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

Target Level Agreement

Paderborn, July 13, 2015

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

During our project group, we are going to develop an Android App Analysis tool. This document describes, which features and requirements our tool will meet, as well as what we are not going to support.

In the beginning we are shortly going to motivate, why such a tool is needed at all and what it can be used for. Afterwards the scope is explained in more detail (see Section 3). There we describe the analysis levels our tool will support, and in more detail what the different levels will support and where the limitations are. In Section 4 we describe which tools and technologies we are going to use during the development of the tool. How we will ensure the quality of our tool describes Section 5. Afterwards we shortly state in Section 6 how we are planning to ensure maintainability. Further work, which optionally will be realized is listed in Section 7 and the delivery schedule (Section 8) sums up what we are going to hand over to our customer. In the end we shortly list the distribution of responsibilities during the development phase.

2 Motivation

The Android API provides so called permissions for accessing personal or security-critical data, which have to be granted by the user before installing an Application. Data resources for which an Application has no permission, should never be accessed by that Application. Unfortunately there exist several possibilities for Applications to access data, for which they have no permission.

The goal of our Android App Analysis tool is to inform the user about the resource usage of an App, e.g. the relation between the permissions he/she grants and the permissions used by the App, to let him know whether the Application will access data without permission. The resource usage can also be distributed over different Applications. Therefore, our tool will also consider this point, and inform the user about such behavior. For more details see Section 3.4.

Another goal our tool will meet is a critical aspect regarding the behavior of the Application, which is the relation between different resources the App can access. An example for such behavior could be security-critical data flowing from one resource to another resource. Therefore, the user of our tool can choose between different analysis levels, which are explained in more detail in the next section.

3 Scope

The analysis tool we will develop in this project group will be able to perform an analysis on three different, selectable levels: 1, 2a and 2b (see Sections 3.2, 3.3 and 3.4). Details, similarities and differences of all levels will be presented in this section. All three levels will focus on different aspects, which help the user to decide whether he/she should install an App or not. Each analysis will work as described in the following work flow description, which is reflected in Figure 1:

The analysis can be started from the command line or via a graphical user interface (GUI). Executing the tool from the command line will allow the user to use the tool as a part of a bigger work flow, e.g. in a script. In this case the user will configure the tool and by that the analysis via parameters. In contrast to running the tool from the command line, the GUI will offer a lot more comfort. This means that the user can configure the analysis through the interface, instead of looking up the associated parameters. A pending feature might offer a third method to configure the tool by a `.config` file. If this method will be realized all configurations in the file will be overwritten by additional parameters. All methods will offer the same amount of features and possibilities of configuration (see Appendix A). The possibility to run several instances of the tool will be investigated at a later point during development.

As input, the tool will accept one or more Android Application Packages (APKs) as well as saved results from a previous analysis. These input possibilities are symbolized in the Figure 1 by the dashed box on the left. Reading the provided input will always be the first analysis step.

Our next step is 'Run selected analysis' (see Figure 1). For that the user has to specify the analysis' configuration. The user has the possibility to choose between two modes: `SUMMARY` and `COMPARISON`. In `SUMMARY` mode the tool will collect information about the use of resources in the provided App. In `COMPARISON` mode it will be possible to compare two Applications. Depending on the mode the user has to provide different inputs.

To run an analysis in `SUMMARY` mode the tool requires at least one `.apk` file. The result will be displayed in detail at the end of the analysis or after loading a previous saved result. In the latter case no `.apk` file will be needed.

In order to run an analysis in `COMPARISON` mode the tool will require a non-empty set of `.apk` files and a previous analysis result. The set of `.apk` files will be analyzed and compared to the previous result.

To assist the user, it will be possible to view the result in a textual and in a graphical way. The textual representation will always be the most detailed one but the graphical visualization of the result will offer a better overview. These two result viewing options are symbolized by the dashed box on the right in the Figure 1. Nevertheless, the result representations can be enabled and disabled separately. If both representations are disabled the tool will skip the 'Present the result' (see Figure 1 step. Depending on the configuration, the result itself can be saved to a file automatically and immediately after the analysis has finished or manually after viewing the result. The saved result might contain even more information than shown in the textual result representation, because it can include information that can be reused in another level of analysis. Also a saved result might be used as an input for an upcoming analysis as

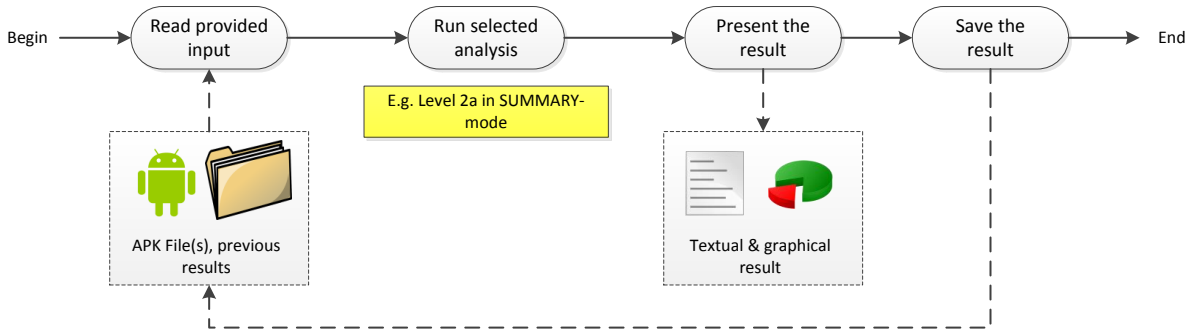

Figure 1: General work flow of the analysis tool

introduced by the dashed arrows in the Figure 1

A complete list of all features considering the tool is provided in Appendix A at the end of the document.

3.1 General Assumptions and Restrictions

We defined some assumptions and restrictions which hold for all three analysis levels. In the following they will be listed:

- The tool will only support intents which are defined and used in the same method. Our tool will not support intents, which are declared as a global variables and cannot be referenced to a single source. This limited support is sufficient, because intents shall not be used for collecting data, instead they should be used for communication between components. Furthermore we will not consider explicit intents for inter-Application communication. If an Application uses explicit intents our analysis will not detect this as inter-Application communication.

  In the case that our tool detects an unsupported intent it will show a warning. The detail of information provided by this warning depends on the fact if the tool can detect why the intent is unsupported. In that way for every intent our tool will either analyze it or throw a warning.

