Universität Passau

Software Product-Lines Group
Dr. Ing. Sven Apel

and

IT-Security Group
Prof. Dr. rer. nat. Joachim Posegga

Master Thesis

Variability-aware Data-flow Analysis for Smartphone Applications

This work was created in the context of the EU Project COMPOSE (FP7-317862).

Author: Daniel Hausknecht
Supervisors: Sven Apel
Joachim Posegga
Daniel Schreckling
Submitted: 2013-09-23

University of Passau
Faculty of Computer Science and Mathematics

Software Product-Line Group
&
Chair of IT-Security

Master Thesis

Variability-aware Data-flow Analysis for Smartphone Applications

Daniel Hausknecht

Date:	23. September 2013
Supervisors:	Dr.Ing. Sven Apel
	Prof. Dr. rer.nat. Joachim Posegga

Abstract

In this master thesis we model huge sets of smartphone applications (e.g. an application store) as a software product line to fit for a variability-aware analysis. We developed and implemented algorithms for the detection of data security leaks emerging from the interaction of applications. An evaluation shows the efficiency of our approach.

Declaration of Authorship

I certify that the work presented here is, to the best of my knowledge and belief, original and the result of my own investigations, except as acknowledged, and has not been submitted, either in part or whole, for a degree at this or any other University.

Passau, 22nd September 2013

.....
(Daniel Hausknecht)

Supervisor Contacts

Dr.-Ing. Sven Apel
Software Product-Line Group
University of Passau
EMail: apel@uni-passau.de
Web: http://www.infosun.fim.uni-passau.de/spl/

Prof. Dr. rer.nat. Joachim Posegga
Chair of IT-Security
University of Passau
EMail: posegga@uni-passau.de
Web: http://web.sec.uni-passau.de/

Contents

List of Figures

1. Motivation

Over the past 10 years, the market for smartphones has grown rapidly and is still enormously growing [2]. In the fourth quarter of 2010, smartphones even outsold PCs for the first time. With the high sales rates for smartphones, the market for applications running on these devices also grew permanently. The two biggest application markets, Apple's App Store and Google's Play Store, reported in total over 40 billion [12] and 25 billion [21] downloads in 2012, respectively.

These statistics are an indicator on how intensively smartphones are used nowadays. In fact, smartphone have become an important part of our private as well as business lives. Users personalise their devices and store sensitive data such as their contacts, their calendar, or other kind of protect-worthy data on their phones. But also the smartphones themselves provide a variety of potentially sensitive data sources, such as the GPS location or the device's identification number (IMEI). If any of this data, or combinations of it, can be unwantedly exposed, this can cause a serious security leak. Therefore, there is a need for mechanisms which can detect and warn users from leaking data-flows.

Today's smartphone operating systems such as Google's Android [14] ship various security mechanisms for the protection of application internal data[26]. Android implements a sandboxing mechanism that, e.g., forbids access to an application's run-time environment from an application external entity. Inside an application sandbox, a developer is completely free how she wants to use sensitive data and it is up to the user whether to trust the developer and install an application or not.

Android also implements a permission mechanism which controls access to certain system services (for example the establishment of network connections or access to device sensors) or to custom application features. All required permission must be requested for an application at installation and can only be granted by the user. The set of requested permissions reflects the capabilities

of an application and helps a user to decide if she wants to trust the application and install it. For example, when an application requests the permissions for Internet access and at the same time to the contact book, it could unwantedly send all contacts to an untrusted server.

However, there are two main problems coming with Androids permission system: firstly, the permissions were criticised to be too coarse grained [18, 19]. Secondly, a recent study showed that permissions are rather rarely used in practice [5]. Octeau et al. [5] also report that only roughly 5% of the system-wide accessible application components[1] were protected by permissions. This high rate of exposed but unprotected components provides a broad range of targets for potential so called permission re-delegation attacks. Permission re-delegation is a special case of the confused deputy problem [11] in which an application component exploits the permissions of another component which it does not have itself [9]. Let's say we have an application A with some component c_A and no granted permissions. Also, we have an application B with granted permission p to access component c_C of application C. Calling component c_B of application B does not require any permissions. So, c_A can call c_B with parameters such that c_B calls c_C. Thus, c_A can access the features provided by c_C without being directly allowed to call c_C but by exploiting the permissions of component c_B. The scenario is illustrated in Figure 1.1.

