found. This is the case when the whole subgraph originating from the source node has been discovered by the `PathFinder`.

After the analysis is finished, the computed paths will be presented to the user as part of the result. Therefore, the found paths are additionally edited and finalized by the `PathFinder`. This means, that the `PathFinder` removes all parameter nodes from the paths since these are only helper nodes for the `BackwardSlicer` and cannot provide additional information in the final result. The removal procedure is the following: Any transition sequence $x \to p_1 \to \cdots \to p_n \to y$, where x and y are nodes of source code elements (class, method or statement) and $p_1, \ldots, p_n$ are parameter nodes, will be replaced by the transition $x \to y$. Moreover, the `PathFinder` adds the source permission at the beginning and the sink permission at the end of each path. By that, all paths contain the required information without any disturbing overhead and, hence, are ready to be shown to the user.

3.6.4 Result Representation

As for the result representations for the analyses described above (see Sections 3.4.4 and 3.5.4), the Information Flow Analysis also provides a textual and a graphical representation. While both representations take different approaches in visualizing the result, they share common detail levels. These detail levels are listed below, ordered from low detail to high detail:

1. RESOURCE TO RESOURCE: Only sinks and sources between which an information flow exists are shown. Elements of the source code like classes, methods and statements are hidden.
2. COMPONENT: Shows the same information as RESOURCE TO RESOURCE. Additionally, Android Components that are involved in the flow are shown.
3. METHOD: Shows the same information as COMPONENT. Moreover, general Java classes and methods are displayed.
4. STATEMENT: Shows the same information as METHOD. Furthermore, all statements that are part of the information flow are shown.

Besides the detail levels, the result representations provide filtering for sources and sinks. The result representations can become huge quickly with increasing detail level. Therefore, we recommend the workflow of starting with detail level RESOURCE TO RESOURCE and increase the detail level only for a subset of flow paths by using filters.

The Information Flow Analysis also supports SUMMARY and COMPARISON mode. In SUMMARY mode all flow paths contained in the App source code are shown. In COMPARISON mode two distinct sets of paths are displayed. The first set contains the removed paths, namely paths that are in the previous version of the App but not anymore in the newer version. The second set encloses the new paths that are not in the previous version but are newly added to the newer version. Paths that stay the same between both versions will not be displayed in COMPARISON mode.

The source code elements that are shown in both result representations are in Jimple syntax. We received this Jimple code from Soot during the decompilation process of the `.apk` file. Jimple code is a representation of Java source code that is based on three-address code. This means that each statement will have at most three components. Therefore, the result might contain additional statements that are not part of the original Java source code.

In the following, we describe the individual properties of the textual and graphical result representation of the Intra-App Information Flow Analysis and give an overview of how to interpret them.

Textual Result Of Intra-App Information Flow Analysis To show the result of analysis after finishing, the PAndA^2 tool provides users a textual mode in both command line and GUI. The result of the Intra-App Information Flow Analysis is mainly shown with the data-flow paths between permissions, if any exists. In the textual mode for GUI, the result is tabular in HTML format with different information for each detail level.

The analysis result is stored in the class `AnalysisResultLvl2a` and `Comparison-AnalysisResultLVl2a` corresponding to SUMMARY and COMPARISON mode. They take the output of the previous phase `PathFinder` to process the results. The list of filters for result contains permissions categorized into two types. One is source permission which protects source method calls. The other one is sink permission which is required for execution of sink method calls (see 3.6.3 for more details of source and sink method calls). If just source permissions are selected, then all flow paths starting from the sources will be shown. It is same for the case in which only sink permissions are chosen, all flow paths from any sources leading to those sink permissions will be displayed. When both source and sink permissions are specified, only flow paths starting from the sources and ending at the sinks will be depicted. In case neither of them exists in the input Android application, obviously no flow path will be found. Furthermore, based on the detail levels, the result can be grouped with common information. For example, the result for the application `SimToSms` shown in the figure below, in the `Method` level, is grouped in same class for methods.

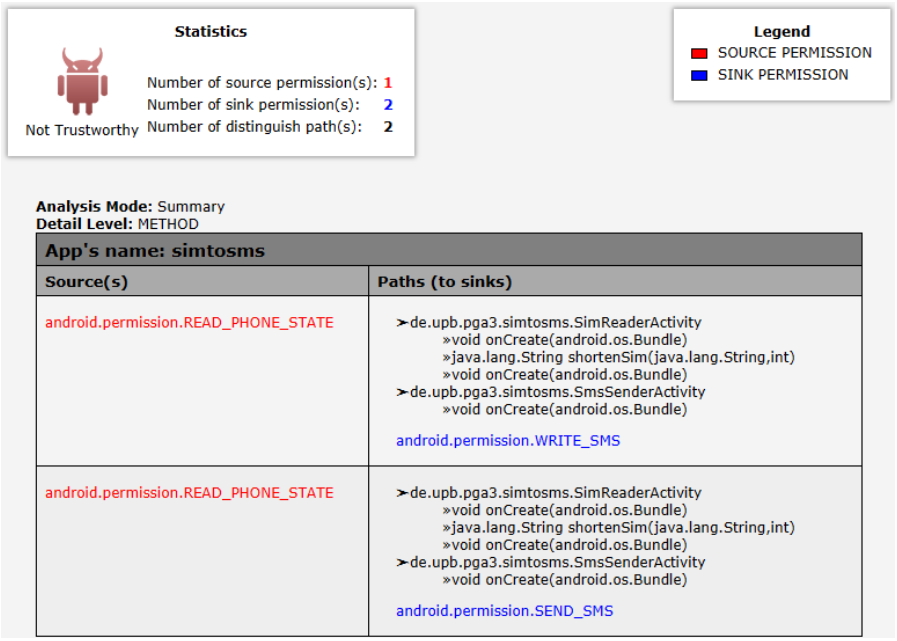

Figure 28: Textual result representation for the App `SimToSms` in Method mode

In addition to the result, the PAndA^2 tool also collects some additional statistic regarding information flow in the Android application. Particularly, the number of source and sink permissions as well as the number of found paths existing between them will be collected. However, the tool only shows distinguish results, if any already exist then they will be omitted. Although the result of an application for each detail levels is derived from the same output of the phase `PathFinder`, the statistics in each detail level are different. For example two different paths starting from a permission to another one, the number of found paths for the detail level `Resource to Resource` is only 1 and for the `Statement` is 2. One important remark for the detail level `Statement` is that the statements shown in result are in `Jimple Code` - an intermediate representation of Java source code when an Android application is disassembled by `Soot Framework`. These `Jimple Code` statements can be changed later on by specific format of users.

Graphical Result of the Intra-App Information Flow Analysis The result of an Intra-App Information Flow Analysis can also be interpreted with the help of a graphical result representation complementary to the textual result representation. In contrast to the textual result representation, this representation tries to give an overview of all information flow paths and their relation to reach other. It gives a good overview of all code elements and permissions that are involved in information flows within the analyzed App. On the downside it can be difficult to track a single path on a high detail level. In this case the textual result representation is more suitable.

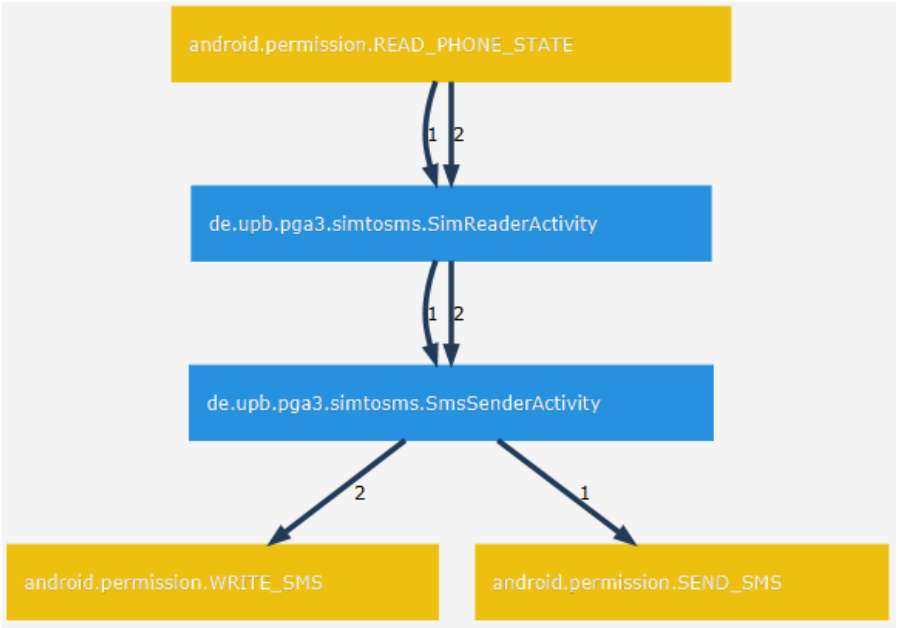

Figure 29: Graphical result representation for the App SimToSms

The graphical representation consists of a graph where nodes are resources (source or sink), classes, methods and statements. A special feature of the graph is that nodes can be nested. In our case method nodes are nested inside class nodes and statement nodes are nested inside method nodes. The types of nodes shown in the graph depend on the chosen detail level. The transitions in the graph represent the discovered information flow between nodes. Each transition has a path index attached to it. The index can be used to track a complete information flow path through the graph. Moreover, the graphical representation provides the highlighting of a path by clicking on a transition that is part of the path. Unfortunately, this useful feature does not work within the PAndA^2 GUI for now. But the result representation can be opened in a general web browser where the path highlighting works fine.

In SUMMARY mode the graphical representation looks straight forward like described above. For COMPARISON the paths and, hence, the transitions are colored for differentiation between removed and newly added paths. Removed paths are shown in red color, while new paths are shown in green color.

Figure 29 shows the graphical result represtaion of the Intra-App Information Flow Analysis for the App SimToSms. The representation is in detail level COMPONENT. It shows two paths from one source to two sinks. The two paths are differentiated by their labels. The first path labeled with 1 starts at the source *android.permission.READ_PHONE_STATE* and ends in the sink *android.permission.SEND_SMS*. The other one ends in the sink *android.permission.WRITE_SMS*. The source and sink of both paths are serially connected through two Android components, namely `SimReaderActivity` and `SmsSenderActivity`. If the graphical result is displayed in a browser both paths could be selected and thereby highlighted.

4 Extensibility

This section describes how PAndA^2 can be extended. Therefore, Section 4.1 deals with the question how to include a new analysis or other result representations to PAndA^2 . Afterwards, Section 4.2 describes the approach to change the user interface and finally Section 4.3 deals with the API level the tool supports and how this can be adapted.

4.1 Adding a New Analysis

For extending the functionality of PAndA^2 by adding new analyses, the tool framework provides several interfaces and classes to work with.

The interface `AnalysisProcedure` has to be implemented as the main part of the analysis. The resulting class should contain the complete logic of the analysis. The interface provides two methods for performing an analysis on an `.apk` file and for performing an analysis on top of preceding analyses. The latter one can be user for analyzing multiple Apps. Both methods return an instance of type `AnalysisResult` which will be passed to the client.

Developers have to extend the general class `AnalysisResult` which stores the outcome of the implemented analysis. Therefore, the methods for passing a textual and a graphical result to the user have to be implemented. Both methods return a String. For the textual result we specified the String to contain a HTML5 document. We recommend to use table and list elements for providing a well-structured result representation to the user. In Figure 30 we show an example of how a textual result could look like. It has a header section containing a statistics block and a legend. The statistics block summarizes the result. Below that is a list with detailed information about the outcome of the analysis.

Figure 30: Example of a textual result

For the graphical result, we refined the specification according to the Architecture Document. The graphical result will also be provided in form of a HTML5 document. However, since the result should be of graphical nature, the document should mainly consist of SVG elements. Compared to the Graphviz DOT Language that we wanted to use initially for the graphical result, SVG gives us more freedom in choosing a type of visual presentation. For example one could use graphs or charts for visualizing the analysis result. In Figure 31 we show an example of how a graphical result could look like. The figure shows also a header with statistics and legends. The statistics can be identical with the textual result representation but does not have to be. The main content of the graphical result

consists of a pie chart that might show relations of different part of the outcome of the analysis. On the right is also a bar chart that might show some absolute numbers of the outcome.

For creating a header section with statistics and legend, our framework provides the class `HTMLFrameBuilder`. This class creates a prestyled header where developers can add custom rows for statistics and legend. Moreover, it provides several options for interaction and look, like the option to hide the header automatically when the mouse hovers over the result.

Figure 31: Example of a graphical result

In general, we recommend to be creative and try to provide a most satisfiable results to the user. Therefore, a styling and coloring should be applied that supports the readability and understanding of the results. Moreover, we encourage developers to include tool tips or other interactive features that can be achieved by using JavaScript or CSS. Inspirations for results can also be taken from the result representations of our analyses. However, we also have to sensitize the developers that using complex interactive HTML5 documents might not be able to be interpreted by every GUI. In addition, we do not recommend to use any remote resources from the web within the results since all analyses should also work offline. Last but not least, we want the developers to be careful while developing the result representations since they are responsible for creating valid HTML Strings. The framework does not check the Strings for validity.

4.2 Integrating a new User Interface

In this section, the reader can find details about how to integrate a new User Interface to our tool PAndA^2 . To develop a User Interface the developer can follow any of the Architecture models of their choice, but MVC will be a nice choice to develop on.

The newly developed User Interface should provide the options for the user to choose the different levels of analysis and its modes if there is any. For example, in GUI, the user should be provided the option to select the specific level of analysis and its mode. The input should be validated by the User Interface so that the input parameters are correctly passed and are in the scope of our available parameters. For example, the primary input should be the .apk file, so the User Interface should be able to validate that the file is of .apk format. In our tool PAndA^2 , we are using JCommander to validate the user inputs in Command Line interface. The developer can also use other libraries like JLine, JOpt Simple, JSAP, etc for Command Line Interface.

The developer can call the `AnalysisFactory` from our project PAndA^2 . The `AnalysisFactory` is created depending on user input, since there are three different levels of analysis that can be performed by our tool PAndA^2 , a separate `AnalysisFactory` is created for each level of analysis. This value from the `AnalysisFactory` is eventually passed

to `AnalysisRunner` and the analysis is started. The developer can directly call upon the functionality of `AnalysisFactory` and `AnalysisRunner` in their User Interface. Since the components `Client`, which is the User Interface and the `Analysis` are independent of each other it would be easy to use them separately.