- In Android every object parameter is given to a method by call by value. By that our tool will only support call by value. But Java/Android provides possibilities to use call by reference. These calls will not be supported by our tool. This means we do consider those cases where a method receives a copy of a variable but not where a reference to a variable is handed over. This would be the case if a method changes the origin by using a method bound to the object. The following three lines show such an example:

```
void test(SomeObject x){
        x.doSomethingWithSideEffect();
}
```

The example shows a method called *test* which takes an object of type *SomeObject* as input. The second line shows a method that changes the origin. The method *doSomethingWithSideEffect* will change the content of the variable itself. This will not be supported. Nevertheless the following similar case will be supported since the object variable *x* is decoupled:

```
void test(SomeObject x){
        x = new SomeObject();
        x.doSomethingWithSideEffect();
}
```

If *doSomethingWithSideEffect* would not have side effects and would not change the origin, both examples would be supported.

Finally, it should be noted that if a reference is handed over and the object that the reference points to is modified our tool will not detect this.

- If external libraries are used we also will not analyze those cases further regarding what effects this can have but the tool will display a warning to make the user aware that external libraries are used and that this possibly could be a security problem. The reason is, that external libraries can be large and complex and it will cost to much to analyze a whole external library since we want to focus on Android itself. But the warning will show the name of the library and by that the user can research and decide if he wants to trust that library.

- Another issue which is common for all our analysis levels is the result presentation and the output of warnings or errors. This will be for all supported levels as described in the Requirement Specification. If there are differences in any level it will be mentioned in the specific section.

  One additional, general warning will be provided if details of an intent cannot be referenced. For example every intent needs to define its target. In case of an explicit intent this could be done by defining a Java class. In most cases this Java class will be defined by a Java class object. If this object cannot be referenced to a constant value the target of the intent is undecidable and the tool will provide a warning. This is sufficient as well, because all intent details should always be available and referring to a constant value.

- In general we will support all APIs but for development we will use API Level 22. Newer and older APIs might lead to a lower accuracy of our analysis without adapting the storage for permission mapping, because among other things permissions may change their protection level. If the user attempts to analyze an Application with minimum API level higher than the provided API level the tool will show an error.

- In general we will not support multi-threading. For our levels of analysis a static approach feigning multi-threading can be enough. This will be done by designing the analysis

level independent of the execution order or by interleaving the component executions. Therefore there is no need to put a lot of resources into developing a analysis that takes multi-threading into account.

Besides these points there are further limitations specific to the different levels which are explained in the corresponding sections.

3.2 Level 1: Intra-App Permission Analysis

The Level 1 analysis will focus on the usage of permissions in a single App. Therefore, the tool will need one `.apk` file as input. With the support of the Soot framework (see Section 4.2) the tool will parse the manifest file and an intermediate representation of the Android source code provided by this `.apk` file. The exact way of parsing the source code representation will be determined in the Architecture Document. Based on the parsed information the tool will compute where and how permissions are used or needed. The result presented to the user will subdivide all detected permissions into five classes as specified in the Requirement Specification Document: REQUIRED, MAYBE REQUIRED, UNUSED, MAYBE MISSING and MISSING.

In order to decide which permission belongs to which group the tool will compare the permissions defined in the *uses-permission* tags of the manifest file with all the statements occurring in the source code. To detect which statement uses a permission, a data storage will be deployed that enables the tool to match statements to API calls to permissions. Besides this, the data storage will allow us to map implicit intents to permissions by comparing the action name attribute of an intent with action names stored in the data storage. On the other hand explicit intents will be mapped to permissions by looking up which component is targeted. All the permissions used by the targeted component will be assigned to the intent. The data storage will have a database like structure and it will be adaptable and extendable to support an evolving environment.

3.2.1 Assumptions and Restrictions

In addition to the assumptions and restrictions, which are defined in Section 3.1, there are some Level 1 specific limitations. These limitations will be explained and justified in the following paragraphs.

Intents To switch from one component of an Application to another component, explicit and implicit intents can be used. That is why intents become an important part to an analysis of the information flow of an Application. Therefore, the tool will be able to handle intents of both types. But on the first level of analysis, looking at a single Application and without information about the information flow, the analysis of intents will be further limited than described in Section 3.1.

Every explicit intent is used to start another component of the same Application. If this targeted component requires one or more permissions the intent will be considered as requiring the same permissions. In case of an explicit intent the analysis itself will gather the information about the targeted component. The targeted component will be identified by e.g. the class name which is used in the source code. Which permissions are used in the targeted component will be determined by the level 1 analysis itself. Once the result for a component is computed the permissions will be assigned to the intent. A result is complete if all permissions that could be mapped by the data storage are mapped to the according statements. This means that the statements used for the definition of the intent will be mapped to the permissions used by the targeted component. Figure 2 shows an example regarding this case. The analyzed component

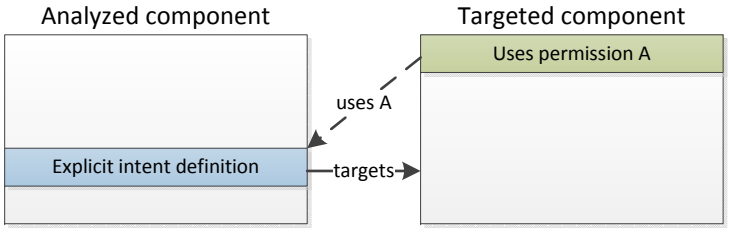

Figure 2: Explicit intent example

is defining and using an intent in a single method. By the statements used for the definition of the intent the targeted component is determined. This component is using Permission A. Therefore, all the statements are mapped to Permission A.

On the other hand, in case of an implicit intent, the tool will use the action name attribute to match the intent to a set of permissions. This can be done by a lookup in the data storage which will provide a set of mappings between intent action names and permissions. With other words the implicit intent will be handled like a statement specified by its attributes. Figure 3

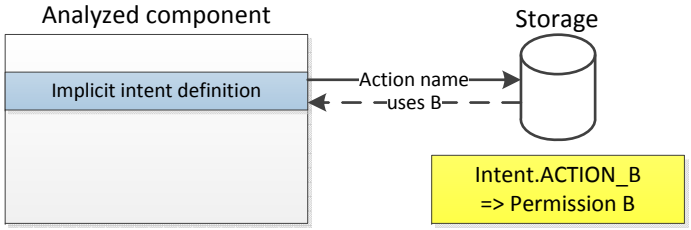

Figure 3: Implicit intent example

shows an example for this specific case. Looking up the action name 'ACTION_B' returns that Permission B is used by the targeted component. That is why all statements in connection to this intent will be assigned using Permission B.