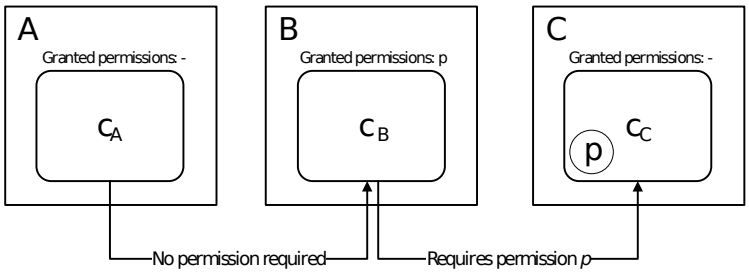


Figure 1.1.: Permission re-delegation exploit using Android application components

[1] components are the basic program modules of Android applications. Simplified spoken is an application a set of component implementations (for more see Section 3)

As it is for any other platform, also smartphone operating systems struggle against security issues through malware [24]. Therefore, many scientific research has been done to either improve the existing security mechanisms [18, 19], or to introduce additional security features for application security [7, 8, 10].

Though most developments vary tremendously in their actual approach, they all have one commonality: the provided solutions are device-centric, i.e. they are integrated in the operating system. As a consequence, the mechanisms all run directly on the device introducing a certain performance overhead. More importantly, only a device with the improved security mechanism installed gains an increased level of protection. Whereas the issue of performance overhead can be reduced to some extent through optimisations, a comprehensive deployment of a mechanism is much more of a practical issue. In fact, smartphone vendors fail to provide updates to newer Android versions for older devices, leading to a high percentage of outdated still-in-use Android versions [22]. Hence, even if one of the approaches was integrated in a new Android version, most Android devices would not be affected by the protection mechanism. It is therefore much more appropriate to develop a protection mechanism that is fully independent of the version of the operating system. Our goal is to detect undesired data-flows in and between smartphone applications. So, to be device independent, we design our approach to run directly in the application store and to analyse the applications before even installing them.

Device-centric approaches have the big advantage that the system's configuration and the set of concretely installed applications are known, whereas for an application store-side approach this knowledge is of course hidden. The information is in particular crucial for inter-component data-flows and the detection of the above described permission re-delegation problem. As a naive approach one can try all possible application install combinations and check them for data security conflicts. However, the number of combinations is of up to exponential size and thus the naive approach is highly impracticable. We will provide a solution which borrows techniques from the field of software product line engineering [1] to cope with the sheer size of the analysis space. The basic idea is not to verify all possible combinations separately but instead to analyse all applications at once.

We introduce VarDroid, a tool that tracks data-flows across application boundaries and detects data security conflicts. VarDroid works in three phases: first, the application internal analysis, second, the interrelating of application

components, and third, the detection of data security conflicts. We use Android as an exemplary smartphone operating system for our prototype implementation.

The master thesis is structured as follows: after looking at related work in Chapter 2, we introduce the Android basics (Chapter 3) and give an overview of variability-awareness in Chapter 4. In Chapter 5 we provide a definition of security labels. Chapter 6 covers the newly developed approach in detail by discussing the basic algorithms and demonstrating their workflow in examples. In Chapter 7, we describe our prototype implementation *VarDroid*, as well as a generator for random components which we use for our evaluation. The evaluation is provided in Chapter 8. Chapter 9 covers current limitations. Last, we give an outlook on future work in Section 10.

2. Related Work

There are several approaches analysing Android applications in terms of security matters.

TaintDroid [7] is a dynamic taint analysis tool which is directly embedded in Android. It tracks the runtime flows of data and detects security violations. However, for the reasons given in Chapter 1, for example that we don't want to modify the operating system itself, we aim for a fully static analysis instead.

Other approaches try to detect possible data insecurities based on Android's permission system [8, 9]. Kirin, for example, statically checks for undesired combinations of permissions requested by an application. Though permissions are an indicator for the capabilities of an applications, the approach is quite coarse and does not consider the true behaviour, i.e. its control- and data-flow, which can lead to misleading results.