In the `AnalysisRunner`, our entire analysis is performed depending on the user input. We have `Enhancer`, `GraphGenerator` and `Analyzer` incorparated in the `Analysis` component. Each sub-component performs a specific task during the analysis. The task performed for each of the sub-component has been covered in detail in previous sections of our documentation.

After the analysis has been complete `AnalysisResult` is generated, which is an HTML document. With the HTML document, the result can be displayed in message, textual and graphical representation. The graphical result can be presented using libraries like Graphviz DOT Language or SVG elements, which we have used in our project to display the result. CSS or JavaScript can be used to obtain the textual result.

In our project PAndA^2 , to perform the analysis based on user input we have implemented a `Client` Class, which has all the functionality to perform our different level of analysis. By extending `Client` class we have already developed `ClientCommandLine` and `ClientGUI` to run our tool through Command Line and Graphical User Interface.

4.3 Changing API

The first commercial version of Android API, Android 1.0 was released on September 2007. Since then, Google has released many versions of Android API.

In our tool, to generate the intermediate representation code, which is essential for all the levels of our analysis, we should pass the API as input. To do that the Jar file of the API has to be stored in the data folder in the PAndA^2 implementation folder. This file path is directly used in `SootAdapter` class in the implementation folder of PAndA^2 package, to import the jar file and to generate the intermediate representation code. Currently we are using Android API 22, So the Jar file of API 22 is stored in the data folder of the project, if we want to change the API version we should replace the specific API's Jar file with the current one.

PAndA^2 simultaneously uses the API to get the API's permissions, the API defines the permissions of various versions of Android libraries. This information is used to link the layers and then map the permissions of many versions of Android library. Our tool interacts with the `DataStorage` while performing analysis to get all the APIs defined permissions of a specific version of Android library. This information will be used for mapping and linking the layers in the `Enhancer`.

In order to get the list of permissions, we use a tool called PScout [6]. PScout leverages the Soot Java bytecode analysis framework to perform static analysis. Extracting permissions from the Android source code is done in three phase. First, PScout identifies all the permission

checks in the Android source code and labels them with the permission that is being checked. Then, it builds a call graph over the entire Android source code including IPCs and RPCs. Finally, it performs a backwards reachability traversal over the graph to identify all API calls that could reach a particular permission check. Then PScout generates the following files

- All Mapping
- Published API mappings
- All API call mappings
- Intents Permissions
- Content Provider (URI Strings) with Permissions
- Content Provider (URI fields) with Permissions
- Android callbacks

More information about PScout generated files can be found in table 2.

The permission files are stored in the Data folder of our project implementation. The `DataStorage` uses these permission file's absolute path and imports these permissions as a map, since its the mapping between API calls and the permissions. Then we have to specify the API version in the apilevel file stored in data folder. We have to make sure that the version entered in apilevel file and the files generated from PScout are of the same API level. For example, if you have entered API level 22 in apilevel file then you should make sure that you stored the permission files generated by passing API 22 to PScout.

There are two ways to get the API permission files, we can either download the permission files for the specific API versions directly from their website [5]. If the permission files for the specific API is not available on their website, then we can always run PScout to get them. PScout can be download from their website and it is available for the users under General Public Licence.

5 Quality Assurance

This paragraph provides an introduction about quality assurance and what approaches, we have taken to achieve it during the development of our PAndA^2 tool.

Customers do not like to deal with defective software. They want their demands to be delivered with high quality and that is where the need for quality assurance comes. Quality assurance in software development or Software Quality Assurance (SQA) consists of processes and methods which are intended to establish the quality, performance, or reliability, especially before it is taken into widespread use [5].

[5] `http://pscout.csl.toronto.edu/downloads.php`

The sources of bugs in a software system can range from one to hundreds. Caused by programming errors, dependencies between code modules, poorly designed/documented code and much more. Bugs cause software failure, which leads to loss of business, loss of data, loss of money and sometimes the impact is so catastrophic that it even claims human lives. Hence Software Quality Assurance assures quality software which is reasonably bug-free, delivered on time, meets requirements and/or expectations, and is maintainable.

With respect to our tool, at several checkpoints we have evaluated the quality and consistency of our tool to meet the requirements and specifications presented in Architecture Document. We have followed standard procedures to drive our system, such as

- Obtain requirements.
- Determine project-related personal and their responsibilities.
- Determine test approaches and methods (Black box, White box, unit, integration etc whichever are in scope.)
- Determine testing requirements (tools for coverage tracking, code quality and automation etc.)
- Set initial schedule estimates, timelines, milestones where feasible.
- Write test cases or test scenarios as needed.
- Perform tests.
- Track problems/bugs and fixes.
- Follow a particular software lifecycle.
- Ensure functionality check along with development.
- Maintain coding standards and quality.

From product's initial development till its delivery, all components of the tool are systematically subjected to various tests. From time to time utmost care has been taken to meet our goal and objectives by keeping the requirements in mind. Some of the initial above points are already described in Target Level Agreement, Architecture Document and to describe remaining points, the content of this section is divided into the following sub-sections:

Types of Testing (Section 5.1) describes the approaches that we have taken for testing, in our project.

Tools (Section 5.2) illustrates about the tools that we have used in Eclipse to ensure quality and coding standards.

Automated Test Executor (Section 5.3) describes the functionalities of the test executor that we have built to run numerous test cases at one go.

5.1 Types of Testing

Testing means 'analyzing' operation of a system or application under controlled conditions and evaluating the results. The controlled conditions should include both normal and abnormal criteria to detect the behavior of a system, i.e. tester should intentionally attempt to make things go wrong to find and eradicate all the behaviors which shouldn't be there in the system and keep only the expected ones.

We have followed standard testing techniques as follows throughout our development to ensure quality in our tool:

5.1.1 Black box / Functional Testing

Black box testing is not based on any knowledge of internal design or code. The tests are based on requirements and functionality, hence sometimes combined with Functional testing as well.

Test Case Test cases are built around specifications and generally derived from descriptions of the software and requirements. Hence at first, we have gone through our architecture document thoroughly to list down all the requirements of our tool. Then we have created a bunch of small Android applications named as `Testing Apps` covering our requirements. These testing applications are kept under folder `test_resources` of our PAndA^2 project and listed in Table 9 in three different groups along with the requirements covered by them.

As per the requirement of Black Box testing, the tester should be aware of what the software is supposed to do. As a developer of the testing Apps, we knew what functionality it poses and what could be the expected output when they are going to run in our tool. So we have used these applications as test cases for our tool with the progress of development and accomplishment of new functionalities. These tests are primarily functional in nature only to test the functionalities of our tool.

5.1.2 White box testing with Unit Testing

White Box testing is based on the knowledge of the internal logic of an application's code. The tests are based on coverage of code statements, branches, paths and conditions. We have combined White box testing with Unit testing as both focus on testing the functionality on a granular level.

Unit Testing Unit testing is the most 'micro' scale of testing to test particular functions or code modules. It requires detailed knowledge of the internal program design and code. Hence it is performed by developers for better reachability of code and goes side by side with development.

Testing Apps	
Group	Covered Requirement
Apps for Intra-App Permission Usage	• contains permissions related to some or all 5 permission-group i.e. REQUIRED, MAYBE_REQUIRED, UNUSED, MAYBE_MISSING , MISSING
Apps for Inter-App Permission Usage	• contains cases for detecting Implicit intents or Explicit intents • contains cases for detecting both Implicit and Explicit intents • contains cases for detecting Direct and Indirect permissions related to different permission-group
Apps for Intra-App Information Flow	• contains CALL BACK methods • contains life cycle methods of Android Activity • contains life cycle methods of Android Broadcast Receiver • contains life cycle methods of Android Service • contains life cycle methods of Android Content Provider • contains different sets of Sources and Sinks • contains different sets of flow paths between Sources and Sinks • contains conditional case for detecting flow paths

Table 9: List of Testing Apps

Unit testing focuses on evaluating a single operation on a set of data rather than large functions performing a number of different operations [5]. So the test cases emphasise more on testing 'behaviour' rather than testing methods. A few advantages of Unit testing are as follows:

- Gives us the ability to verify that all behaviours work as expected with a set of inputs. Also, we can determine if the function is returning the proper values and handling failures during the course of execution with valid and invalid input.
- Helps us to identify failures in our algorithms and/or logic to help improve the quality of the code as the development progresses with addition of new functionalities.
- Code remains well-tested so that we can prevent future changes from breaking the existing functionality. Hence, as the project grows, we can simply run the tests that are developed to ensure that existing functionalities aren't broken when new functionalities are introduced.

JUnit We have used the JUnit testing framework for writing unit test cases. JUnit is a regression testing framework for the Java programming language and can be easily integrated with Eclipse [13].

Important features of JUnit test framework that we are leveraging are as follows [13]:

Fixtures is a fixed state of a set of objects used as a baseline for running tests.

- setUp() method which runs before every test invocation.
- tearDown() method which runs after every test method.

JUnit classes are the classes which is used in writing and testing JUnits.

- Assert which contain a set of assert methods.
- TestCase which contain a test case defines the fixture to run multiple tests.

For writing unit test cases, we have followed a set of rules to keep the code understandable and maintainable. Sample code of a test class will look like,

```
1  public class ManifestXMLParserTest {
2
3  ManifestXMLParser mp;
4  EnhancedInput ei;
5  String path;
6
7  @Before
8  public void setUp() throws Exception {
9
10 this.path = "test_resources/SimpleIntents.apk";
11 this.mp = new ManifestXMLParser(this.path); }
12
```

```
13 @Test
14 public void getLstActivitiesValuetest() {
15
16 final List<String> activities;
17 activities = this.mp.getLstActivities();
18
19 assertEquals("com.mycompany.simpleintents.
20 MainActivity", activities.get(0).toString()); }
21
22 @Test
23 public void getLstActivitiesNegativetest() {
24
25 final List<String> activities;
26 activities = this.mp.getLstActivities();
27
28 assertNotEquals("com.mycompany.simpleintents",
29 activities.get(1).toString()); }
30 ..... }
```

Listing 9: Sample Test Class

- We have created a separate folder called `test` under our project PAndA[2] and written all the test packages (de.upb.pga3.panda2.test.*) under it following the same hierarchy similar to the source code folder.
- We have created test classes following the same package hierarchy and same name as in source code with a keyword 'Test'. For example, test class `ManifestXmlParserTest.java` for source class `ManifestXmlParser.java`.
- We have created test methods following the same name as in source code with a keyword 'test' plus the functionality that is being tested. We have written one test method for each functionality. So, if any method has different functionalities, then we have written separate test methods to test the functionalities separately.
- In each test class, we have created the mock-ups (test data) in `setUp()` method. Then we have written small test methods and verified through *ASSERT*. For example, we have written two test methods `getLstActivitiesValuetest()` and `getLstActivitiesNegativetest()` for source method `getLstActivities()` to check one positive and one negative case.
- We have used the testing Apps as listed in Table 9 for creating the mock-ups (test data) and written our test methods on top of that.

5.2 Tools

In this subsection, we have briefly described the tools that we are using in our project for tracking code coverage and improving coding standards. After analyzing many tools that are

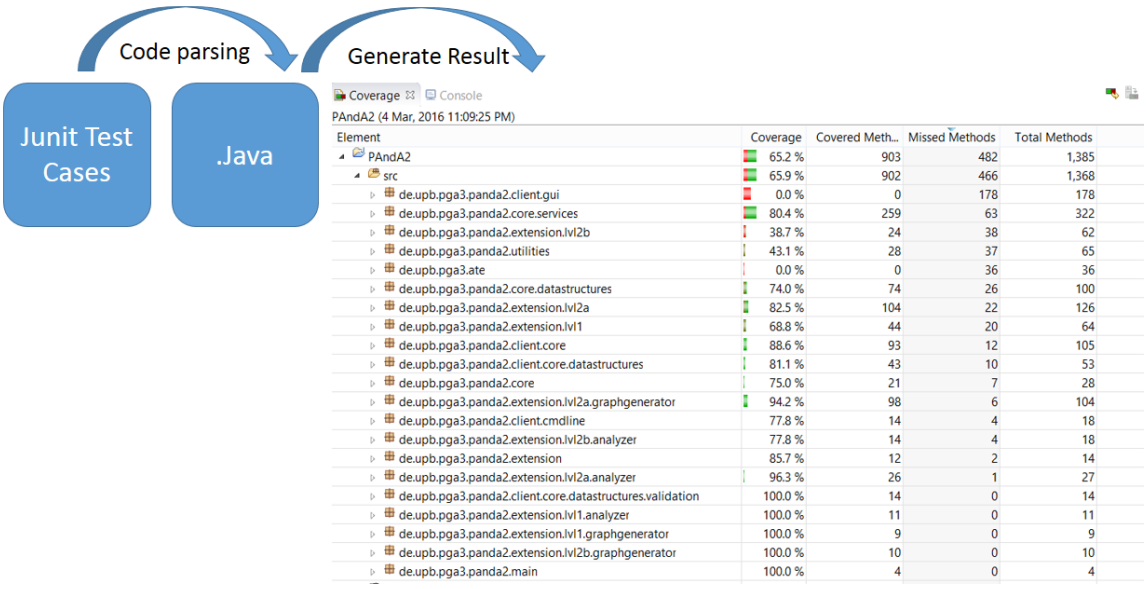

Figure 32: Code Coverage displayed in Coverage window

currently available in the market, we have chosen the below mentioned tools and leveraged its functionalities to achieve our goal. Also, we have described how far we have succeeded in achieving our goal with the help of these tools.

5.2.1 EclEmma

EclEmma is a free Java code coverage tool for Eclipse, available under the Eclipse Public License [7]. It basically helps in finding how much code has been covered by the JUnit test cases written.

Why are we using EclEmma For code coverage, many tools are available in the market such as Cobertura, CodePro, Coverlipse etc. But, we have chosen EclEmma for our project because of its simple integration method with Eclipse and easily understandable result summarising technique. Features of this tool that we are using and makes it different from the other existing tools are as follows:

- Analysis: JUnit framework can be easily integrated with EclEmma for code coverage analysis. After the analysis is finished, coverage results are immediately summarized and highlighted in the **Coverage View** and in Java source code editors [7]. This helps in finding out the percentage of code reached drilling down from package level to method level on **Coverage View** as shown in Figure 32.