Furthermore, the tool will provide a warning if it cannot determine which permissions will be used by a component started by an intent. For instance, if it cannot match the action attribute

of an implicit intent to a set of permissions.

Once a component has been started by another component through an intent, the targeted component can read the data transmitted by the intent. This will happen by a statement like the following one:

```
getIntent().getStringExtra("data");
```

Regarding the analysis the user can choose between two ways how to handle such statements:

The first option is to just ignore them and only provide a warning if the source code contains such a statement. But since almost all Apps will contain these statements there will be another option: The tool will provide an abstract permission representing all permissions that could be provided to this statement. Which permissions these are in detail can be found out by running a deeper analysis. Therefore, the tool will suggest to run an analysis on Level 2b (see Section 3.4).

Code Sharing An Application may share some of the implementation among different components. Basically this is the case if two or more components use the same method, e.g. use one listener implementation for several view elements. Code sharing can save time and cost during programming.

In the Requirement Specification Document there is an example regarding the case that only the *if* or the *else* part will be executed. Depending on the user action the branch that will be taken differs and different permissions might be involved. In our Level 1 analysis, we will evaluate every statement independent of its origin. In other words we will not take the control flow into account at this level. In general and with respect to the example this means that the tool will interpret any method as if it uses all permissions which are in connection with any statement somewhere in the method. The classification of the permission will be accordingly to the definition of the five classes. So the tool analyzes all possible branches of the code ignoring the conditions.

User Interface Actions Consider the case where a button's action is defined in the activity's layout file. Since the tool will not analyze such files, it will not be able to answer the question whether a method that is not used in the code is used by a button. Accordingly the tool will consider all methods as used somewhere in the component. In that way every permission occurring in a method will occur in the result. This is sufficient, because as described in the last paragraph our Level 1 analysis will be independent of any flow in the Application.

Temporary Permissions An Android Application consists out of different components of different types. One type is a Content Provider. As suggested by the name, a Content Provider provides shared content to other components like a database. The data can be accessed via URIs.

The access to a Content Provider can be restricted in the same way as the access to any other component can be restricted, namely by a permission. Another component will only be allowed to access the Content Provider, if it uses the required permission, unless the Content Provider allows temporary permissions.

Assume there exists a Content Provider requiring a permission and temporary permissions can be granted. In this case the tool will recognize that temporary permissions can be used by analyzing the manifest file. It will map the specific URIs declared in the manifest to the statements using this URIs. These statements will be considered as not using any permissions. If there is a statement attempting to access a URI that cannot be matched to any URI provided by the manifest, the tool will interpret this statement as maybe using a permission. This permission cannot be determined, hence an additional warning will be reported to the user.

In all other cases, if the Content Provider does not:

1. require any permission - all statements connected to the Content Provider won't.
2. grant temporary permissions - all statements will involve the permission(s) in the Content Provider's specification.

If a Content Provider of another Application is targeted it will always be considered as needing the involved permission(s).

3.2.2 Result Representation

The result of the Level 1 analysis will be available in textual and graphical form. The textual form will present the collected information to the user in detail. It can be viewed at `application`, `component`, `class` or `method` level. For example at `method` level the result will show every method with a reference to its unique source and the connected set of permissions. On the other hand the graphical result representation of the `SUMMARY` and the `COMPARISON` mode will be limited to `application` and `component` level, because the graphical representation should be used to get an overview while all the details are provided in the textual representation. In case of the `COMPARISON` mode there are two possibilities:

1. Two different Applications are compared: In this case components, classes and methods of both Applications mostly have nothing in common and it will be sufficient to provide an `application` level representation.
2. Two versions of the same App are compared: In this case the components should be more or less the same. Therefore, the tool will provide the `component` and `application` level result representation.

The tool might not be able to decide which case is investigated. This question will be answered in the Architecture Document. Depending on the availability of this information, the tool will show a warning message to the user if Case 1 is investigated. Nevertheless, the `application` and the `component` level representation will always be available.

The saved result will provide information on `statement` level, too. This is needed to be able to reuse the result in the Level 2b analysis (see Section 3.4).

Further the tool will have options to filter the analysis result in order to have a closer look at a subset of the detected permissions. There will be possibilities to filter the result accordingly to the five groups (REQUIRED, MAYBE REQUIRED, UNUSED, MAYBE MISSING, MISSING) and to filter out a single permission.

3.3 Level 2a: Intra-App Information Flow Control Analysis

On Level 2a, the Android App Analysis tool supports intra-App data flow analyses for tracing the information flow through a single Application. In particular, the tool will analyze the propagation of information protected by some permissions to statements or components which use other permissions. Based on the support of the Soot framework (see Section 4.2) the tool will decompile the input `.apk` file to a Jimple model. With specific analyzing techniques, the tool will then compute and analyze the information flow. Depending on the selected mode (`SUMMARY` or `COMPARISON`), the tool will accordingly display the analysis result which covers the two cases below:

1. Checking secure behavior by summarizing the intra-App information flow.

2. Checking the changes of security in an installed Application by comparing the intra-App information flow with a previous result.

SUMMARY mode In `SUMMARY` mode the tool will compute the resource flow from components providing protected data (known as sources) to components requiring different permissions (known as sinks). There are three different detail levels for users to analyze an `.apk` file. Those are `component flow`, `statement flow` and `resource to resource`, as described in the Requirement Specification. According to the chosen detail level, the tool analyzes the control flow for such kind of detail level.

On all detail levels it will use the same control flow model. If the detail level `component flow` is chosen, the tool will focus on the components of the Application and the connections between them. The detail level `statement flow` takes a closer look on every statement occurring on an information flow path from a source to a sink. The `resource to resource` level has the same processing but concentrates on the information from which source to which sink an information flow exists.

COMPARISON mode In `COMPARISON` mode, the tool provides options for analyzing the already installed Applications based on their previous analysis result and a newly installed Application. So before running the analysis in this mode, the user needs to provide all the necessary `.apk` files and the previous analysis result as input. The tool will analyze the already installed Applications by considering their previous analysis result along with the information

provided in the newly installed Application. The tool will produce the output by analyzing the control flow and also by comparing it with the previous result.