Tools like FlowDroid [3] or Chex [17] analyse the data-flow of Android applications. Though FlowDroid and Chex differ completely in the way they're implemented, both identify data sources and sinks within the program code and detect security violations. They focus on the analysis of individual applications and their internal behaviour. They do not consider inter-component or inter-application data-flows. However, these tools match the requirements for Phase 1 of our approach. In fact, we are currently working on integrating FlowDroid for extracting the relevant information from Android applications and for providing the information to VarDroid's inter-component analysis.

ComDroid [4] and Epicc [5] are tools which focus on inter-component and inter-application analysis. Their goal is to find vulnerabilities and attacks enabled through inter-component communication, e.g. application hijacking. They first identify entry and exit points of application components, and second, interrelate them. The main object of observation is the usage of intents. In contrast, we want to detect when incautious uses of intents lead to data security vulnerabilities. So, instead on the misuse of intents, we focus on the misuse of data sent via intents.

SCanDroid [10] is the approach closest to ours. Their approach is to incrementally analyse applications as they are installed on a device. They perform a component internal analysis based on the Java source code and additional sources such as the manifest file. A checker module then uses the application information to track data-flows including cross-component and -application flows. The result of their analysis are constraints over a permissions. The first problem with SCanDroid is that they work on the actual Java source or byte code, not on Android specific code. In practice, however, applications are distributed in the Dalvik byte code, an Android specific variant of the Java byte code. VarDroid in contrast works on a component black box representation which makes our own implementations completely independent from any kind of source code. For the black box generation, we use analysis tools specialised in Android handling the application Dalvik byte codes for VarDroid. Also, specialised tools incorporate the exact behaviour of the Android API in their results. SCanDroid, on the other hand, uses self-defined stubs to model API calls which are rather imprecise and incomplete. SCanDroid was designed to analyse an new application at install time in the context of the already installed ones. So, SCanDroid follows an incremental approach. Analysing a huge set of applications, as we aim to do with VarDroid, with an incremental approach would demand runs in the quantity of the size of the application set and each run with growing complexity. VarDroid was designed to handle a huge set of applications and only needs a single pass.

3. Android Basics

Our tool is designed to analyse sets of applications created for Google's smartphone operating system Android. In this chapter, we will cover the basic structure of Android applications and the basic mechanisms for inter-application communication. It is not required to fully understand the Android platform for this work and we will focus only on the aspects which directly affect our approach. For a complete description of the Android system we refer to the official documentation [13].

3.1. Basic System Architecture

Android is a Linux-based operating system for mobile devices. Besides the Linux kernel and device drivers, Android comes with libraries, e.g., for database support or the standard C-libraries. The upper layers provide programming interfaces and services which can be used by the actual applications. Android's system architecture is shown in Figure 3.1.

All applications are written in the Java programming language. However, the applications are compiled into so called Dalvik byte code which is suited for running in a register-based Java virtual machine as it is used by Android. By default, each application is started in its own virtual machine sandboxing an applications runtime environment. Also, Android uses the Linux user identifiers for making files accessible only to the applications which created them. Application internal data can therefore only be accessed by other applications through Android's inter-application communication mechanisms.