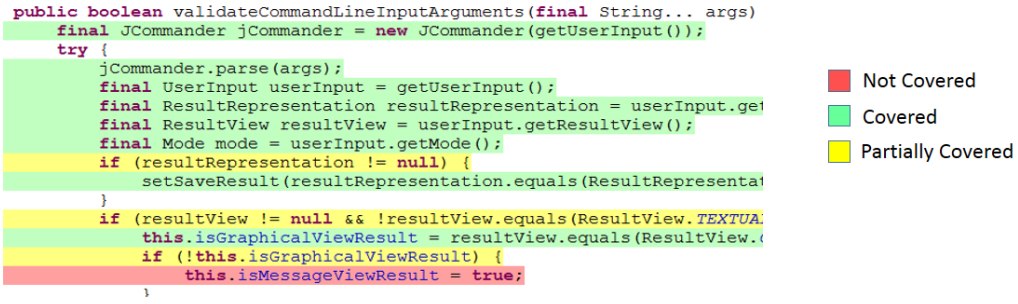

Figure 33: Color code displaying the coverage in Java source editor

- Code highlighting: The result of a coverage session is directly visible in the Java source editors [7]. For highlighting, we are using GREEN for full coverage, YELLOW for partial coverage and RED for not covered lines as shown in Figure 33. This helps in analyzing the areas for which additional test cases are required.
- Counters: It allows to summarize the coverage in instructions, branches, lines, methods etc. It helps in analysing the coverage in different perspectives.

Usage We have used EclEmma for tracking our progress in writing JUnit test cases. As per the best practices, we should test as much functionality as possible of our implementation with JUnit test cases. Hence the tool helps in keeping a track on the code that is being reached through JUnit test cases.

We have written test cases as described in the subsection 5.1.2, along with our development so that the unit testing runs in parallel with the implementation phase. But it is not sufficient to write only unit test cases. It is necessary to check if all possible areas are covered functionality wise. From that perspective, testing team have monitored our progress with respect to code coverage (method-wise) in every 10/15 days throughout development phase by using this tool as shown in Figure 34. Tracking of progress from the result shown is actually very easy as the user can check the area covered showed in percentage (%) starting from the package level to the method level. Also user can open the code in the Java editor window and track the area covered line-wise by looking at the customizable color code highlights.

Target and Outcome We had initially set our goal to cover at least 70% of the code through unit test cases. Due to the usage of EclEmma, we were able to track down the areas not covered through test cases, which actually lead us to decide what kind of unit test cases are needed and in what number to cover the remaining functionalities. Hence we have achieved our target and in fact, we have written 265 JUnit test cases covering more than 80% of the overall code

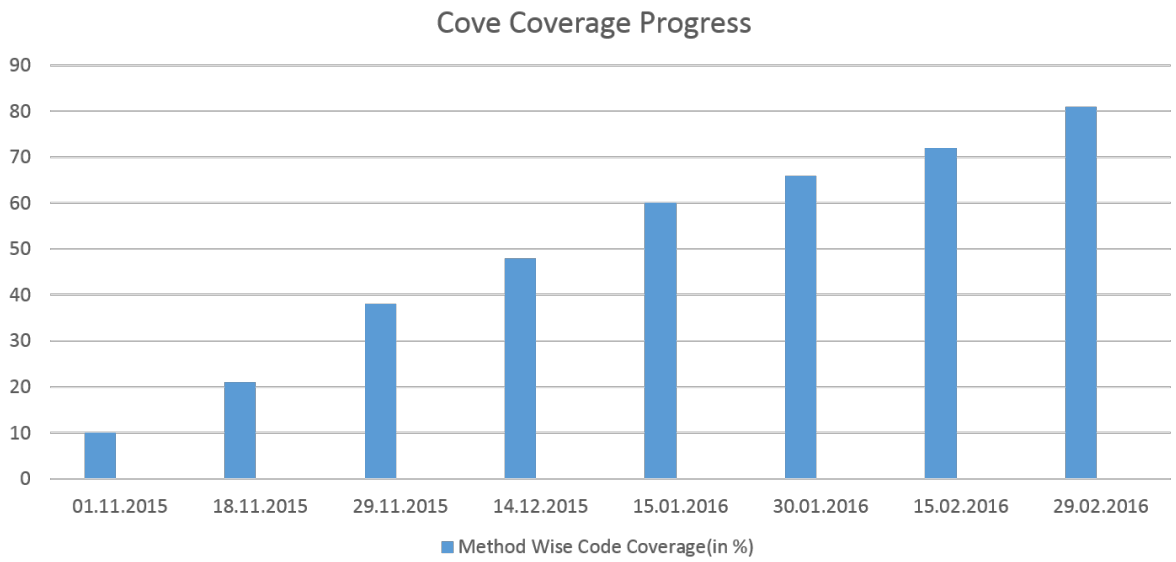

Figure 34: Code Coverage tracking with time

(method-wise) except GUI. So currently 971 methods have been covered out of 1171 methods (except GUI) through JUnit test cases.

5.2.2 PMD

PMD is an open source project designed to inspect Java code and point out possible bugs, dead code and overcomplicated expressions [1]. It allows programmers to structure their code.

Why are we using PMD 'Good code' is the necessity of a quality software which not only works, but is reasonably bug free, secure, readable, maintainable and is properly structured [5]. Normally organizations have their coding 'standards' that all developers are supposed to adhere to. We have used PMD for following a particular coding standard in order to achieve quality code in our project.

We have chosen PMD over other existing similar tools like CheckStyle, FindBugs because of its rich customization features along with user-friendly results displaying technique.

Usage Features of this tool that we are using and makes it distinguishable are as follows:

- Violation overview: After the analysis has finished, code violation information is automatically available in PMD perspective. Then, users can look for violations drilling down from package level to class level in **Violation Overview** and the exact line number in **Violation Outline** window as shown in Figure 35. This feature has saved a lot of

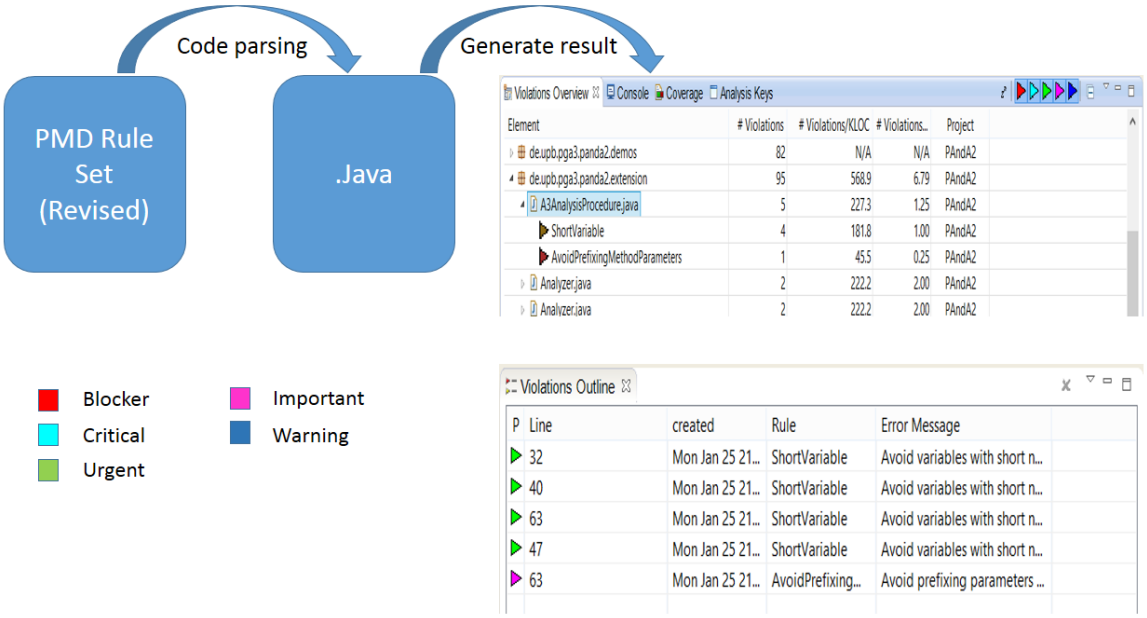

Figure 35: Code violations shown in Violation windows

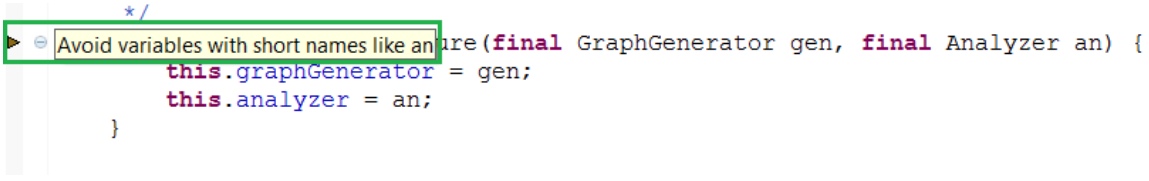

Figure 36: Code violations shown in Java source editor

```
<?xml version="1.0" encoding="UTF-8"?>
<ruleset xmlns="http://pmd.sourceforge.net/ruleset/2.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         name="PMD_Rules4_Custom"
         xsi:schemaLocation="http://pmd.sourceforge.net/ruleset/2.0.0 http://pmd.sourceforge.net/ruleset_2_0_0.xsd">
   <description>PMD Plugin preferences rule set Ver 4_Custom</description>
   <exclude-pattern>.*/PAndA2/src/de.upb.pga3.panda2.client.cmdline/CommandLine.*</exclude-pattern>
    <exclude-pattern>.*/PAndA2/test/.*</exclude-pattern>
   <rule ref="rulesets/java/design.xml/AbstractClassWithoutAbstractMethod"/>
    <rule ref="rulesets/java/design.xml/AbstractClassWithoutAnyMethod"/>
```

Figure 37: PMD Rule set XML file

our effort as the developer can directly go to the exact line where the code violation has occurred and take appropriate decisions to tackle it.

- Color highlighting: The result of Violation Outline is directly visible in the Java source editors as shown in Figure 36. Color code highlights for identifying different sets of rule violation, i.e. RED for Blocker, TURQUOISE for Critical, PINK for Important, GREEN for Urgent and BLUE for Warnings. Through this we have analysed which kind of rule violation has occurred and how severe it is.

- Analysis: PMD inspects Java code using a rule-based approach. It includes a series of rules considered common in every Java application [1]. They are organized in rulesets inside an XML-based file as shown in Figure 37.

  By default PMD 4.0 gives 347 rules for checking code violations. We have revised the ruleset, according to our usage and hence, currently we have 252 custom rules in use grouped by Blocker, Critical, Important, Urgent and Warnings.

Outcome All the developers have used PMD for finding the code violations and analysed them according to their severity.

In the course of time, PMD has helped us in finding problems like `AvoidTrailingComma`, `AvoidBranchingStatementAsLastInLoop`, `DuplicateImports`, `EqualsNull`, `GuardLogStatement`, `SystemPrintln`, `UnnecessaryFullyQualifiedName`, `UnusedImports`, `UselessParentheses`, `UseVarargs`, `UnreachableCode` etc. Refer SourceForge.Net Rulesets[6] for more information about the code violation rules. As per the availabilty of time, we were able to remove the violations under Blocker, Critical and Important priority from our code base ignoring some of the violations due to the project specific implementation.

Though we had very less time for ensuring coding standards, still we were able to remove the most critical code violations from our code with the help of PMD in order to achieve a quality code base.

[6] `http://pmd.sourceforge.net/snapshot/pmd-java/rules/index.html`

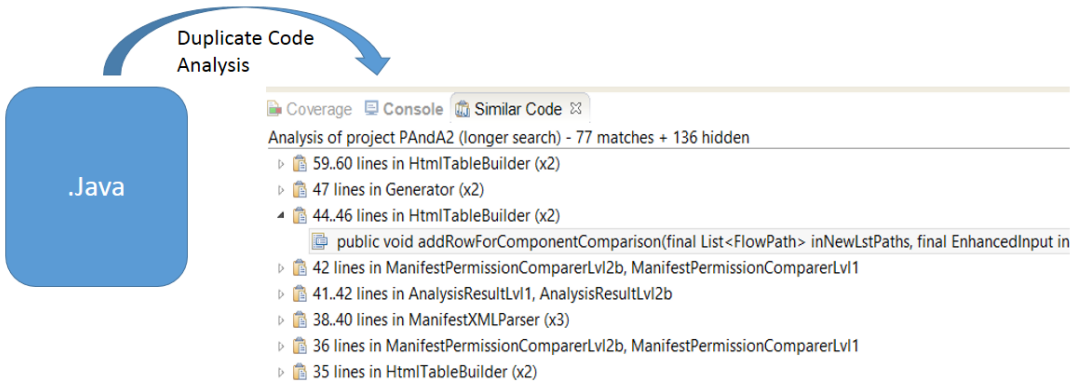

Figure 38: Duplicate Code displayed in Similar Code view

5.2.3 CodePro

CodePro Analytix is a Java software testing tool for Eclipse for improving software quality and reducing development costs. It is offered as a free download by Google [3].

Why are we using CodePro Similar kind of functionalities and copied fragments, leads to code duplication in a project. That is where the need of *Similar/Duplicate Code Analysis* comes.

We are leveraging *Similar Code Analysis* from CodePro Analytix tool due to its duplicate code analysing technique on a very granular level. It not only looks for very similar code throughout the project, but even finds the copied code with renamed or manipulated variables. This makes refactoring code much faster and more efficient.

Usage Features of this tool that we are using and makes it user-friendly are as follows:

- Analysis: **Similar Code** view opens automatically when the analysis is complete and the list of matches is displayed as shown in Figure 38, drilling down from package level to classes. The user can click on a match to open the **Compare Editor** as shown in Figure 39. Time to time, our testing team has *Exported* the result for later use or to share among team members.
- Color highlighting: The editor shows textual differences between the matched code snippets. GREEN lines designate identical code, YELLOW lines designate differing code (with differing tokens highlighted with RED background), and RED lines designate inserted/removed code.

  This feature has saved a lot of our effort as the developer can directly look for the exact difference in the piece of similar code and take appropriate decisions.