The detail levels in this mode are the same as the detail levels available in the `SUMMARY` mode, i.e. `component flow`, `statement flow` and `resource to resource`. The user can do the analysis based on the detail level selection. Modeling the control flow of the Application components works in the same way as in the `SUMMARY` mode.

As in Level 1 we have to distinguish between cases that two versions of the same App or two different Apps will be compared. In the latter case we will only provide `resource to resource` detail level for the same reason as in Level 1.

For instance, suppose the user selected, the detail level `resource to resource`, then the tool will start evaluating by loading the previous analysis result and by considering the available permissions. Permissions availability is based on the manifest files inside the provided `.apk` files.

The tool will now detect, whether there is any change to the resource flow compared to the already loaded result. If it is available, then the flow related information to the sources and sinks will be available in the analysis result.

The other detail levels, such as `component flow` and `statement flow` can be processed and analyzed similar to the `SUMMARY` mode, except that the provided final result is based on comparing the previous result with the current analysis result.

Information Flow Control In `SUMMARY` mode as well as in `COMPARISON` mode, the tool will deal with explicit and implicit information flow from one resource to another one within an Application. The information flow analysis will be flow-sensitive and includes intra-Application communication and collaboration realized with explicit intents. Furthermore, it will support context sensitivity, which means the analysis will take into account only valid call-return patterns. A pending option is to support object sensitivity to make the analysis more precise and reduce the number of false alarms.

Our tool will assumingly apply one of the analysis techniques listed below for determining the information flow. Which one is going to be used, will be decided during the design phase.

- **Information Flow Control with Flow-Sensitive Typed Systems:** This analysis associates levels of security with variables according to their content type and then provides a different abstraction at each program point.[1]

- **Information Flow Control with Program Dependence Graphs:** This analysis models information flow through a program into a Program Dependence Graph (PDG). PDGs can be applied for both sequential and multi-threaded programs with high precision due to flow, context and object sensitivity. [2]

[1] cf. Hunt and Sands, *On flow-sensitive security types, POPL 2006, ACM, 2006*

[2] As introduced by Hammer, Krinke and Snelting in *Information Flow Control for Java Based on Path Conditions in Dependence Graphs, IEEE International Symposium on Secure Software Engineering (ISSSE 2006), pp. 87–96, IEEE, Arlington, VA, March 2006*

- **Information Flow Control in Logic:** The information flow is specified into domain-specific logics (Hoare like logic) or general program logic (dynamic logic). The logic is mainly based on an abstract interpretation of program traces for program variables. The analysis can prove security or insecurity of programs including advanced features such as method calls, loops and object types. However, it still remains some disadvantages such as requiring theorem proving for dynamic logic. [3,4]

3.3.1 Assumptions and Restrictions

For this analysis level there are also some additional limitations, which will be listed in the following.

In addition to computing the control flow model for components, the tool also decides what permissions are required by what components. The permissions are already processed in Level 1 by analyzing the manifest file and a intermediate representation of the source code. Therefore we can reuse some parts of the Level 1 analysis to get a mapping from statement to permissions.

Based on the Android source code of the Application, we are not able to compute the control flow between components. The reason is that Android Applications are not sequentially programmed and most of the components are launched based on events. So to make sure the model is precise, we are considering the following aspects:

Starting an Application As Android is not having a single main method to start an Application, we have to construct a mock-up main method for starting anyone of the starting components to run the Application for analysis purpose in our tool. The mock-up method was modeled to support all the below mentioned aspects, too.

Lifecycle of Components In general, Android Applications consist of multiple entry points, which are the methods called implicitly by the Android framework in the component. Entry point methods are called as lifecycle methods of the component, which are used to start or stop or pause or resume the same. All the components will have their complete lifecycle defined by Android, as they are implemented by overwriting the lifecycle methods. So we are considering this aspect to exactly analyze how a component behaves and what events are triggered during the different transition of the lifecycle methods.

Execution of Components In most cases Applications consist of multiple components and the order of execution of the components from an Application perspective cannot be predetermined. It often depends on the event which got triggered based on some user input.

[3] cf. Amtoft and Banerjee, *Information Flow Analysis in Logical Form, Department of Computing and Information Sciences Kansas State University, Manhattan KS 66506 USA*

[4] cf. Darvas, Hahnle and Sands, *A Theorem Proving Approach to Analysis Secure Information Flow, Swiss Federal Institute of Technology, Chalmers University of Technology Sweden*

Services can also be executed in parallel with other components. Our tool will not detect such parallel or interleaved executions but we will assume that the components present inside the Application can run in any arbitrary sequential order as well as repetitively. Since we do not support multi-threading this will offer us the best representation of parallel executed components.

Sources and Sinks Possible sources for our information flow analysis will be as described in the Requirement Specification. As sinks we only consider library method calls as described in the Requirement Specification.

The solution for modeling the control flow of the Application components mentioned above is based on the FlowDroid paper[5].

Object Orientated Concepts The tool will support multiple classes/components and it will support communication between these. Also it will not forbid inheritance between classes. But it will not consider consequence of inheritance like method overriding or method propagation. Method overwriting means, that a method of class A is replaced by a method of class B which inherits methods from class A. If a method is unavailable because it belongs to the parent class it will be considered as unknown (method propagation). In this case the tool will provide a warning. This will reduce the accuracy of the analysis just a little bit but on the other hand it will reduce the effort to implement this analysis a lot.

Data Exchange Intents used to communicate between components or between Apps are able to carry data. In our Analysis we will consider all the data that is assigned to an intent as transported with the intent to the targeted component/App. In case of a custom object that might not be the truth but the tool will not further investigate which parts of the object are send with the intent. It will assume that the whole object is sent. Since this is an overapproximation this is sufficient and there will be no loss of information.

3.3.2 Result Representation

In `SUMMARY` mode the tool will show the flow from sources to sinks. In the final result, the user can filter the flow from a specific source to a specific sink. The tool supports both textual and graphical result representation.

The provided result of the `COMPARISON` mode is based on comparing the previous result with the latest analysis result. Similar to the `SUMMARY` mode, users can filter the flow details from the specific sources to specific destinations. Provision is available for the user to save the

[5] As mentioned by Arzt, Rasthofer, Fritz, Bodden, Bartel, Klein, le Traon, Octeau and McDaniel in *FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps; Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, 2014*

provided result, too. The user can also load an existing file for reviewing the previous analysis result saved in that file.