[1] http://source.android.com/devices/tech/security/index.html

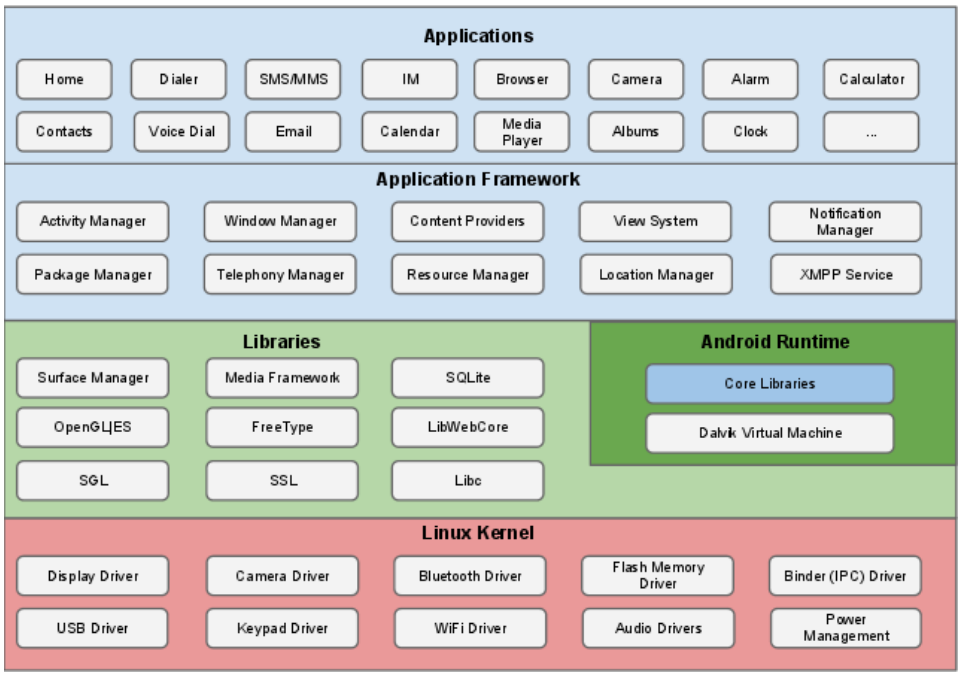


Figure 3.1.: Android system architecture[1]

3.2. Application Components

An Android application is basically a composition of so called *components* functioning as its logical units. There are four different types of components predefined by Android which fulfil certain purposes. The component types are as follows:

Activity: These components implement the graphical user interface (GUI) for applications and short computations. Every application must ship at least one Activity component which acts as the start-up component for an application.

Service: A Service component can be used to implement background tasks which might take longer (e.g. a file download), or just do not require any direct user interaction. A Service are controlled though hooks for callbacks and other inter-component communication.

Content Provider: The purpose of a Content Provider is to provide a data management module. Its predefined programming interface is designed to resemble common database interfaces, in particular to ease the usage of SQLite which ships with the Android base libraries. Content Providers are a common way to provide application-external components access to the application-internal data. This data is otherwise protected by the application sandboxes and therefore not directly accessible to application-external components. Note that Content Providers are not the only way to transfer data between components. Another way is, e.g., via Intents (see Section 3.3).

Broadcast Receiver: The fourth component type is for listening for and reacting to system-wide announcements, e.g., Android sends a broadcast message on low battery level. Broadcast messages are sent as Intents which are described in Section 3.3.

3.3. Intents and Intent Filters

An Intent is a message for inter-component communication. Among others, it can be used to start other components and transfer data. So, technically spoken, an Intent is a container object holding a description of the addressed target components, flags (e.g. how to start a component), and optionally data which can be transmitted with an Intent object. For inter-component communication, an Intent is created and filled with the respective data. Then it is handed over to the system which resolves for the target component and initialises the intended action. For example, when an Intent is given to the system through calling the API function *startActivity*, Android tries to start the addressed Activity. There are two ways on how to specify the target components:

Explicit Intents When creating an explicit Intent, the target component is explicitly named using the component's unique identifier. The processing of an explicit Intent succeeds only if the specified component exists and the Intent creator has the required permissions to communicate with the target component.

Implicit Intents Implicit Intents do not exactly define the target component but provide a description of the wanted feature the target component should implement. A component can be described with a category of the provided feature (e.g. "CATEGORY_GADGET" for an Activity which can be embedded in a host Activity) and the action it performs (e.g. "ACTION_CALL" if the target component should be able to initialise a phone call). With implicit Intents it is possible to have multiple target component candidates. In this case Android asks the user to select an application from a list before, respecting the user's choice, continuing with the processing of the Intent.

3.4. Permissions

With Intents Android provides a mechanism to access other application components. But Android also provides ways to restrict this access if in certain cases it should not be allowed for security reasons. In case a component should only be available for components within the same application, a flag can be set at development time which reduces the visibility of the particular component to its application. The other case is that access to a component is required from outside an application but its availability should be restricted nonetheless. Android comes with a permission mechanism that addresses this situation.