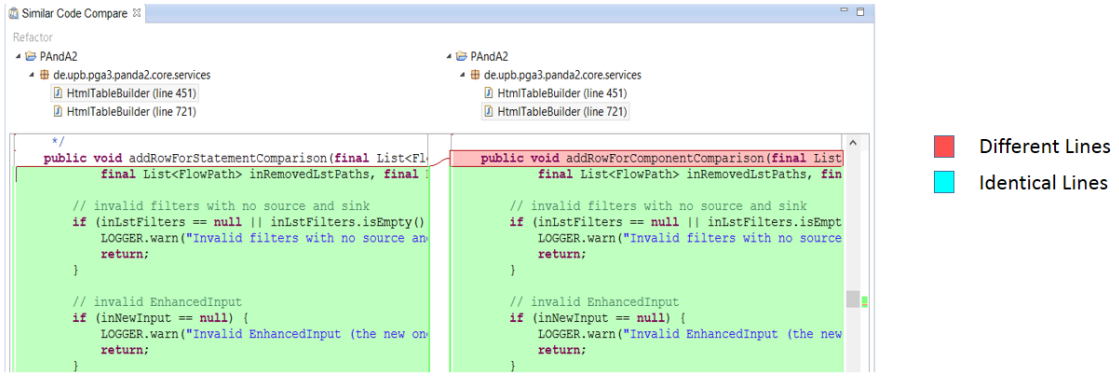

Figure 39: Exact lines no. for matching code in Compare editor

Outcome For reducing the code duplication and for further code optimization, our testing team has generated and analysed the similar code report time to time. We have found duplicate code pieces in areas like `GraphGenerator`, `Analyzer`, `Client` due to similar functionalities. With the analysed report, along with the developers we have taken further decisions to remove the code duplications. We have optimized the code where ever possible in our code base ignoring areas like `Intra-app Resource Usage`, `Inter-app Resource Usage` for keeping a separation of concerns.

Though we had very less time for ensuring coding standards, still we were able to remove most of the possible code duplications with the help of *Similar Code Analysis* in order to achieve a quality code base.

5.3 Automatic Test Executor

As driving our development process sprints, there are changes, updates regularly. Alterations to the source code of application can ripple outward in surprising ways, breaking functions that seem completely unrelated to the new modification. So to make sure our tool still adhere to the functions throughout the cycles, `Regression tests` i.e the test carried out after next releases to safeguard the working of previous functionalities, needed to be carried out. Because when we run regression tests, we are checking to make sure that our modification not only behaves as we want it to, but that it also has not inadvertently caused problems in functions that had otherwise worked correctly when previously tested. Each time we modify our source code, we had simply re-ran the potentially relevant tests to ensure that they continue to pass. For the same cause (*RegressionTesting*) we have built our own separate tool and named it Automatic Test Executor (ATE).

Automatic Test Executor has its defined Test cases categorized. Test cases are made in systematic defined hierarchical manner. Automatic Test Executor executes the Test cases

sequentially using threads running on the nightly build and displays the result output on GUI screen. It also has got the functionality to run on some previous builds (previous version of tool as jar file input) rather nightly build which helps in the evaluation procedure. And on the top of its basic function i.e. to evaluate the output of functional behavior of the tool, it also performs basic analytics over the analysis result generated by PAndA[2] .

Coming to basic outlook of all the functionalities that Automatic Test Executor performs, they are as follows

- Consistency of tool with real-world `.apk` files
  To make sure our tool is consistent with real-world `.apk` files with its functionalities, we are analyzing `.apk` files in sets whether after updates or any alteration, is the code still adhere to all the previous functionalities. Some big applications like *WhatsApp* and application that needs bunch of permission like *FileExplorer* with their different versions are taken into consideration with some of self-developed `.apk` files.
- Expected Result Analysis
  This functionality in Automatic Test Executor performs tests to check the presence of expected results in the original Analysis result computed by tool. As we have three levels of Analysis in our tool so for all the levels, we can predict the result and check its existence as

  For Permission Usage (Intra App- Level 1)
  User can list expected permissions as REQUIRED, MISSING, MAYBE_REQUIRED, MAYBE_MISSING or UNUSED and the ATE will tell whether the expected permissions exists in the Analysis result or not.

  For Information Flow (Intra App- Level 2a)
  During this level, user can expect path between `SOURCE` and `SINK`. ATE will further check existence of such a path with the help of `Flowpath` obtained from the Analysis result.

  For Permission Usage (Inter App- Level 2b)
  At this level, user can predict whether the specific permission has *Indirect*, *Direct* or *Both* access within the applications. To test permission groups (REQUIRED,MISSING etc.), user need to run $PermissionUsage(IntraApp-Level1)$.
- Exceptions Handling
  Apart from validating the expected results and ensuring consistency, ATE also catches and displays Exceptions if it encounters any during the execution of Test cases such as missing files or any Internal exception thrown by tool like unable to fetch analysis result.
- Generation of Analysis Result
  A successful generation of Analysis result will be reported to the user on screen right after the processing of Test case. In case, the analysis result is not generated, it will tell the user about that and will enter in state discussed above.

Now brief explanation of the above mentioned functionalities are as follows

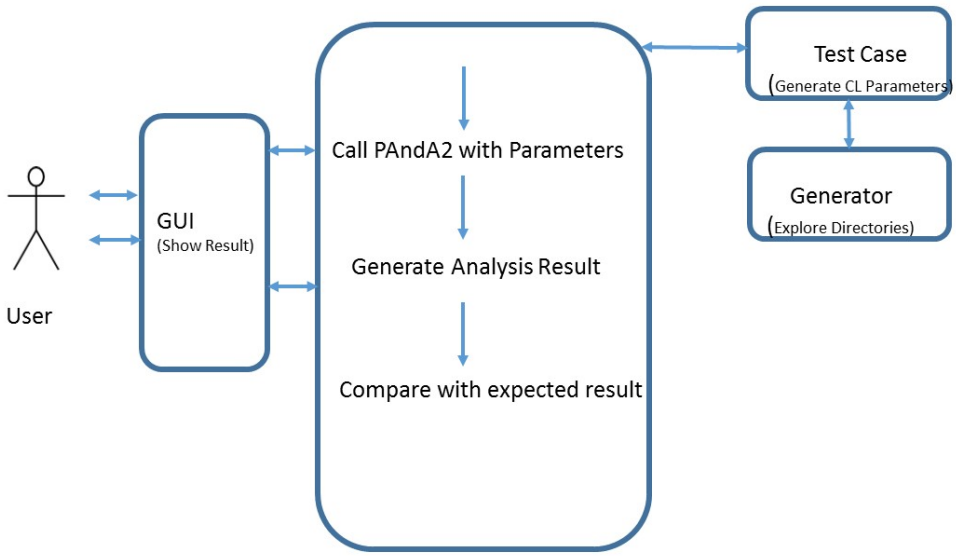

Figure 40: ATE WorkFlow

5.3.1 Workflow of Automatic Test Executor (ATE)

Automatic Test Executor (ATE) has a graphical user interface embedded in PAndA^2 . It can be started from `AutomatedTestGUI.java`, which when invoked run on the newly build source code of PAndA^2 automatically and requires path of the Test Cases. ATE can also be started from command prompt in Windows or terminal using Linux by using the following parameter

```
java -cp PAndA2.jar de.upb.pga3.ate.AutomatedTestGUI
```

As shown in Figure 40, ATE `Generator` will navigate through test cases directory (refer section 5.3.2) and ATE `Test Case` will create input parameters to start analysis in PAndA^2 via *CommandLine* Interface (refer section 3.1). Then it checks for the generation of Analysis result or exceptions, messages if encountered any. Later, if Expected results mode, ATE will compare the analysis result generated with the expected result of the user.

Once initiated, it will execute all the test cases and will display user, the following information in text field.

- Whether the tool PAndA^2 had successfully ran specific analysis.
- Whether any exceptions encountered.
- Whether the expected result matches the Analysis result.

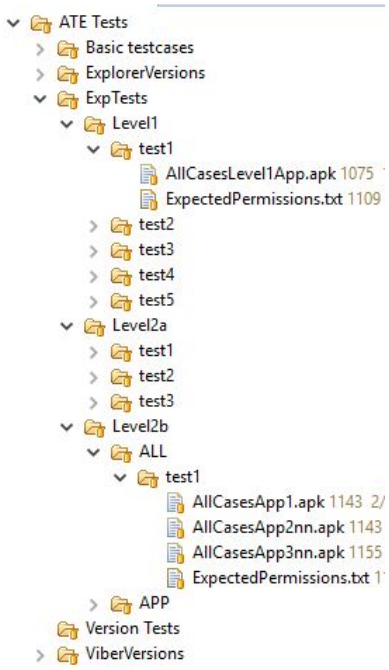

Figure 41: ATE Folder Structure

5.3.2 Structure of Test Cases:

To run the tool successfully, prior need to focus is the creation of Test cases. ATE has its predefined structure which covers all the possible functionalities available in PAndA[2] . Areas to consider are as follows

Specific hierarchical structure For the generation of Test cases directory in ATE, defined hierarchical structure is used as shown in Figure 41. It is basically grouping of test cases that belong to one level together making sets and predefined naming structure. We just need to give the address of Root node (e.g. *ExpTests* in Figure 41) as input parameter to ATE GUI and it will traverse all through the test cases of its own.

Protocols for Expected Result file We have discussed how the test cases should be placed in specific structure. In this section, a brief description of how the expected results be described along with Test cases is provided. Basically we need to write statements in `.txt` file and place them in the test case folder along with `.apk` file. For the different levels and different expectations, we need to append a *regex*, which will be decrypted by the ATE while processing. Example of which, categorized to different levels are as follows.

For Permission Usage (Intra App- Level 1) At this level, user can check for MISSING, REQUIRED, MAYBE_REQUIRED, UNUSED and MAYBE_MISSING permissions i.e. five permission groups in the application. Syntax with example is here under

- :M (MISSING)
- :R (REQUIRED)
- :O (MAYBE_REQUIRED)
- :U (UNUSED)
- :G (MAYBE_MISSING)

e.g- If we are expecting *android.permission.CAMERA* to be MISSING, we should create a text file and place in it the statement
android.permission.CAMERA:M

For Information Flow (Intra App- Level 2a) At this level, user can check for path between `SOURCE` and `SINK`. User need to provide two statements for this. Syntax with example is mentioned below

- :A (SOURCE)
- :B (SINK)

e.g-If we are expecting that *android.permission.ACCESS_LOCATION_EXTRA_COMMANDS* is a `SOURCE` permission and *android.permission.SEND_SMS* to be `SINK` permission, then we need to write two statements to text file, they are as
android.permission.ACCESS_LOCATION_EXTRA_COMMANDS:A
android.permission.SEND_SMS:B

For Permission Usage (Inter App- Level 2b) At this level, user can check for Direct, Indirect or Both access of permissions between the applications. Syntax with example is mentioned below

- :I (INDIRECT)
- :D (DIRECT)
- :B (BOTH)

e.g- If we are expecting the *Direct* access of *android.permission.CAMERA* between the `.apk` files, then the statement to put in text file is
android.permission.CAMERA:D

Definition of Test cases We have some possibilities of functionalities while running the test cases, in accordance we need to define the files inside the test cases. Basically there are three possibilities which are explained below, on top of it third possibility has its combination with first and second possibility. The brief description is as

Summary mode For specification of a test in *SUMMARY* mode, we need to provide at least one `.apk` file in our test folder. Tool will consider the test to be in Summary mode. For $PermissionUsage(IntraApp-Level1)$ and $InformationFlow(IntraApp-Level2a)$, only one `.apk` file is required while for $PermissionUsage(InterApp-Level2b)$, multiple `.apk` fiiles can be provided in input as it can contain one main application and other non native applications. Non native applications are passed with suffix `nn` (e.g. DemoAppnn.apk).

Comparison mode For the *COMPARISON* mode, two inputs are required in which one should be `.apk` file and another can be either `.pa2` file (which is previous saved Analysis result) or `.apk` file with suffix `comp` (e.g. DemoAppcomp.apk) at the same location inside test case folder.

Expected Result mode Now on the top of the above two modes, If we want our tool to perform an additional analysis for us i.e to validate our expected result in analysis result, we need an additional `.text` file in the test case directory in combination with the structure mentioned in either of the two cases mentioned above ie. an `.apk` file and `.pa2` file. This mode will be followed subsequently by the former ones.

5.3.3 Result Output:

The result or outcome of the tool can be seen in a GUI window which contains a text field that displays the outcome of every test cases that the tool has been through after each execution. It will update the user about its state or position by display field which keeps on updating itself as the tool goes.

It displays the test case counter as shown in 42, if the analysis result has been successfully generated or in case encountered any exceptions during the analysis and on top of it the validation of User expected result with the result obtained from Analysis. General overview of how the outcome will be like, with some of the Test cases result is shown in the Figure 42.

6 Evaluation

This chapter will cover the evaluation of our tool and especially the evaluation of the three types of analysis which are delivered with PAndA^2 . It will be evaluated if the analyses work as expected. Additionally the accuracy of each analysis will be stated. And furthermore we will compare our analyses to other state-of-the-art approaches. In order to do so three different sets of Apps are used:

- **Custom Apps**
  These are designed and implemented by the project group and used to turn out special properties of our analyses.