3.4 Level 2b: Inter-App Permission Analysis

Within Level 2b the Android App Analysis tool will support inter-App data resource usage. This is an extension to the functionality of Level 1. But in this level our tool additionally computes the resources used by an App via any sort of supported intent (explicit and implicit) without further restrictions. Hence, it covers the possibility to use data resources indirectly by calling another App's component.

For computing the data resource usages, a graph which models all possible inter-component calls via intents is computed. Moreover, this graph contains all direct resource usages of components. An example usage graph is given in Figure 4. It shows two Apps containing

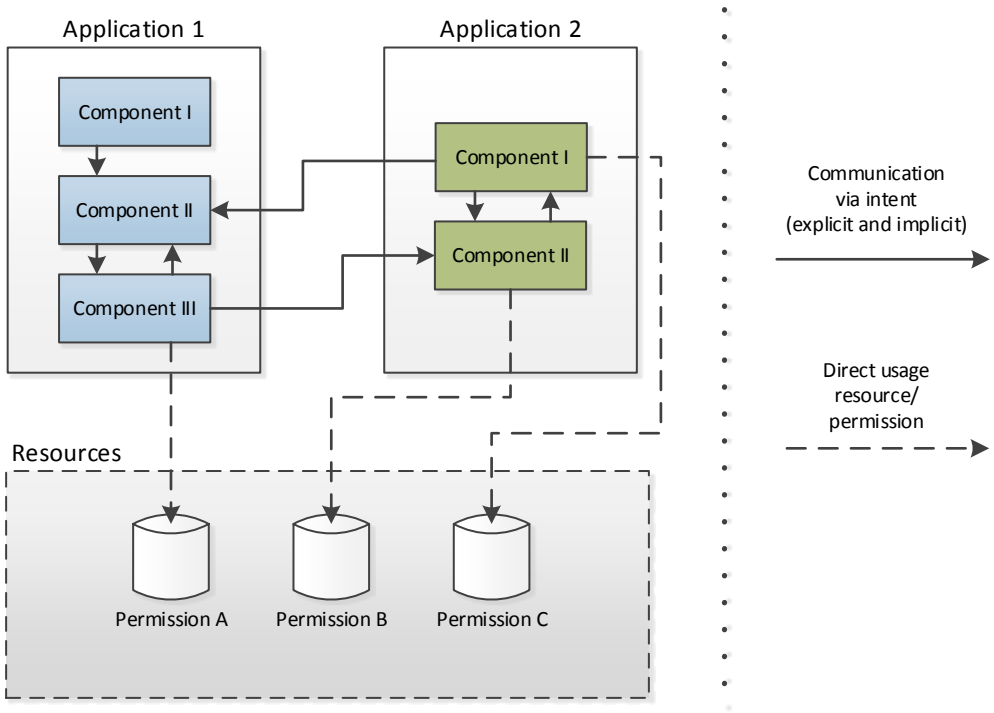

Figure 4: Example usage graph

a different amount of components. The solid arrows symbolize the communication between these components through intents of any kind. On the other hand the dashed arrows reflect the direct resource usage of components.

The analysis of resource usage will be done with a least fixpoint computation within the computed graph and on top of the resource usage identification of Level 1, namely the mapping of intents and system calls to permissions. Afterward we are able to classify direct and additionally indirect resource usages like in Level 1.

As required our tool will extend the two modes `SUMMARY` and `COMPARISON` with a parameter of computation range, namely `APP` and `ALL`. In `APP` mode the tool will compute the resource usages of an App including the indirect resource usages via system Apps and other non-native Apps. The computation will be extended in `ALL` mode to considering all Apps of the input set as starting point. Hence, the tool will additionally detect resource usages that are provided by the considered App. Using the Cartesian Product over the two sets of modes leads to four modes overall: $\{\texttt{SUMMARY},\texttt{COMPARISON}\} \times \{\texttt{APP},\texttt{ALL}\}$

Due to the higher complexity of the Level 2b computation and output because of the consideration of arbitrary many Apps our tool will only provide the detail levels `app` and `component` here. This will guarantee a low computation time and an output with a not too high complexity for easy understandability.

To execute the Level 2b part of our tool the user has to provide at least a single `.apk` file. For the two `SUMMARY` modes additionally a set of `.apk` files (may be empty) is needed that build the android system environment the App to be analyzed works in. For the `COMPARISON` modes the user needs to provide a previously computed Android system environment in form of a usage graph persisted in a file.

3.4.1 Assumptions and Restrictions

Since the computation for Level 2b is an extended version of the computation in Level 1, Level 2b will stick to the same assumptions as Level 1.In addition, we assume that the set of given `.apk` files might be incomplete, meaning that there might be implicit intents for which no intent filter exists.

Like above, Level 2b will also inherit most of the restrictions of Level 1. But in contrast to Level 1, Level 2b will not ignore inter-App intents to foreign, non-native Apps since detecting inter-App resource usage is the main feature of Level 2b. These foreign, non-native Apps are all the Apps that are not provided by the Android system and by that are not supported by our data storage. On level 2b now all Apps specifying intent filters will be taken into account and not only the Applications which are provided by the Android system itself. According to the additional assumption for Level 2b, our tool will give a warning if no matching intent filter is available. Furthermore, a restriction for the opposite case is needed: If there are multiple intent filters for an action string in the given set of Apps we will consider all of them since we will not model default application settings.

3.4.2 Result Representation

Like in Level 1 the resource usages will be assigned to five groups REQUIRED, MAYBE REQUIRED, UNUSED, MAYBE MISSING and MISSING. In addition, for Level 2b we will distinguish between the direct and indirect usage of resources. Therefore, we will also subdivide each group into this two categories and end up with ten groups overall. The filtering options will be the same as in Level 1. It will be possible to filter by groups and to filter out a single permission. But filtering by groups will consider all 10 groups now.

Since all warning and error messages noticed in the Requirement Specification Document will be supported, the tool will give an error message if the group MISSING appears in the result. Equally the tool will give a warning message if the group UNUSED or MAYBE MISSING appears in the result. This holds for both direct and indirect resource usages.