Permissions are defined and associated with a component during development time. If a component has to be allowed to access a permission-protected component, the permission must be specified for request at development time, and granted by the user at installation time. The set of requested permissions is static, i.e. it cannot be changed at any time later after installation. An application can only be installed if all requested permissions are granted.

4. Variability-awareness

In this chapter, we look at the basic verification strategies commonly used in the field of software product line engineering, project the scenario of analysing a whole application store into the verification strategies, and eventually argue why we chose the variability-aware strategy for our analysis. The explanations of the strategies, in particular the one about variability-awareness, not only helps to better understand the overall design decisions but also the workflows of the algorithms presented in Chapter 6.

Note that this chapter only provides an brief overview of software product lines and variability-awareness. For more details on this topic, we refer the reader to the literature, e.g., the survey by Apel et al. [1] or Thüm et al. [23].

4.1. Software Product-Line Verification Overview

A software product line (SPL) is a set of software systems which share common features. The different end products are created from a common source base by configuring the compilation to enable or disable certain features, respectively. A famous non-commercial example is the Linux kernel providing configuration options for hardware features like USB, network interfaces etc. [15].

Our overall goal is to analyse a smartphone application market with regards to data security. We see the set of all applications as our SPL and the individual applications as its features. The installation and removal of applications to and from a device is therefore equivalent to enabling and disabling of features, respectively. The feature model dependencies are given through the accessibility information of applications and their components (for example through Android permissions).

When analysing SPLs the differences of the end products must be taken into account. It is not only important that each feature by itself provides valid code, but also that the interactions of enabled features do not lead to defects. For our approach, feature interactions are equivalent to application interactions.

There are three basic analysis strategies which we will now briefly discuss and rate for applicability in the context of our approach.

4.2. Product-Based Strategy

The naive way of analysing a SPL is to simply verify all possible products. Though this strategy provides full coverage it is only applicable to very small SPLs. For instance, for a SPL with only three features there are up to 8 ($= 2^3$) possible feature combinations. When increasing the number of features to only 10, there are however already 1024 ($= 2^{10}$) possible combinations. So, the product-based verification strategy does not scale for SPLs with even only moderate feature size. In fact, as just demonstrated in the small example the number of combinations can grow exponentially with the number of features.

For our approach, we see each applications as a feature which can be enabled through installation on a device. With an application market of roughly 700.000 applications [21], the product-based strategy is clearly not an option for us.

4.3. Sample-Based Strategy

A way to reduce the enormous number of possible products as described for the product-based strategy is to only analyse a selected subset. There are various heuristics for the selection criteria for the sample products, e.g., single conf, pairwise or code coverage.

Since we modelled applications as the features in our approach, samples are certain sets of installed applications.

The main issue with sample-based strategies is that it reduces the analysis coverage. This implies for our approach that security conflicts are likely not to be detected. For the above given example heuristics, this is in particular the case when security conflicts emerge from a call chain of more than two application components. Also, a recent study showed that the variability-aware strategy outperforms the sample-based strategy in terms of detection efficiency while providing full analysis coverage [1]. Because of these drawbacks, we chose not to use the sample-base strategy but to implement variability-awareness instead.

4.4. Variability-aware Strategy

The third strategy is called variability-aware or family-based analysis. The basic idea is to analyse all possible products but to reduce the analysis space by exploiting the similarities of the products in a product line. The key features are late-splitting and early-joining which allow to merge equivalent program states as often as possible. The common states must then be handled only once during the analysis instead of multiple times for each individual product. A variability-aware analysis runs in a single pass while providing full analysis coverage.

In our approach, the source code of a SPL is given through the source codes of the installed applications. Thus, we can merge data-flows as soon as the states in an application execution are the same. Since we want to detect security conflicts in data-flows, states are later defined with respect to the security properties of the flowing data. Consequently, we split the analysis on data-flows with different security properties.

We chose the variability-aware strategy for several reasons: first, it provides full analysis coverage, i.e. we detect all possibly emerging security conflicts. Second, recent studies showed that variability-awareness scales when applied to real-world product-lines and can even outperforms the sample-based strategy [16].