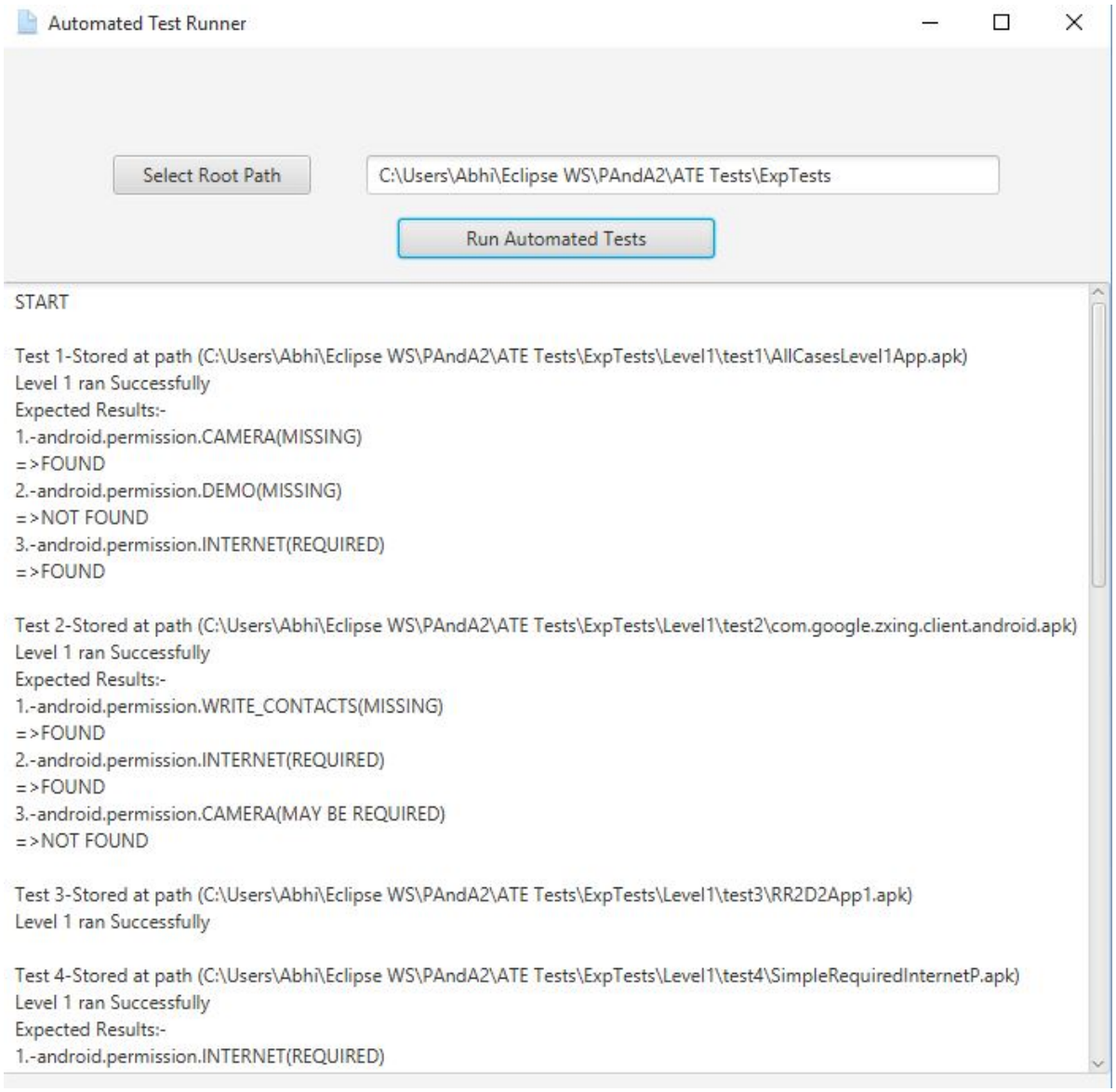

Figure 42: ATE Output

- **DroidBench**
  DroidBench[7] is a collection of Apps provided by the Secure Software Engineering Group. It is used for benchmarking Android taint-analyses based on Information Flow approaches.
- **Real-World Apps**
  All Apps in this set are downloaded from the Google Play Store[8] and listed below:
  - ADAC Pannenhilfe[9]
    App of a german breakdown service which has the purpose to speed up call in process.
  - Adobe Acrobat Reader[10]
    A basic PDF viewer that can be used to display PDFs from the device's internal storage or other sources.
  - Barcode Scanner[11]
    This App uses the camera to scan e.g. QR-Codes or barcodes.
  - ES File Explorer[12]
    A powerful file explorer that is able to access all possible file sources e.g. local storage or network resources.
  - Google Photos[13]
    Google's version of a gallery App. Its main task is to display the pictures stored on the device.
  - Instagram[14]
    This App supports all the functionality of the famous website instagram.com.
  - Tiny Flashlight[15]
    A very simple App that allows the user to use the photoflash of her/his device as a flashlight.
  - WhatsApp Messenger[16]
    This messenger App is currently the most downloaded App on the market. It is a complex messenger with a lot of extra functions.

[7] https://blogs.uni-paderborn.de/sse/tools/droidbench/
[8] Downloaded from Google Play Store on 5th March 2016
[9] https://play.google.com/store/apps/details?id=de.adac.mobile.pannenhilfe
[10] https://play.google.com/store/apps/details?id=com.adobe.reader
[11] https://play.google.com/store/apps/details?id=com.google.zxing.client.android
[12] https://play.google.com/store/apps/details?id=com.estrongs.android.pop
[13] https://play.google.com/store/apps/details?id=com.google.android.apps.photos
[14] https://play.google.com/store/apps/details?id=com.instagram.android
[15] https://play.google.com/store/apps/details?id=com.devuni.flashlight
[16] https://play.google.com/store/apps/details?id=com.whatsapp

The execution environment was a computer with a Intel i7 2600 (3.4 GHz) cpu and 8 GB memory (6 GB assigned to Java virtual machine).

6.1 Intra- and Inter-App Permission Usage Analysis

This section focuses on the evaluation of the Permission Usage Analyses. It will be pointed out, that both analyses work as intended. With more specific words, it will be shown that all Permission-Groups are assigned as expected. Furthermore it will be evaluated if possibilities of information leaks are successfully found and thereby the higher precision of the Inter-App Permission Usage Analysis will be pinpointed.

6.1.1 Custom Apps: Description

In this first scenario four Apps, developed by the project group, are used to show that both analyses work as expected. The Apps will not work on any Android device. They are only feasible for the evaluation. As a first step the four Apps themselves are going to be described. Then different analysis setups will be build based on these four Apps and the expected result of each setup will be described and compared to the actual results generated by PAndA[2] .

App 1: DirectApp The source code of this App can be found in Listing 10. This App is directly accessing the Camera (Line 7) and the Internet (Line 14). By that it will require two permissions, namely the *android.permission.CAMERA* and the *android.permission.INTERNET* permission. But only the *android.permission.CAMERA* permission along with the *android.permission.VIBRATE* permission, which is actually never used, is marked as being used by the App in it's Android manifest.

```
1  public class DirectActivity extends Activity {
2     @Override
3     protected void onCreate(final Bundle savedInstanceState) {
4        // Use android.permission.CAMERA
5        final Camera cam = (Camera) getApplicationContext().
6                           getSystemService(Context.CAMERA_SERVICE);
7        cam.open();
8
9        // Use android.permission.INTERNET
10       try {
11          final URL url = new URL("http://website.net");
12          final HttpURLConnection conn = (HttpURLConnection) url.
13                                          openConnection();
14          conn.connect();
15       } catch (final IOException e) {
16          Log.e("Error", e.getMessage());
17       }
```

```
18      }
19  }
```

Listing 10: Source code of the DirectApp App

App 2: IndirectApp Listing 11 represents the source code of this App. It does not access any permission directly but it uses an implicit Intent (lines 4-6) to call the previously described DirectApp App. Thereby it might use any permission through the called App. Nevertheless the *android.permission.CAMERA* permission is the only permission declared as being used in the manifest file.

```
1  public class IndirectActivity extends Activity {
2      @Override
3      protected void onCreate(final Bundle savedInstanceState) {
4          // Indirect Intent calling DirectApp
5          final Intent intent = new Intent("de.upb.pga3.CallDirectApp");
6          startActivity(intent);
7      }
8  }
```

Listing 11: Source code of the IndirectApp App

App 3: DirectAndIndirectApp The third App is called DirectAndIndirectApp. It's source code is printed in Listing 12. This third App combines the first two Apps. It shares the two statements which require the *android.permission.CAMERA* and *android.permission.INTERNET* permission with App 1 as well as the implicit Intent which was used in App 2. As before the only permission assigned as being used is the *android.permission.CAMERA* permission.

```
1  public class DirectAndIndirectActivity extends Activity {
2      @Override
3      protected void onCreate(final Bundle savedInstanceState) {
4          // Use android.permission.CAMERA
5          final Camera cam = (Camera) getApplicationContext().
6                                       getSystemService(Context.CAMERA_SERVICE);
7          cam.open();
8
9          // Use android.permission.INTERNET
10         try {
11             final URL url = new URL("http://website.net");
12             final HttpURLConnection conn = (HttpURLConnection) url.
13                                            openConnection();
14             conn.connect();
15         } catch (final IOException e) {
16             Log.e("Error", e.getMessage());
17         }
18
```

```
19          // Indirect Intent calling DirectApp
20          final Intent intent = new Intent("de.upb.pga3.CallDirectApp");
21          startActivity(intent);
22      }
23  }
```

Listing 12: Source code of the DirectAndIndirectApp App

App 4: MaliciousApp The last App's name is MaliciousApp (see Listing 13). This last App uses the *android.permission.VIBRATE* (Line 7). This is also the only permission that is marked as begin used in the App's manifest file. Furthermore it contains a valid implicit Intent that allows this App to call App Number 2 and 3.

```
1   public class MaliciousActivity extends Activity {
2       @Override
3       protected void onCreate(final Bundle savedInstanceState) {
4           // Use android.permission.VIBRATE
5           final Vibrator vib = (Vibrator) getApplicationContext().
6                                getSystemService(Context.VIBRATOR_SERVICE);
7           vib.vibrate(1000);
8
9           // Indirect Intent calling IndirectApp and DirectAndIndirectApp
10          final Intent intent = new Intent("de.upb.pga3.CallOtherApps");
11          startActivity(intent);
12      }
13  }
```

Listing 13: Source code of the MaliciousApp App

Summary Table 10 shows a summary of the properties of these four Apps. Additionally the

Table 10: Summary of Custom Apps

App	Actual Permission Uses	Permissions in Android Manifest	Implicit Intent Targets
1: DirectApp	CAMERA, INTERNET	CAMERA, VIBRATE	
2: IndirectApp		CAMERA	1: DirectApp
3: DirectAndIndirectApp	CAMERA, INTERNET	CAMERA	1: DirectApp
4: MaliciousApp	VIBRATE	VIBRATE	2: IndirectApp 3: DirectAndIndirectApp

Graph pictured in Figure 43 depicts the communication between these Apps. The nodes stand for the Apps and the purple, dotted arrows symbolize the implicit Intents.

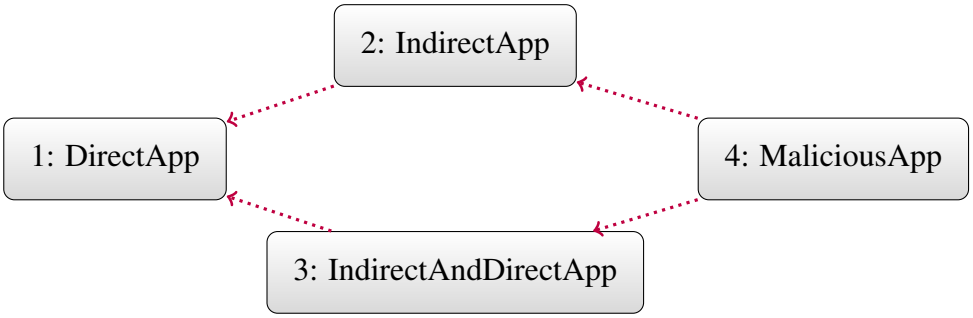

Figure 43: Communication graph

Table 11: Setup summary

Permission-Group assigned	**Setup**								
	1A	**1B**	**2A**	**2B**	**3**	**4**	**5A**	**5B**	**6**
REQUIRED (direct)	✓	✓							✓
REQUIRED (indirect)					✓				✓
REQUIRED (direct & indirect)						✓			✓
MAYBE_REQUIRED			✓	✓			✓	✓	
UNUSED	✓	✓							
MISSING (direct)	✓	✓							✓
MISSING (indirect)					✓			✓	✓
MISSING (direct & indirect)						✓			
MAYBE_MISSING			✓	✓			✓	✓	

6.1.2 Custom Apps: Evaluation

Based on the previously described Apps, different setups will be evaluated now. The goal is to check if all Permission-Groups are assigned correctly. Table 11 lists all available PGs and shows in which setup a specific PG is assigned. Another goal is to point out the higher precision of the Inter-App Permission Usage Analysis compared to the Intra-App Permission Usage Analysis. In all setups the considered detail level is APP. Furthermore all result screenshots which show 4 maybe missing permissions should show all unmentioned permissions as maybe missing but in order to keep it clear the screenshots are cut.

Setup 1A The first setup consists of an Intra-App Permission Usage Analysis that analyzes the DirectApp App. The expected result should contain the following permissions: *permission.android.CAMERA*, *permission.android.VIBRATE*, *permission.android.INTERNET*. The *permission.android.CAMERA* permission should belong to the REQUIRED PG, because it is

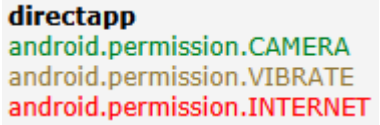

Figure 44: Result Setup 1A

assigned in the manifest and there exists an use of this permission in the App's source code. Permission *permission.android.VIBRATE* should be assigned to the UNUSED PG since there exists no use of this permission in the source code while it is still assigned in the manifest. Last but not least permission *permission.android.INTERNET* should be assigned to the MISSING PG, because there exists a use but it is not marked as being used in the manifest.
Figure 44 shows the result generated by PAndA^2 for this setup. Clearly it fits to the expected result.

Setup 1B is almost equal to Setup 1A but instead of an Intra-App Permission Usage Analysis a Inter-App Permission Usage Analysis is considered. The result generated by PAndA^2 is shown in Figure 45. It matches the expected result.

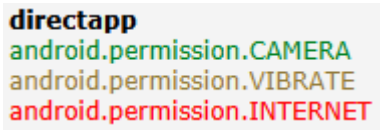

Figure 45: Result Setup 1B

Setup 2A The next setup consists of an Intra-App Permission Usage Analysis again that analyzes the IndirectApp App this time. The expected result should contain the *permission.android.CAMERA* permission in the MAYBE_REQUIRED PG, because the permission is marked as being used in the manifest but there only exists an implicit Intent and no statement which directly uses the permission. Because of the same reason, but with the difference that the following permissions are not mentioned in the manifest, the *permission.android.INTERNET* permission along with all other permissions should be assigned to the MAYBE_MISSING PG. In Figure 46 the generated result is pictured. It is identical with the expected result.

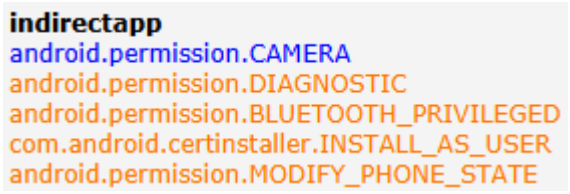

Figure 46: Result Setup 2A

Setup 2B is very similar to Setup 2A but this time the Inter-App case is considered. The result should be and is the same as before (see Figure 47).

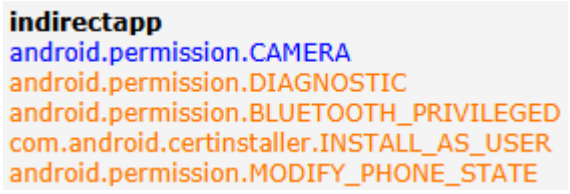

Figure 47: Result Setup 2B

Setup 3 The third setup is the first setup that involves more than one application. It is an Inter-App Permission Usage Analysis that analyzes the IndirectApp App. But along with this App the DirectApp App is provided as non-native, optional input. In contrast to Setup 2B the target of the implicit Intent defined in the analyzed App can be determined now. Thereby the expected result should contain the *permission.android.CAMERA* permission in the REQUIRED (indirect) PG and the *permission.android.INTERNET* permission in the MISSING (indirect) PG. All other permissions will remain in the MAYBE_MISSING PG. The results fits the expected result (see Figure 48).