The results will be presented to the user in textual and graphical form as for previous levels depending on the chosen mode and detail level. Since this analysis is restricted to the detail levels `app` and `component`, the result representation will be restricted in the same way. The output (textual and graphical) will be equal to those shown in Figure 17 in the Requirement Specification with the option of improvements for a even better user experience. Another graphical result representation could be a visualization of the usage graph extended with edges for indirect resource usage equal to Figure 13 in the Requirement Specification Document. Both representations will be computed for all results. But the visualization of the extended usage graph is a pending feature and might not be implemented. An example for such an edge would be one between Component II of Application 2 and Permission B in Figure 4. All in all, the results can be interpreted by the user as in Level 1 but here they additionally show indirect resource usages for an App through other non-native Apps.

4 Technology Stack

In this section a description of the tools and frameworks that might and will be used during development of our Android App Analysis tool is given.

4.1 Development Software

Our analysis tool will be developed using **Java**. This will make our tool portable across most operating systems.

We will use **Apache Subversion (SVN)** for version control during the whole development phase. Using a version control system ensures that all team members can work concurrently on the project since it provides e.g. a merge tool to combine conflicting versions of files. SVN provides the possibility to get access to previous versions of files, which will also ensure, that achieved development of any timestamp in the past of our process always can be restored. Besides these features there are many more which will help us to have a working process of high quality.

To keep an overview on upcoming milestones and problems or open tasks we will encounter during development, we are going to use the Issue Tracker **Trac** during the development of our project. Trac provides the generation of tickets which can optionally be assigned to an owner to manage our work effectively.

4.2 Analysis Framework and Tools

Soot will be the framework of choice for the implementation. Soot was originally developed by the Sable Research Group, McGill University and is currently maintained by Secure Software Engineering Group, Darmstadt University of Technology. The current version of Soot can be obtained from the framework's website[6]. Soot's components are based on Java and currently Soot can be run with Java 1.7 and below.

The main advantage of Soot over other frameworks is that Soot can be extended for our project specific requirements. Soot can be used with extensions like **Heros, Flowdroid**[7] and also provides options to add our own extensions. In addition to off-the-shelf analysis like Null Pointer Analysis, we can also create and inject our own analysis into Soot. Other advantages of Soot rely on the facts that Soot is used in a lot of projects and comes with a well structured documentation. In that way there are many sources for examples, tutorials and descriptions of the framework. Moreover Soot offers the possibility to directly generate Call Graphs, which might be of use in order to create our own graphs.

[6] `http://sable.github.io/soot/`

[7] Extensions developed by the maintainer: `http://www.ec-spride.tu-darmstadt.de/forschungsgruppen/secure-software-engineering/tools/`

Dexpler is a submodule of Soot and will be used to obtain Jimple[8] files for the Java class files in the `.apk` file, that will be provided by the user as input.

The name Dexpler refers to the fact that all Java class files are combined to a single file in Dalvik executable format (`.dex`) inside the `.apk` file. Since `.dex` files need to be decoded in order to continue, the Jimple format is introduced to represent the decoded source code. Those extracted Jimple files in turn will be used in the Level 1 analysis to analyze every single statement and match each one to the involved permission(s). As a result, all involved permissions will be assigned to one class, e.g. MISSING.

In addition to frameworks, there exist already other tools which follow a similar goal to ours. To speed up development we will check those tools and might extract useful ideas and approaches. Those tools are:

FlowDroid identifies intra-component taint flows and focuses on information that flows inside a single component of an Application.

We might use some of the approaches of FlowDroid in our tool for taint analysis and depending upon complexity we might extend them to analyzing potentially tainted flows between multiple components.

Epicc identifies properties of intents such as its action attribute string to track both inter-component and intra-component data flow in a set of Android Applications. We might use the ideas of Epicc for tracking both inter- and intra-component information flows.

Finally we want to give a brief overview of how our tool will collaborate with Soot and what the general work flow of an analysis computation is. This is graphically shown in Figure 5. The work flow itself is modeled by the rounded boxes and the arrow between. The dashed arrows roughly reflect where the Soot framework may come into play. Which part of our work flow will be backed by Soot will be specified in the upcoming Architecture Document. At first we will decompose the `.apk` files into the Android bytecode files and manifest file. Then we will pass the bytecode files to Soot to get the Jimple representation. Based on that we will compute the necessary analysis information (e.g. a graph) depending on the chosen level. Furthermore, Soot might be used for parts of the analysis itself. After computing the analysis result Soot will not be used until another analysis is started. The work flow is finished after the result is saved to a file and/or presented to the user.

5 Evaluation

To provide a statement of trustworthiness for our tool we will evaluate it against competitiveness and quality. How the evaluation will be done is described in the following sections.

[8] Jimple is an intermediate representation between Java bytecode and Java source code

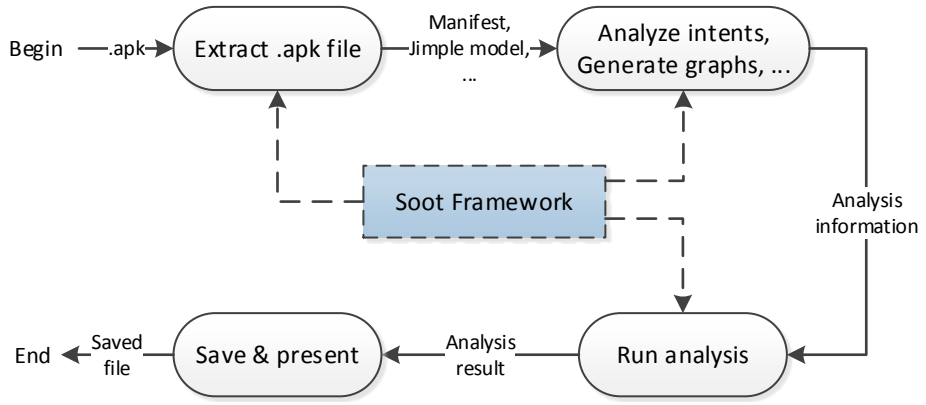

Figure 5: Soot Framework

5.1 Competitiveness

For evaluating the competitiveness of our tool we deal with the following questions:

1. How does our tool compare with other tools?
   - Is the accuracy higher or lower?
   - Does it find more or less leaks?
2. Can our tool detect information leaks in real world apps?

For comparison with other tools we will use common benchmarks for android security. Those benchmarks are **DroidBench**[9] and **ICC-Bench**[10].

DroidBench is a set of `.apk` files created by the FlowDroid team for testing their tool. It contains test cases amongst others for implicit flows, inter-app communication and field and object sensitivity.