5. Security Labels

Security labels are the main object of our analysis. In our approach, we propagate them through a component black box call graph to detect security conflicts. Through security conflicts we can identify possible data leaks in the applications. Therefore, it is important to have a clear understanding of what security labels and security conflicts are. In the following sections we will define both terms in general and also give a concrete definition which we use in the course of this master thesis.

5.1. Definition

The goal of our approach is to trace security labels associated with data-flows. So, a security label is in our context some kind of security related data "history", i.e. it records the security classifications of actions performed on the flowing data. As an example, we assume that the current location of a mobile device is protect-worthy data. The location manager of a device as the producer of the location data is therefore annotated as *private*. As consequence, all data-flows having one of their sources in a location manager must somehow include *private* in their security label. The same applies to data sinks. Lets assume we declare any outgoing network connection as an untrusted output. So, we annotate every network socket as *public*. As an implication, the security labels of data-flows reaching a network socket are updated with the *public* security label.

In the two above examples we used terms which need clarification: annotations, and *private* and *public*, respectively.

We use annotations to associate security labels with certain Android API calls. The annotated security labels classify the implications of an API call execution for the processed data in terms of security. In the first example, the classification of the location manager API was that we labelled it as *private* to express protect-worthiness. Network sockets, on the other hand, were classified

as untrusted and thus annotated with *public*. For the master thesis we assume the annotations as given for all API calls.

In the examples we used *private* and *public* as the security labels. Though these were intuitive enough and therefore sufficient for the examples, we will now abstract from it and give a general formal definition of security labels as we use them throughout this work.

We define a security label as a set of security tokens. For each analysis exists a set of pre-defined security tokens of a fixed size n.

$$\text{Security Tokens} := \{t_1, t_2, ..., t_n\}$$

The concrete value for n is configurable for each analysis run (see Section 7.2.1). In the above examples we used the token set {*private*, *public*} of size $n = 2$.

The set of all possible security labels is therefore defined as

$$\text{Security Labels} := \mathcal{P}(\text{Security Tokens})$$

The definition of security labels as a set of security tokens is a way to keep the explanations of the security label processing in this master thesis simple. However, the concrete definition of security labels is not the central point and the introduced algorithms are widely independent from it. In fact, only the security label update and conflict detection functions are directly related to this definition. This eases a later change of the definition and reduces the efforts for adjustments in the function implementations. In future work we will move to a academically more accepted definition of security labels, e.g., the definition as a lattice by Denning [6].

5.2. Security Conflicts

During the analysis we propagate security labels through the component black box call graph and detect security conflicts. We define a security conflict as an undesired security label. Thus, a security conflict is a security label which gets reported when it emerges during the evaluation. A set of security conflicts is therefore defined as

$$\text{Security Conflicts} \subseteq \mathcal{P}(\text{Security Tokens})$$

In the exmaple of the previous section we used the security tokens *private* and *public*. When we want to detect flows in which *private* data goes to a *public* output we define the set of security conflicts as $\{\{private, public\}\}$. Recall that security labels are a set of security tokens and we therefore get a set of sets. Here we also see a drawback of our definition of security labels as a set. With the security conflicts specified as $\{\{private, public\}\}$ we detect flows from private sources to public sinks and from public sources to private sinks likewise, though the latter flows are usually deemed as harmless. However, the main focus of this work is on the propagation algorithms and we leave an improvement of the security label definition to future work.

6. Application Analysis

After providing the necessary background and giving the basic definitions we are now ready to introduce the workflows and algorithms of our approach. Our approach performs in three phases:

1. Component black box generation
2. Creation of the component black box call graph
3. Propagation of security labels through the graph of Phase 2 and detection of security conflicts

The first phase runs a component-internal analysis. The goal is to collect information about each existing application component and to create black boxes as an abstraction from the source code. Later phases only work on the basis of these abstractions. The black boxes hold information about input- and output-interfaces, internal data sources and sinks, and component relations, i.e. when and how components can interact.

The second phase explores the relations of application components. The goal is to construct a component black box call graph for later propagation of security labels during the third phase. Starting with the black boxes of all components a user can directly launch from the operating system (in our case Android) the functions of the second phase incrementally search for the black boxes of the respectively callable successor components.