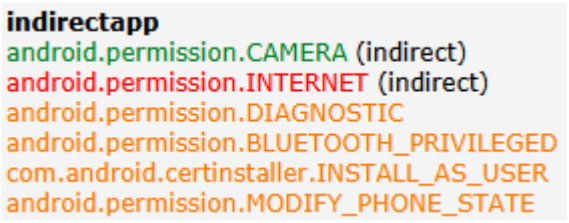

Figure 48: Result Setup 3

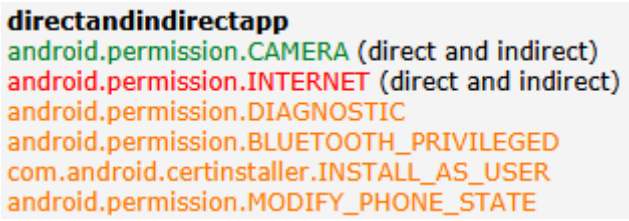

Figure 49: Result Setup 4

Setup 4 Setup 4 again is very similar to Setup 3. The only thing that changed is the analyzed App. Instead of analyzing the IndirectApp App, in this setup the DirectAndIndirectApp App will be analyzed. The result will slightly change through this modification. The *permission.android.CAMERA* permission will now be assigned to the REQUIRED (direct and indirect) PG and the *permission.android.INTERNET* permission to the MISSING (direct and indirect) PG. The visualized result in Figure 49 reveals that the result is equal to the expected one.

Setup 5A This setup again consists of an Inter-App Permission Usage Analysis but for the first time the mode will make a difference in the result. For this setup the chosen mode is APP. The Apps which are taken into account are, the IndirectApp App as analyzed App and the MaliciousApp App as non-native App. The expected result is equal

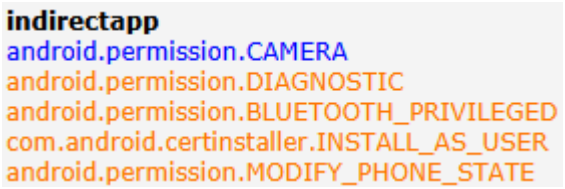

Figure 50: Result Setup 5A

to the expected and actual results of Setup 2A and 2B. Figure 50 shows the result which matches the expected one.

Setup 5B This setup is almost equal to the previous one but this time the mode is set to ALL. The difference that should come out in the result is that the permission *permission.android.VIBRATE* should be assigned to the MISSING (indirect) PG now. The reason is that the connection from the MaliciousApp App to the IndirectApp App is only considered in this mode. The actual result matches the expected one (see Figure 51).

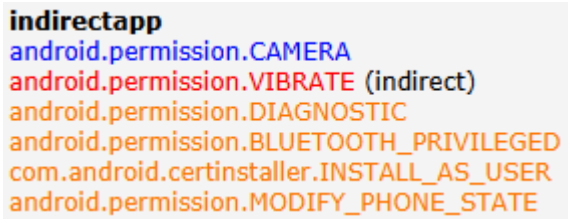

Figure 51: Result Setup 5B

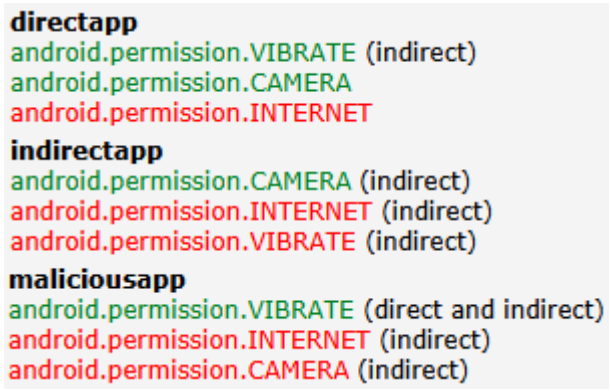

Figure 52: Result Setup 6

Setup 6 The last Setup, which is going to be considered, is again an Inter-App Permission Usage Analysis in ALL mode. This time three Apps will be considered: DirectApp, IndirectApp and MaliciousApp. The analysis will be run three times. Each time another App is setup as the analyzed App and the others are used as non-native Apps. The evaluated, expected result will show that the permissions *permission.android.CAMERA*, *permission.android.INTERNET* and *permission.android.VIBRATE* are spread across all three Apps. The Figure 52 shows the associated actual results. Clearly they fit to the expected results.

Its left to say that all the actual results match the expected ones. Both analyses are working as expected.

6.1.3 Real-World Apps: Evaluation

All these previously described and evaluated setups are used in order to check if PAndA^2 is working as expected. In the following it will be evaluated if the results acquired with the Intra- and Inter-App Permission Usage Analysis are accurate. To do so the real-world Apps described at the beginning of this chapter are used. In the following the trustworthiness of these Apps is evaluated.

Table 12: Summary of Real-World Apps Results

App	REQUIRED	MAYBE_REQUIRED	UNUSED	MAYBE_MISSING	MISSING	MISSING edited	R1	R2	R3
ADAC Pannenhilfe	3	5	0	165	2	0	0,05	0,60	*1,00*
Adobe Acrobat Reader	2	2	0	170	1	0	0,02	0,67	*1,00*
Barcode Scanner	6	3	0	162	4	2	0,05	0,60	0,75
ES File Explorer	10	9	0	151	5	3	0,11	0,67	0,77
Google Photos	15	5	0	143	12	6	0,11	0,56	0,71
Instagram	9	3	0	155	8	3	0,07	0,53	0,75
Tiny Flashlight	4	3	0	162	6	5	0,04	0,40	0,44
WhatsApp Messenger	25	7	0	140	3	1	*0,18*	*0,89*	0,96

Intra-App Permission Usage Analysis At first an Intra-App Permission Usage Analysis will be executed per real-world App. The results of all these analyses can be found in Table 12. The first column refers to the App analyzed. The next five columns stand for the number of permissions in the associated PGs. The sixth column with the title "MISSING edited" refers to the number of permissions in the MISSING PG without all permissions whose protection level is normal.

> "Normal" permissions should imply minor risk and serve only as a "heads-up" for the user that the application is requesting access to such functionality.[11]

The next three columns show different trustworthiness-rankings:

- **R1** simply shows the ratio of missing and maybe missing permissions to all permissions.

$$R1 = 1 - \frac{|MISSING| + |MAYBE\_MISSING|}{|ALL\_PERMISSIONS|}$$

- **R2** is equivalent to R1 but it ignores the maybe required and maybe missing permissions.

$$R2 = 1 - \frac{|MISSING|}{|REQUIRED| + |UNUSED| + |MISSING|}$$

- **R3** is equivalent to R2 but ignores missing normal permissions as well.

$$R3 = 1 - \frac{|MISSING\ edited|}{|REQUIRED| + |UNUSED| + |MISSING\ edited|}]$$

By these rankings we can assume how trustworthy an App is. A value of 1 stands for trustworthy while a value of 0 stands for not trustworthy at all. All three of these rankings show that the majority of Apps is not 100% trustworthy. Only the ADAC Pannenhilfe and the Adobe Acrobat Reader App could achieve a value of 1.00 at ranking R3. Nevertheless the WhatsApp Messenger App seems to be almost fully trustworthy according to the numbers of the rankings R2 and R3. Also in ranking R1 the WhatsApp Messenger App could reach the highest trustworthiness value. Regardless of all these numbers all of the evaluated Apps are missing at least one permission and by that none of them can be trusted blindly. The user still has to decide by himself if he trusts the App or waits until a new version of the App is released that might fix the issues. Anyway PAndA^2 could successfully provide sufficient information to support that decision.

Table 13: Execution times (seconds)

App	FlowDroid	Amandroid	PAndA^2 *
ADAC Pannenhilfe	7	112	13
Adobe Acrobat Reader		197	31
Barcode Scanner	14	23	17
Google Photos		1062	138
Instagram		4246	83
Tiny Flashlight		1328	28
WhatsApp Messenger			12330

* Execution time of PAndA^2 executing an Intra-App Permission Usage Analysis.

Inter-App Permission Usage Analysis The last case that is going to be evaluated in this section consist of an Inter-App Permission Usage Analysis (ALL mode) of all described real-world Apps. The selected analyzed App is the WhatsApp Messenger App. All other Apps are used as non-native Apps. They represent the App environment on an arbitrary Android device. The result of this analysis can be viewed in Figure 53 It shows that the App is accessing some permissions in a direct and indirect way. It looks as if the Inter-App Permission Usage Analysis could not provide a result with a higher accuracy this time. But with more detailed background knowledge some users might know that the missing permission *com.google.android.c2dm.permission.SEND* for example is only used together with an implicit Intent. And by this analysis result we can tell that there exists no App in the current environment that can be targeted by this Intent. Hence this result can be considered as being more precise.

After evaluating the expected functionality and the higher precision this evaluation section will be concluded with some remarks regarding the performance of the Permission Usage Analyses. Since we could not obtain or find another tool that is performing at least a similar analysis we could not compare the execution time of our analyses. But still we can prove on numbers that these analyses are quiet fast compared to other state-of-the-art information flow analyses. The Table 13 shows the execution times of our Intra-App Permission Usage Analysis compared to the execution times of the competing tools FlowDroid and Amandroid which will be described in more detail in the next chapter (see 6.2). On average the Intra-App Permission Usage Analysis is ~ 22 times faster then Amandroid. Of course the reason is the lower accuracy of this analysis but in many usecases a Permission Usage Analysis can be sufficient. Because of some special properties of FlowDroid, FlowDroid can even be faster. But on the other hand most of the analyses executed with FlowDroid did not finish at all and threw an exception. In case of the WhatsApp Messenger App PAndA^2 was the only tool that finished at all.
All in all the Permission Usage Analyses can be a reliable and fast alternative to the mostly slower and more complex information flow analyses.

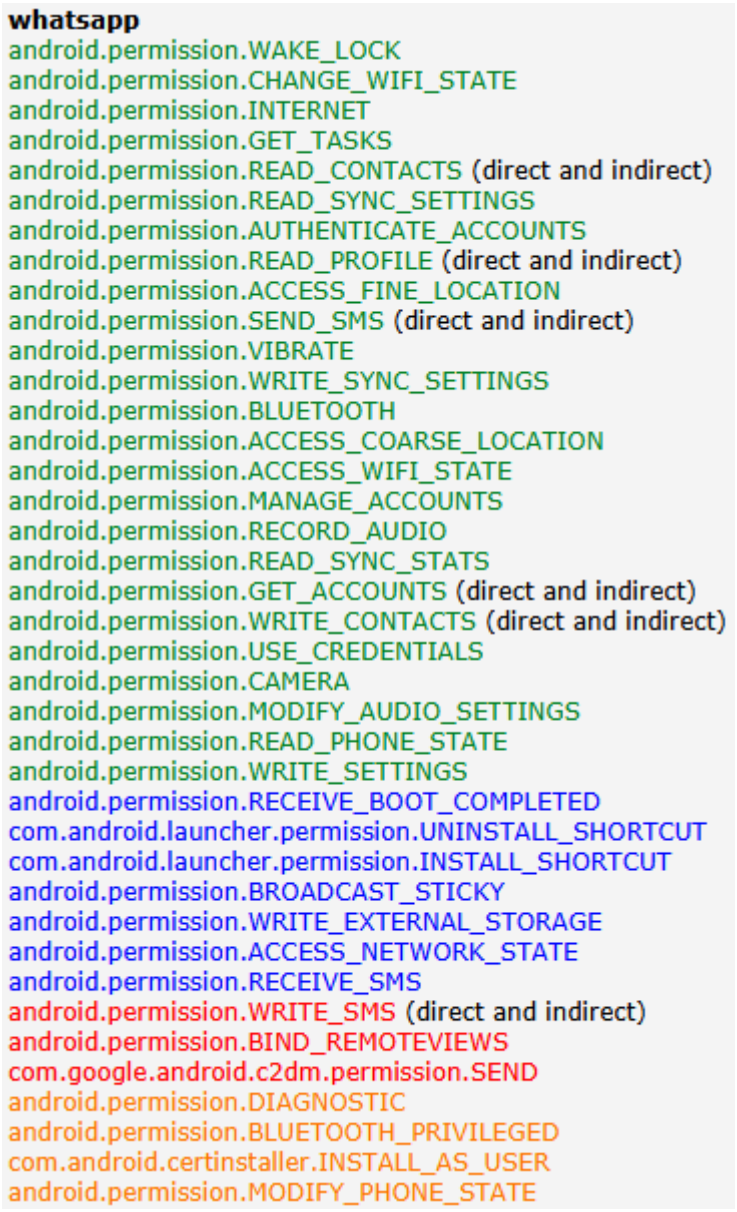

Figure 53: Result WhatsApp Messenger

6.2 Intra-App Information Flow Analysis

In this section we want to evaluate the performance of our Intra-App Information Flow Analysis regarding the ability of detecting information flows. Therefore, we compare our analysis with the tools FlowDroid And Amandroid which perform the same kind of analysis.

Although all three tools have the same goal, they take different approaches. Therefore, we had to clear some hurdles in order to be able to compare the three tools. Since Amandroid and FlowDroid do not support different detail levels as we do, we only used our detail level RES_TO_RES.

Another major problem is that all tools use different definitions of sources and sinks. This definition is important since information flows are only computed from a source to a sink. For example, if a tool does not consider a statement as a source by definition, it will not show an information flow in the result even if there exists a flow from that statement. If another tool does consider the mentioned statement as a source then it will be able to detect the information flow. To get an idea of whether those differences in definitions cause the change in performance of information flow detection or not, we will additionally evaluate the detection of sources and sinks for the different tools.

Along with the previous problem comes the fact that our tool considers permissions as sources and sinks and not protected statements themselves. For statements which are protected by more than one permission this means that there are more than one sources resp. sinks found in our tool whereas the other tools consider such a statement as one sink independently of the number of permissions it is protected by.

In addition, a property of FlowDroid is to perform the analysis on each source file independently. Therefore, information flows that we consider as a single path are split up in the result of FlowDroid. For the evaluation, we will count those split paths as one for not disadvantaging FlowDroid in our evaluation.

Before we start with the evaluation of the detection performance regarding information flow, we first define some naming conventions to be able to specify our evaluation measures. We call an information flow that was detected by the tools a *positive*. If an information flow was detected correctly, we call this a *true positive*. Otherwise, if a non-existent information flow was detected mistakenly, we call this a *false positive*. The last case is that a existent information flow was not detected by the tools. We call this then a *false negative*.