ICC-Bench is another small set of android applications introduced by the **Amandroid**[11] team. Amandroid is also a tool for testing information flow control on android. ICC-Bench covers information flow via implicit and explicit intents.

Those two tools, FlowDroid and Amandroid, will be the tools we compare our tool with according to the benchmark results. Moreover, some other commercial tool might additionally be taken under consideration. This could be **IBM's AppScan Source**[12] or **DidFail**[13].

To test our tool on real world apps we will choose some of the most popular apps from the **Google PlayStore**[14]. We will consider one more possibility to test our tool with a special set

[9] `http://sseblog.ec-spride.de/tools/droidbench/`
[10] `https://github.com/fgwei/ICC-Bench`
[11] `http://amandroid.sireum.org/`
[12] `http://www.ibm.com/software/products/de/appscan`
[13] `https://www.cert.org/secure-coding/tools/didfail.cfm`
[14] `https://play.google.com`

of Malware apps provided by a package called **MalGenome**[15].

5.2 Quality

To ensure reliability and correctness, there will be zero tolerance for errors in the code. Therefore we will use external tools that run tests, analyze and even automatically optimize our code. The tool we use will be an analysis tool like **Findbugs**[16], **Codepro AnalytiX**[17] or **SonarQube**[18].

Code analysis tools like those mentioned above most likely integrate automated **JUnit**[19] tests. Additionally, they provide methods for analyzing the code structure with respect to coding standards. They create reports to help us to identify weaknesses in our code.

Moreover, to check the reliability of our tool we will use system tests. For such tests we will craft special sets of testing Apps for which we already know the analysis result. Each test set will be made for testing a single kind of security leak, e.g. illegal resource usages. We then will compare our expected result with the actual result of our tool.

Last but not least, maintainability and reusability are also important features for code quality. This topic will be covered separately in Section 6.

6 Maintainability

After all our tool should have a good maintainability and be easily extensible. This means that we will provide features that reduce the workload to extend/adapt our tool to new tasks and environments. To reach this goal we will make use of the object-oriented approach of the Java programming language.

Moreover we will use model-based software patterns[20] whenever the usage of those is appropriate. This also includes providing interfaces, extensible classes and a software API for extension and reuse of the functionalities of our tool. For instance, we want to be able to:

- Add new analysis types (levels)
- Add new result representations (for new analyses)
- Add new data storages for permission mappings (supporting newer, upcoming Android APIs)
- Use completely new GUIs for our tool (e.g. web interface)

[15] `http://www.malgenomeproject.org/`
[16] `http://findbugs.sourceforge.net/`
[17] `https://developers.google.com/java-dev-tools/codepro/`
[18] `http://www.sonarqube.org/`
[19] `http://junit.org/`
[20] As introduced by Gamma, Helm, Johnson and Vlissides in *Design Patterns - Elements of Reusable Object-Oriented Software, Prentice Hall*

Additionally, we will use the tool's API ourselves to separate the tool's logic from any kind of user interface.

Finally, our tool will support different Android APIs. This is highly needed because Android evolves continuously. The API that should be used for analysis can be selected by the user. During development our tool we will work with Android API 22.

7 Further work

We decided to envision three more goals which are worth to be achieved in future. If our time-schedule allows it, we will try to achieve these goals by ourselves.

- **Level 3 analysis:** Analyzing the inter-App information flow would be the next step regarding the analyses. Mostly this analysis would work like the Level 2a analysis (see Section 3.3). The difference would be, that we would take the information flow between Apps into account and stop being stuck in one Application.

- **User image:** In the motivation (Section 2) the situation, the tool will be made for, is described. A user wants to know if he/she can install an App without being afraid of security issues. To offer more comfort the tool itself should be able to hold a set of applications equal to the ones on the user's device. This set, the user image, would always be available to the tool without loading previous results or reconfiguring a previous analysis. The goal of this approach is to reduce the time spent by the user to configure and start an analysis and to reduce the runtime of an analysis itself working with this specific user image. Furthermore, it should reduce the effort to check a single App before the user decides if he/she installs it.

- **Android App Analysis App:** Another goal would be to provide an App that synchronizes all Apps installed on one device with our tool. More specifically the App as a client would send all `.apk` files to our tool, that plays the role of a server. This App would require superuser access to read all the `.apk` files stored in the system area of the device. By that it would only work on rooted devices. Apart this issue the App would offer a lot of comfort to the user and would enable the tool to do a *1-Click-Analysis* of the whole set of Applications installed.

8 Delivery Schedule

The first thing we are going to deliver is this Target Level Agreement itself. Then until July 31st we will deliver our Architecture Document. This document will include all the details about the Android App Analysis tool we are going to implement. The number of open questions will be significantly reduced. In contrast to this document it will be more detailed regarding the implementation.

After developing the Architecture Document, we will start implementing the tool itself. By then, in the optimal case the implementation should just be an roll out of the described architecture. Nevertheless, in any case the Architecture Document will be the guideline for the whole development of our tool.

The implementation will be done until the end of the project group, namely end of March 2016. The final implementation will be delivered as source code and as an executable `.jar` file including all required dependencies. The license needed in order to use our tool will finally be determined later on. Until now it will only require the GNU Lesser General Public License (LGPL), because we are using Soot as a basis and Soot requires the LGPL. With the final implementation we will deliver a final documentation, which will be based on the Architecture Document and extended by the information gained during the implementation phase. Furthermore, it will include the documentation of all executed tests described in the evaluation Section 5, information about the test material itself and a user manual. The user manual will describe how to install and use the tool and will be delivered including examples and tutorials.

In the Architecture Document we will also define milestones. After reaching a milestone we will prepare a presentation in order to be ready to present the current status of the project. At the end of the project there will be a final presentation prepared as well.

Another extra we will prepare is a website, that enables the user to easily access all the information and deliveries the project group will provide.