The third and last phase produces the actual result of our approach, i.e. propagates the security labels through the global call graph and detects security conflicts. Here, the component information stored in the black boxes and the call graph from the previous phases are used. Since the call graph from the second phase can grow to exponential size, we apply techniques from the field of product line engineering to reduce the actual analysis space [25].

In the remainder of this chapter, we discuss all three phases in more detail (Sections 6.1 - 6.3). We will motivate each phase, look into the respective

workflows, and discuss their results. We explain the essential steps of the used algorithms and provide examples to demonstrate their functionality. There are several optimisations for the performance of the basic approach, e.g., the combination of Phase 2 and Phase 3 which we cover in Section 6.4. In Section 9 we address the most significant limitations of the current stage of our approach.

6.1. Phase 1: Black Box Generation

In this section, we first motivate the abstraction from application components and discuss the usage of existing tools for the data-flow analysis of application components. Next, we describe the result of Phase 1, the component black boxes and their elements as the abstraction from the component source code, in more detail.

6.1.1. Phase Description

The goal of Phase 1 is to create abstract representations of the components from an application set. For every component, a black box with input- and output-interfaces is created. Input-interfaces represent the program code through which a component can be accessed. Output-interfaces represent the program code with which a component starts another component. Since we are not only interested in the inter-component but the overall data-flow and their security properties, component black boxes can include internal data sources and internal data sinks. A black box also holds the relation information of the black box elements, e.g. whether data might flow from a certain input-interface to a certain internal sink. Component black boxes are described in more detail in Section 6.1.2.

There are several advantages coming with the creation of component black boxes. The first one which was also the initial idea for this design decision is that the subsequent phases of our approach must no longer operate on the actual program source code but can rely on the abstractions. It allows to solely work on component black boxes, their elements and relations, but hides technical details such as how component entry points can be implemented. The abstraction simplifies the view on components to four different types of component elements, namely input-interfaces, output-interfaces, internal data

sources, and internal data sinks. Having only the four element types instead of numerous kinds of different possible source code fragments the abstraction eases the definition of the algorithms of Phase 2 and 3 tremendously. However, care must be taken not to strip too many details from the abstraction in order to preserve a certain degree of precision and reference to the original component code.

The relevant information about application components can be retrieved from two different sources: the application manifest file and the program source code.

Manifest File: For every application there is a mandatory manifest file called *AndroidManifest.xml*. It contains general information about the whole application (e.g. the used API-level) as well as component specific one. Relevant information retrievable from the manifest file is which components are available at all and to whom, i.e. if it is only accessible from within the same application or globally, which permissions restrict the access and so forth.

Source Code: Most of the relevant information can be found in the actual component implementations. When analysing the source code, we must identify the parts through which a component can be accessed (input-interfaces), another component can be called (output-interfaces), where data is created (internal data sources,) or leaves the runtime environment (internal data sinks), respectively. Also important are the actual data-flows within a component, i.e. from where to where data traverses the source code. The flow information is represented as connections of the elements in a black box. Note, that at this point we do not look for inter-component data-flows yet. This task is left to Phase 2. However, we are interested in which data may leave a component through an output-interface.

The retrieval of information about applications and their components is a quite complex task. Though, the analysis of the manifest file is rather straight forward, the inspection of the source code demands a full data-flow analysis. Therefore, we decided to rely on already existing tools specialised in the analysis of data-flows in Android applications, and not to implement Phase 1 ourselves. This shows another advantage of the abstraction through black boxes: Phase 2 and Phase 3 operate only on the basis of the created black

boxes and are otherwise completely independent from the actual information retrieval process and the utilised tools. Hence, we can modify the information retrieval and exchange the tools at any time later without affecting the other phases. We plan to use two different tools: First, FlowDroid [3] developed by Fritz et al., and the second, a tool developed by Daniel Scheckling at the University of Passau[1]. The use of different tools will allow us to experiment with different data-flow analysis strategies and their precision and performance. The integration of the analysis tools is not done yet and is left to future work. Instead, we simulate the data-flow analysis through randomly generated component black boxes. The implementation of the component black box generator is described in Section 7.1.