Built on that, we can describe the evaluation measures for the tools. We use the measures precision, recall and F-measure which are common for evaluating detection-related experiments. The definitions of these measures are given below:

Precision The precision describes the efficiency of detection. It is defined as the ratio of the number of *true positives* TP to the number of all *positives* which is the sum of occurrences of *true positives* TP and *false positives* FP: $precision = \frac{TP}{TP+FP}$

Recall The recall describes the sensitivity of detection. It is defined as the ratio of the number of *true positives* TP to the number of all existent information flows which is the sum of occurrences of *true positives* TP and *false negatives* FN: $recall = \frac{TP}{TP+FN}$

F-measure The F-measure combines precision and recall with the harmonic mean: $f = 2 \cdot \frac{precision \cdot recall}{precision + recall}$

In the following, we evaluate the three tools FlowDroid, Amandroid and PAndA[2] regarding their ability of detecting information flows. We do this first on a set of custom Apps that we have created for this evaluation in Section 6.2.1. Afterwards we perform the evaluation on a set of Apps provided by the DroidBench benchmark in Section 6.2.2 and last on a set of real-world Apps in Section 6.2.3.

6.2.1 Custom Apps

For evaluation we created a small set of Apps. We defined four test cases which we wanted to test the tools with. For each test case we created a single App. The first two Apps contain information flow only within a single Android component. Most of the sources and sinks are not accessed directly by the Android component but are wrapped by normal Java classes. The other two Apps cover inter-component communication. In one case the sensitive information flows via intent to another component. In the other case the sensitive information is looped through a second component and flows back to the sender component. All Apps are described in detail below.

App 1: ClassToClass The application `TestCase3` has one Android Activity class and two regular Java classes. First Java class has a statement protected by the permission *android.permission.READ_PHONE_STATE*. This will be the source. In the second Java class, we have a statement protected by two permissions *android.permission.SEND_SMS* and *android.permission.WRITE_SMS*. This will be the sink. The Android Activity class instantiates both of the Java classes, obtains sensitive data from the source in the first Java class and passes it to the sink in the second Java class. There is one actual information flow path in this app.

App 2: ComponentAndClassToClass This App has a source statement in one normal Java class. The source is protected by permission *android.permission.READ_PHONE_STATE*. We have two sink statements, one in a Java class and another in a Android Activity class. Each sink statement is protected by the two permissions *android.permission.SEND_SMS* and *android.permission.WRITE_SMS*. The Activity instantiates both Java classes, obtains sensitive data from source in the first Java class and passes the data to one sink in the second Java class and another sink within the Activity. Now there are two distinct information flow paths in this Android application.

App 3: ExplicitIntentApp App 3 has two Activity classes. One normal Java class has a source statement, protected by the permission *android.permission.READ_PHONE_STATE*. Another Java class has a sink statement protected by the permissions *android.permission.SEND_SMS*

Tool	TP	FP	FN	Precision	Recall	F-measure
PAndA^2	5	1	0	0.83	1	0.9
Amandroid	1	0	4	1	0.2	0.3
FlowDroid	5	2	0	0.71	1	0.83

Table 14: Precision, Recall and F-measure for the found paths in custom Apps

Tool	TP	FP	FN	Precision	Recall	F-measure
PAndA^2	4	0	0	1	1	1
Amandroid	1	3	3	0.25	0.25	0.25
FlowDroid	4	20	0	0.17	1	0.28

Table 15: Precision, Recall and F-measure for the found sources in custom Apps

and *android.permission.WRITE_SMS*. The first Activity instantiates the Java class that contains the source statement and obtains sensitive data from the source. The data is added to an explicit intent and the second Activity is started. The second Activity instantiates the Java class with the sink and passes the data to the sink. Here we have one information flow path from a source protected by a permission to a sink through explicit intent between two Android Activity classes.

App 4: IntentResultApp This application has two Android Activity classes. The first Activity class has a source statement protected by the permission *android.permission.READ_PHONE_-STATE*. The Activity class instantiates a Java class containing a source and sensitive data from the source is obtained. The data is then added to an intent and the second Activity is started for result. On receiving the intent, the second Activity sends the received data back to the sender. Within the `onActivityResult()` method in the first Activity, we have a sink statement protected by the permissions *android.permission.SEND_SMS* and *android.permission.WRITE_-SMS*. We have one information flow path from source in the first Activity to the second Activity and then back to the first Activity through explicit intents.

For performing the evaluation, we applied the adaptions to the actual results that we described in Section 6.2. Afterwards we calculated Precision, Recall and F-measure independently for found paths, sources and sinks. The results are shown in Table 6.2.1 for paths, in Table 6.2.1 for sources and in Table 6.2.1 for sinks.

The worst performance was achieved by Amandroid. For paths it reached only a Recall of 0.2. This is caused by the different definition of sinks and sources, because many many of the defined sources in the Apps were not found by Amandroid. PAndA^2 and FlowDroid performed much better. Both reached an F-measure of over 0.8 in detecting information flows. Here PAndA^2 performed slightly better. Where FlowDroid performed really worse is in detecting sources. It often finds way too many sources, which lowers the Precision remarkably. Unfortunately we did not find a reason for this behavior of FlowDroid.

Tool	TP	FP	FN	Precision	Recall	F-measure
PAndA^2	5	1	0	0.83	1	0.9
Amandroid	1	1	4	0.5	0.2	0.28
FlowDroid	5	3	0	0.63	1	0.77

Table 16: Precision, Recall and F-measure for the found sinks in custom Apps

All in all, the evaluation on the custom Apps went well for our tool and for FlowDroid. However, the amount of four Apps is not representative. As the next step, we evaluate the tools on a larger amount of Apps that are provided by the DroidBench benchmark.

6.2.2 DroidBench

In addition to evaluating our tool with a set of custom Apps we used the benchmark DroidBench as already described in Section 6. DroidBench is a collection of applications that describe test cases especially for tools that track information flow through Android applications. The source code as well as a short description for every application is available so we were able to check for every App which should be the result of the analysis. DroidBench 2.0 contains 120 test Apps of which we selected nearly fifty. Since many of the 120 applications describe special cases that we decided not to handle in our analysis (see the Target Level Agreement) it was not reasonable to use the complete benchmark. One example of such not handled features is Inter-App communication.

As already mentioned we executed the test cases with our PAndA^2 tool as well as with Amandroid and FlowDroid. For each test case we collected the expected result and the outputs of the three tools. For the Intra-App Information Flow Analysis we measured, as for the own apps, the expected result in terms of number of sources, number of sinks and number of available paths from source to sink. Furthermore, we listed the execution time of the different tools for each App.

The next step then was to compare the expected and actual results. There we had to clear some hurdles in order to be able to compare the three tools. These were the problem concerning the definition of sources and sinks as well as dealing correctly with the split paths generated by FlowDroid. Mainly these were already described in Section 6.2. But besides there was another problem which occurred in particular when executing the DroidBench applications. Many of the DroidBench test cases use a logging statement as sink. Since our definition of sources and sinks did not consider logging as a sink, we had do adapt the Intra-App Information Flow Analysis such that it also takes logging into account. This was necessary because more than half of the test cases used logging as a sink and therefore adapting our tool to get a comparable result was more reasonable than selecting the test applications according to their sink statements.

After having done the just described adjustments we were able to compare the results of PAndA^2 with FlowDroid and Amandroid. The used metrics are precision and recall as well as

Tool	TP	FP	FN	Precision	Recall	F-Measure
PAndA^2	8	2	41	0,8	0,16	0,27
Amandroid	15	2	34	0,88	0,31	0,45
FlowDroid	39	9	10	0,81	0,8	0,8

Table 17: Precision, Recall and F-measure for the found paths

the F-measure.

In case of the Intra-App Information Flow Analysis precision describes how well a tool behaves in terms of how many of the found paths are real paths in the application. To compute this measure the true positives and false positives are taken into account. Table 6.2.2 shows the number of true positives (TP), false positives (FP) as well as the value of precision for the three tools. This value is quite high for all three of them which shows that PAndA^2 is able to keep up with the other tools regarding the question whether a found path is a real path in the application. Moreover, the two false positives listed in the table result from special cases where special information from the manifest has to be taken into account to realize that e.g. an activity is inactive and therefore no information is flowing through it.

Recall describes for the Intra-App Information Flow Analysis how many of the possible paths are found with the tool. This metric takes the true positives as well as the false negatives into account. Here the value computed for the results of the DroidBench applications is quite low. The main reason for this is the fact that DroidBench describes almost in every of the used applications special cases of information flow through an application. Many of these are excluded in the current version of the Intra-App Information Flow Analysis but could be integrated as future work (see Section 8).

The weighted harmonic mean, which is described through the F-measure, is a combination of the two metrics precision and recall and since the recall of PAndA^2 is quite low (as just described) it is justifiable that the F-measure is lower that those of the other tools as well.

The Tables 6.2.2 and 6.2.2 show the just described metrics for the number of found sources and found sinks. Here PAndA^2 is the only tool which in both cases has a precision of 1 which means that there is no false positive in the sources resp. sinks found. Moreover, for the found sources and sinks PAndA^2 even has the best F-measure of all three tools.

Finally, it can be said that the computation of sources and sinks works already very good whereas the computation of paths between these sources and sinks still could be refined.

In summary it can be said, that even though PAndA^2 has not the best results concerning recall and the F-measure in case of found paths this can be mainly traced back to the fact that DroidBench often deals with quite rare artificial cases. Much more important that handling all these special cases is a good behavior on real-world Apps. How the Intra-App Information Flow Analysis of PAndA^2 behaves in such cases follows now.

Tool	TP	FP	FN	Precision	Recall	F-Measure
PAndA^2	36	0	16	1	0,69	0,82
Amandroid	24	0	28	1	0,46	0,63
FlowDroid	49	108	3	0,31	0,94	0,47

Table 18: Precision, Recall and F-measure for the found sources

Tool	TP	FP	FN	Precision	Recall	F-Measure
PAndA^2	69	0	3	1	0,96	0,98
Amandroid	47	33	25	0,59	0,65	0,62
FlowDroid	69	16	3	0,81	0,96	0,88

Table 19: Precision, Recall and F-measure for the found sinks

6.2.3 Real-World Apps

Besides the custom Apps and the benchmark DroidBench we executed PAndA^2 with a set of real-world Apps as mentioned in Section 6. Since the source code of these applications is not known to us, the goal for the evaluation of executing the Intra-App Information Flow Analysis with these Apps was not to check for a correct result. Moreover, we wanted to prove that, besides finding information flow in constructed examples, this analysis can deal with real-world applications. Here again we compared the behavior of our analysis with the behavior of the tools FlowDroid and Amandroid.

The result which is visualized in Table 6.2.3 was quite surprising. The dashes express that the analysis could not be executed due to an exception. FlowDroid, which was quite good in finding many different cases of information flow (see previous section) managed only to execute a complete analysis on two of the eight examples. These were the ADAC Pannenhilfe and the Barcode Scanner.

Amandroid was better concerning the number of applications for which the analysis succeeded. This tool could analyze four out of eight applications. But the best of the three tools was PAndA^2 with six out of eight Apps. Another noticeable fact is that the number of sources and sinks found by the tools differs so much. There is no result where this number is equal for all tools. A reason for this behavior is that the definition of sources and sinks is different in the tools. When looking for example at the application Instagram the number of sinks found in PAndA^2 and Amandroid is extremely divergent. But here the 156 sinks found in Amandroid reflect statements where information is added to an intent. In contrast to that, PAndA^2 does not consider such statements as sinks (they might be part of a possible information flow path but not a sink) but concentrates on statements protected by permissions. Therefore, it is not surprising that the results considering sources and sinks but also considering paths differ that much.

	PAndA^2			FlowDroid			Amandroid		
App	**Sources**	**Sinks**	**Paths**	**Sources**	**Sinks**	**Paths**	**Sources**	**Sinks**	**Paths**
ADAC	1	0	0	84	29	10	0	2	0
Adobe	1	1	0	-	-	-	-	-	-
Barcode	2	2	0	0	0	0	0	0	0
ES File Explorer	-	-	-	-	-	-	-	-	-
Google Photos	8	7	0	-	-	-	-	-	-
Instagram	8	2	0	-	-	-	1	156	0
Flashlight	4	1	0	-	-	-	6	3	0
Whatsapp	-	-	-	-	-	-	-	-	-

Table 20: Evaluation Results for Real-World Apps

Table 21: Execution times (seconds)

App	**FlowDroid**	**Amandroid**	**PAndA^2 ***
ADAC Pannenhilfe	7	112	18
Adobe Acrobat Reader		197	68
Barcode Scanner	14	23	26
Google Photos		1062	1238
Instagram		4246	537
Tiny Flashlight		1328	76

* Execution time of PAndA^2 executing an Intra-App Information Flow Analysis.

The execution times of the three tools are shown in Table 21. Since FlowDroid successfully executed the analysis only for two of the Apps, the comparison with this tool is not very expressive but what can be seen in the table is that the execution times of PAndA^2 can keep pace with those of the other tools. There are some applications where the others are faster (Barcode Scanner, Google Photos) but there are also some where PAndA^2 is faster (Adobe Acrobat Reader, Instagram, Tiny Flashlight) than the others.

6.3 Feature Comparison

Besides the difference and similarity in analysis when comparing the PAndA^2 tool to the FlowDroid or Amandroid tool, there are also more advanced features in the PAndA^2 tool. In this section we will describe the extra features of analysis modes and user interfaces.

Regarding the analysis modes, the PAndA^2 tool can analyze a single Android application to extract the desired information accordingly to the type of the analysis. This feature is same as in the FlowDroid or Amandroid. However, the PAndA^2 tool also supports an aggregation analysis which allows users analyze an Android application within an existing environment. The environment is indeed created by many other Android applications which may or not influence the analyzed application. This type of analysis is performed by the Inter-App Permission Usage Analysis in the PAndA^2 tool. In addition, users can find more information about the difference between two versions of the same application by using the COMPARISON mode. It means that the tool is able to analyze multiple Apps and then if needed, they can compared with previous results to obtain the similar or the different information.