9 Responsibilities

During the development phase the work will be distributed among all group members. To ensure that this distribution works well and that there is always one person who keeps the overview, we elected some people from our team, who are responsible for different areas. The responsibilities are listed below:

- Project Leader: Felix Pauck
- Implementation Manager: Pham Thuy Sy Nguyen
- Quality Assurance Manager: Arjya Shankar Mishra
- Documentation Manager: Monika Wedel

A Feature Table

I. Deployed Environment			
No.	Feature	Supported	Pending
1	Windows and Linux	✓	-
2	JDK (or JRE) 1.7	✓	-

II. Android App Analysis Tool			
No.	Feature	Supported	Pending
3	User can run analysis by command line for android application(s)	✓	-
4	User can run analysis on GUI for android application(s)	✓	-
5	User can view analysis result in command line (in textual representation) or call GUI for viewing analysis result (in graphical representation)	✓	-
6	User can view analysis result on GUI (in textual or graphical representation)	✓	-
7	Single instance tool	✓	-
8	Multi-instance tool	-	✓
9	Input file is `.apk`	✓	-
10	Input file is `java byte code`	×	-
11	Analyzing multiple applications (multi input files) at the same time	✓	-
12	Analysis result can be stored in an output file	✓	-
13	Analysis result file can be loaded again for reviewing and comparing	✓	-
14	Show error message when errors occur (cannot parse `.apk` file, missing manifest file etc.)	✓	-
15	Show notification message for each operation of the tool	✓	-
16	Log system for tracking tool's operations	✓	-

II.1 Lv1 Resource-Usage of an application			
No.	Feature	Supported	Pending
17	User can configure analysis on GUI	✓	-
18	User can configure analysis on command line by parameters	✓	-
19	User can configure analysis in `.config` file	-	✓
20	User can run analysis in two modes: `SUMMARY` or `COMPARISON` (`SUMMARY` mode for information of resources used in the application and `COMPARISON` mode for comparing changed resources between two versions of the same application or between two different applications)	✓	-

No.	Feature	Supported	Pending
21	User can run analysis in four detail levels: `APP`, `COMPONENT`, `CLASS`, `METHOD`	✓	-
22	User can filter information of analysis result based on chosen detail level	✓	-
23	Based on analysis result, a deeper analysis might be suggested	✓	-
24	Analysis result can show resources directly used by the application	✓	-
25	Analysis result can show resources directly used by specific App's components (Activities, Services, Broadcast Receivers, Content Providers)	✓	-
26	Analysis result can show resources directly used by specific classes including the component classes	✓	-
27	Analysis result can show resources directly used by specific methods	✓	-
28	User can specify a path for saving the analysis result	✓	-
29	User can specify a path and load a previous analysis result of the same or a different application to compare them.	✓	-
30	User can filter the usages of permissions on the comparison result	✓	-
31	Permissions will be classified into five different usages: REQUIRED, MAYBE REQUIRED, UNUSED, MISSING, MAYBE MISSING	✓	-
32	User can save comparison result to file	✓	-

II.2 Lv2 Intra- App Information Control Flow and Inter- App Resource Usage			
	Lv2a. Intra-App Information Flow Control		
No.	Feature	Supported	Pending
33	User can run analysis in two modes: `SUMMARY` or `COMPARISON` (`SUMMARY` mode for information control flow of resources through statements or components in the application and `COMPARISON` mode for analyzing changes of information control flow in an application when a new other application being installed)	✓	-
34	User can run analysis in three detail levels: `resource to resource`, `component flow`, `statement flow`	✓	-
35	The tool takes implicit information control flow into account	✓	-
36	The tool processes flow-sensitivity	✓	-
37	The tool supports context-sensitivity	✓	-

No.	Feature	Supported	Pending
38	The tool supports object-sensitivity	-	✓
39	The analysis result can show flows of information from specific resources to specific resources	✓	-
40	User can check whether there exists a flow from a specific resource to another one or not	✓	-
41	The analysis result can show which components are involved in a flow from a specific resource to another one	✓	-
42	The analysis result can show the respective execution paths from a specific resource to another one	✓	-
43	User can extract paths from specific resources to another resource in detail level `resource to resource`	×	-
44	User can extract paths from specific resources to another resource in detail level `component flow`	✓	-
45	User can extract paths from specific resources to another resource in detail level `statement flow`	✓	-
46	User can filter in the analysis result based on the start nodes and the end nodes	✓	-
47	The tool shows a warning to the user if the tool encounters an violation of information flow	✓	-
48	User can check a changed information control flow of an already installed application influenced by new application by loading a previous analysis result (Lv2 analysis result) and comparing them	✓	-
49	User can check the changed information control flow of two versions of the same application by loading a previous analysis result (Lv2 analysis result) and comparing them	✓	-
50	The user can check that no new paths are introduced to analysis result of information control flow in one application	✓	-
51	The tool supports the same permission category like Lv1 analysis (see case 31)	✓	

	Lv2b. Inter-App Permission Analysis		
No.	Feature	Supported	Pending
52	User can run analysis in four modes: `SUMMARY APP`, `SUMMARY ALL`, `COMPARISON APP` and `COMPARISON ALL`	✓	-
53	User can run analysis in only two detail levels: `app` and `component`	✓	-
54	User can provide multiple `.apk` files for building the android environment in which the App is tested	✓	-

No.	Feature	Supported	Pending
55	In `COMPARISON ***` mode, user must provide a previously computed android system environment in form of a usage graph persisted in a file	✓	-
56	The analysis result can show what resources are accessed by an App's component directly or indirectly	✓	-
57	The analysis result can show that direct or indirect resource usages are missing or resource usage definitions are superfluous	✓	-
58	The analysis result can show the resource usage classification (explicit and implicit) change (in comparison to a previous analysis result) for already installed Apps	✓	-
59	The analysis result can show a component graph which describes the inter-component communication of the Apps' components among each other	✓	-
60	Based on the analysis result, if MISSING permission appears, the tool will show an error message	✓	-
61	Based on the analysis result, if UNUSED or MAYBE MISSING permissions appear, the tool will show a warning message	✓	-
62	The user can filter the analysis result by permission	✓	-

Software Development Contract

By this contract

AG Wehrheim

- in the following called 'Customer' -

and

PG Android App Analysis

- in the following called 'Contractor' -

agree on the following:

The Contractor will develop the software product for the Customer as described in this Target Level Agreement.
The product will be delivered to the Customer by the end of March 2016 .

Paderborn, on July 13, 2015

.....	.....	.....
Marie-Christine Jakobs	Manuel Töws	Heike Wehrheim
.....	.....	.....
Abhinav Solanki	Anand Devarajan	Arjya Shankar Mishra
.....	.....	.....
Fabian Witter	Felix Pauck	Monika Wedel
.....	.....	.....
Pham Thuy Sy Nguyen	Ram Kumar Karuppusamy	Sriram Parthasarathi