With respect to the user interfaces, one of the concerns in the PAndA^2 tool is the ease of use. The PAndA^2 tool was designed not only for extensibility and maintainability but also the ease of use. Similar to the both tools - FlowDroid and Amandroid - the PAndA^2 tool is built into a JAR file to allow users running analysis by command line. In general, users can only get textual results through the command line interface. However, the PAndA^2 tool is more advanced than the others because it supports GUI. The GUI lets users perform exactly same analysis as in command line interface, but the result representation is more readable and understandable. In particular, the GUI can represent results in textual and graphical mode. This can help users to easily get the information of results in textual mode or have a clear view about them in graphical one. Furthermore, all interactions such as selecting operations, filtering results etc. from users to the PAndA^2 tool become more comfortable when comparing the GUI with the command line interface.

Last but not least, the results obtained in PAndA^2 , can be reused whenever required. Currently in the FlowDroid or Amandroid tool, users can only view the result of each analysis once. If they want to view the result again, they have to rerun the analysis. In contrast, the PAndA^2 tool allows users saving results into files. Therefore users can review it anytime they want without performing analysis again. In addition, the saved result is not only for reviewing,

but it can also be used as inputs for COMPARISON mode in the PAndA^2 tool. This increases the performance of the PAndA^2 tool since it does not need to run the analysis again.

In summary, with the advanced features for both analysis modes and user interfaces, the PAndA^2 tool tries to show precisely and correctly more information in a common way of command line and in the most efficient way of GUI to the users.

7 Future Work

In Section 3 we described the as-is state of our PAndA^2 tool and in Section 4 of the PAndA^2 framework. Both have been developed to the best of our knowledge and belief. However, we could not realize a fully comprehensive piece of software due to time constraints. In the following, we describe three enhancements that we would like to apply to our work in the future.

7.1 Improving Existing Analyses

The evaluation in Section 6 for the Intra-App and the Inter-App Permission Usage Analysis already led to a positive outcome. Nevertheless, we know that we did not cover some special cases related to the permission usage. For example it is possible to grant temporary access to resources without the need of permissions at run-time instead of defining the grants in the manifest file. For now, we cannot detect this case among some other run-time related cases and, hence, our analysis might show a wrong result. To include these special cases we need to adapt the core service `StatementAnalyzer` and the `Enhancer`. However, the realization would cost huge effort and the improvement of the analysis performance would only be minor.

In contrast to the Permission Usage Analyses, the evaluation of the Intra-App Information Flow Analysis has revealed some issues compared to the tools FlowDroid and Amandroid. To be precise, the performance of our tool was significantly worse than the other tools on the DoidBench Apps. This is majorly caused by the lack of covering information flow through global variables. Implementing this feature would only mean a small change to the Program Dependence Graph (PDG). Therefore, we are confident that this improvement can be realized in a small amount of time. Other useful but more heavy improvements are the support of object sensitivity and a partial support of concurrence sensitivity. Since our analysis is based on a PDG, we can profit from existing work. Object sensitivity in PDGs was already described in detail by Hammer in [4]. His work on this topic can be implemented one to one as an extension of the PDG and backward slicing algorithm. To support concurrence sensitivity, we can use the work of Krinke [8] as a basis. He describes concurrence sensitivity for general Java programs. The Android execution model is much more complex than the execution of a general concurrent Java program. Hence, the work by Krinke might has to be adapted to the Android environment and it might not be possible to cover all cases of concurrency. When all improvements are

implemented, we are confident that our tool can keep up with the performance of FlowDroid and Amandroid.

7.2 Extending the Set of Analyses

Within our PAndA^2 tool, we provide the Inter-App Permission Usage Analysis that we described in Section 3.5. A useful extension of our tool would be to provide also an Inter-App Information Flow Analysis that is based on the existing Intra-App Information Flow Analysis. Due to the architecture of our framework the new analysis can be realized with small effort since we can reuse the results of the Intra-App Information Flow Analysis.

With the new analysis we would be able to detect information flow through multiple Android Apps. For illustration we can distribute the information flow in our example App SimToSms over two Apps. We call the resulting Apps SimToIntent and IntentToSms. The App SimToIntent transfers the sim serial number via an implicit intent to the IntentToSms App which sends the received information via SMS. Obviously, it is not possible to detect the described information flow between resources with an Intra-App analysis. Hence, the new Inter-App Information Flow Analysis would enrich the features of the PAndA^2 tool remarkably.

7.3 Improving the Framework

In our Architecture Document, we defined a list of `AnalysisProcedures` as a model for an Android App Analysis. This list will be processed by the `AnalysisRunner` to perform the analysis. Such a list is sufficient for our kind of analyses. But regarding the framework which should be able to run arbitrary analyses, using a simple list is restrictive. Therefore, it is appropriate to use a more flexible data structure.

Our idea is to use a dependence graph containing the sub-analyses. A dependence graph is a directed acyclic graph with a single root node which is the aggregation analysis that creates the final result. All other nodes in the dependence graph are preceding sub-analyses that provide their results indirectly or directly to the final aggregation analyses. The leaf nodes of the graph will be initial analyses that process the `.apk` files to be analyzed.

In Figure 54 we show an example of what a dependence graph could look like. The example consists of six `AnalysisProcedures` named $A_1, \ldots, A_6$. The leaf nodes A_4, A_5 and A_6 on the lowest level perform an initial analysis on one `.apk` file each. The top `AnalysisProcedure` A_1 is the final aggregation analysis that will create the concluding result. The nodes A_2 and A_3 are

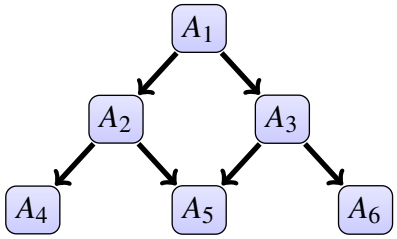

Figure 54: Dependence graph with six `AnalysisProcedures` $A_1, \ldots, A_6$ representing an Android App Analysis

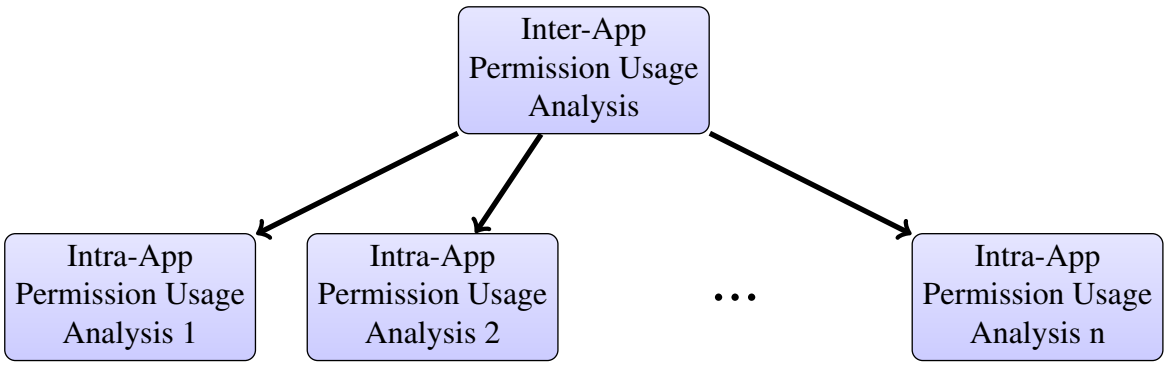

Figure 55: Dependence graph of a Inter-App Permission Usage Analysis

intermediate aggregation analysis that serve the final analysis step. All transitions in the dependence graph point downwards and represent the "depends on" relation.

In case the dependence graph for analysis is realized, we would have to transform our analyses from the list approach into such a graph. In Figure 55 we have visualized the dependence graph of our Inter-App Permission Usage Analysis that we described in Section 3.5. It has two levels: The lower layer consists of multiple Intra-App Permission Usage Analyses, one for each App. The top layer is the aggregation analysis that combines the preceding Intra-App Analyses into the final Inter-App Permission Usage Analysis.

After the dependence graph is implemented, we can additionally provide the option to run the analyses of a level in the dependence graph in parallel. This can speed up analyses significantly, especially for real-world Apps. Unfortunately, the parallelization leads to higher memory consumption. During the evaluation of our tool in Section 6 we observed already demanding memory consumption of analyses on single real-world Apps. Regarding this fact, we would have to use machines equipped with a high amount of memory to profit from the concurrent execution. Furthermore, the Soot framework, which our tool is based on, does not allow the decompilation of `.apk` files in parallel. Hence, we would need to find a workaround for this or we have to restrict the access to Soot within the PAndA^2 framework to be serial only. The second approach would reduce the parallel performance of our framework.

8 Conclusion

The previous sections described PAndA^2 and in particular which algorithms and concepts are implemented in the different components. The three analyses were described and possible ways of extending the tool were introduced. Furthermore, the quality assurance process PAndA^2 passed through was shown and the evaluation of the tool was described. Besides this we suggested ideas how PAndA^2 can be further extended and optimized.

In conclusion, we introduced a tool which is able to perform different analyses on Android applications and even provided three analyses implemented by ourselves. The evaluation showed that the Intra-App Information Flow Analysis has problems in finding certain constructs that can forward information. But even though the range of handled cases is not complete the analysis works on a good basis which can be extended by further features to handle more concepts. Besides this, we provide the two Permission Usage Analyses which perform a type of analysis which was not yet available in this form on the market. We saw that the Permission Usage Analyses can detect possible information leaks in an App and even through different Apps. In summary we saw that in some areas our tool has some weaknesses but there are other areas where our tool performs better than others. Therefore, it can be said that our analyses can keep up with other tools doing comparable analyses. Moreover, we provide with our Intra and Inter-App Permission Usage Analysis additional new features.

Besides the three analyses PAndA^2 provides the opportunity to use the framework and especially the client and core services with other analyses that can be easily integrated into PAndA^2 . Concluding the document, it can be said that with PAndA^2 we provide the possibility to users (thanks to the graphical user interface even to inexperienced users) to find possible information leaks in Android applications.

Appendix

```
1  public class SimReaderActivity extends Activity {
2     private static final int MY_REQEST_CODE = 1;
3     public static final String SIM_DATA = "SIM_DATA";
4     String serviceNumber;
5
6     @Override
7     protected void onCreate(Bundle savedInstanceState) {
8        TelephonyManager manager = (TelephonyManager)
9           getSystemService(Context.TELEPHONY_SERVICE);
10       String simSerialNumber = manager.getSimSerialNumber();
11
12       simSerialNumber = shortenSim(simSerialNumber, 5);
13
14       Intent explicitIntent = new Intent(this, SmsSenderActivity.class);
15       explicitIntent.putExtra(SIM_DATA, simSerialNumber);
16       startActivityForResult(explicitIntent, MY_REQEST_CODE);
17    }
18
19    @Override
20    protected void onActivityResult(int reqCode, int resultCode,
21       Intent resultIntent) {
22       if(reqCode == MY_REQEST_CODE) {
23          if(resultCode == Activity.RESULT_OK) {
24             serviceNumber = resultIntent
25                .getStringExtra(SmsSenderActivity.SERVICE_NUMBER_DATA);
26
27             Intent implicitIntent = new Intent("de.upb.pga3.sendData");
28             implicitIntent.putExtra(SmsSenderActivity.SERVICE_NUMBER_DATA,
29                serviceNumber);
30             startActivity(implicitIntent);
31          }
32       }
33    }
34
35     private String shortenSim(String simSerialNumber, int length) {
36         if(simSerialNumber.length() > length) {
37             simSerialNumber = simSerialNumber.substring(0, 4);
38         }
39         return simSerialNumber;
40     }
41 }
42
43
44 public class SmsSenderActivity extends Activity {
45    public static final String SERVICE_NUMBER_DATA = "RECEIVER_NUMBER";
46    private static final String SERVICE_NUMBER = "+49123456789";
47
48    @Override
```

```
49      protected void onCreate(Bundle savedInstanceState) {
50          Intent receivedIntent = getIntent();
51          String simSerialNumber = receivedIntent
52              .getStringExtra(SimReaderActivity.SIM_DATA);
53          SmsManager.getDefault().sendTextMessage(SERVICE_NUMBER, null,
54              simSerialNumber, null, null);
55          Intent resultIntent = new Intent();
56          resultIntent.putExtra(SERVICE_NUMBER_DATA, SERVICE_NUMBER);
57          setResult(RESULT_OK, resultIntent);
58          finish();
59      }
60  }
```

Listing 14: Source code of the SimToSms App

List of Figures

References

[1] Tom Copeland. Pmd 4.0. `http://tomcopeland.blogs.com/juniordeveloper/2007/07/pmd-40-released.html`, July 2007, (accessed February 25, 2016). 71, 73

[2] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987. 49

[3] Inc Google. Codepro analytix evaluation guide. `https://google-web-toolkit.googlecode.com/files/CodePro-EvalGuide.pdf`, 2006 - 2010, (accessed February 25, 2016). 74

[4] Christian Hammer. *Information Flow Control for Java - A Comprehensive Approach based on Path Conditions in Dependence Graphs*. PhD thesis, Universität Karlsruhe (TH), Fak. f. Informatik, July 2009. ISBN 978-3-86644-398-3. 50, 52, 55, 56, 101

[5] Rick Hower. Software qa and testing resource center. `http://www.softwareqatest.com`, 1996 - 2016, (accessed February 22, 2016). 63, 67, 71

[6] Zhen Huang Kathy Wain Yee Au, Yi Fan Zhou and David Lie. *PScout:Analyzing the Android Permission Specification*. PhD thesis, University of Toronto, Department of Electrical and Computer Engineering, October 2012. 20, 62

[7] Mountainminds GmbH & Co. KG and Contributors. Java code coverage for eclipse. `http://eclemma.org`, 2006 - 2016, (accessed February 25, 2016). 69, 70

[8] Jens Krinke. Context-sensitive slicing of concurrent programs. *SIGSOFT Softw. Eng. Notes*, 28(5):178–187, September 2003. 101

[9] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, January 1979. 49, 50

[10] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. *SIGSOFT Softw. Eng. Notes*, 19(5):11–20, December 1994. 52

[11] Asaf Shabtai, Yuval Fledel, Uri Kanonov, Yuval Elovici, and Shlomi Dolev. Google android: A state-of-the-art review of security mechanisms. *CoRR*, abs/0912.5101, 2009. 90

[12] Eric Bodden Steven Arzt, Siegfried Rasthofer. Susi: A tool for the fully automated classification and categorization of android sources and sinks. Technical Report TUD-CS-2013-0114, Secure Software Engineering, Group European Center for Security and Privacy by Design (EC SPRIDE), Technische University Darmstadt and Fraunhofer SIT Darmstadt, Germany, May 2013. 20, 54

[13] TutorialsPoint. Junit tutorial. `http://www.tutorialspoint.com/junit/junit_test_framework.htm`, (accessed February 22, 2016). 67