6.1.2. Component Blackboxes

As motivated in the previous section, we use black boxes as a way to abstract from the source code of application components. In this section we describe the black boxes in more detail. In particular, we will informally and formally define the so called black box elements. A black box elements is either an input interface, an output-interface, an internal data source, or an internal data sink.

Component Black Box: A component black box represents an application component as a unit holding all its relevant information, i.e. the component elements, the defined permissions, and meta-data such as the component identifier. Black boxes are directly created from the manifest file and the respective source code. Since the source code is assumed to be static and thus cannot be changed, black boxes are also static and will never change their structure during the analysis. For example, component elements cannot be added to or removed from a black box, respectively.
Formally, a black box is defined as a 5-tuple of a set of input-interfaces *in*, a set of output-interfaces *out*, a set of internal data sources *src*, a set of internal data sinks *sink*, and some metadata *meta*:

[1]Unfortunately, the tool is still not published and there was no official reference available at the time of writing the master thesis

(in, out, src, sink, meta) with $in \in \mathcal{P}$("Inputs-Interfaces"),
$out \in \mathcal{P}$("Output-Interfaces"), $src \in \mathcal{P}$("Internal Data Sources"),
$sink \in \mathcal{P}$("Internal Data Sinks"), $meta \in$ "Metadata"

Internal Data Source: Component-internal data sources are system API calls which in some way introduce data items to a runtime environment. The API method calls are defined by Android and are therefore all known prior to the analysis. This allows us to statically specify all data sources through annotations. An examples for an API call treated as internal data source is *getDeviceId()* for retrieving the device's identification number (IMEI). The annotations are also used to attach security labels to the API calls. These labels classify the security relevance of the created data, for example they can define whether the source creates data to which access is generally allowed (labelled with *public*) or to which data should be restricted (labelled with *private*).

In the context of our analysis we are interested in where data can flow to. Internal data sources represent program code where data is created. Therefore, we need to know to which sinks data can flow. A sink is either an internal data source or an output-interface from the same component as the internal data source. An internal data source references all directly reachable sinks and additionally holds some metadata such as which API call it represents.

Formally, an internal data-flow is a 3-tuple of a set of data sinks (*sinks*), a security label (*label*), and some metadata (*meta*):

(sinks, label, meta) with
$sinks \in \mathcal{P}$("Internal Data Sinks" $\cup$ "Output-Interfaces"),
$label \in$ "Security Labels", $meta \in$ "Metadata"

Internal Data Sink: Component-internal data sinks are system API calls which in some way expose data items from the runtime environment. As for internal data sources, the method calls are defined by Android, and can therefore be statically annotated prior to the analysis execution. An example for an API call, which is treated as an internal data sink, is *sendTextMessage* for sending SMS. Similar to internal data sources, the annotations are also used to attach security labels to the API calls. This time the interpretation of the labels is the security classification of the target, e.g., the API method *sendTextMessage* could be annotated as a public output because the sent data

leaves the phone and the recipient could be untrusted. Internal data sinks also hold some metadata such as the represented API call.

Formally, an internal data sink is a tuple of a security label (*label*) and its metadata (*meta*):

(label, meta) with *label* ∈ "Security Labels", *meta* ∈ "Metadata"

Input-interface: Component input-interfaces are data entry points to application components. Similar to internal data sources, input-interfaces are certain pre-defined API calls. The difference is that the data origin is yet unknown for an input-interface. This is caused by the fact that at analysis time it is not known which component will concretely access the input-interface at runtime. As an implication the security labels for input-interfaces are not known prior to the analysis in Phase 3 of our approach. An example for an entry point is the *onCreate()*-method of an Activity component.

To know how an input-interface can be reached, we must create an interface description. In Android the information for the description is given through the definition of intent-filters, in particular the Action and Category definitions to which the component can react. In our work, we will only handle Action and Category tags, and do not consider tags such as Data for specifying the MIME-type of the involved data. We leave an extension to future work.

Since input-interfaces deal with inter-component communication, Android's permission system applies to them. More precisely, we must know which permissions are required to be allowed to access the input-interface during runtime. This permission information which is provided in the manifest file is also stored with an input-interface.

Input-interfaces are a special kind of data source. Thus it is relevant where the data handed over to an input-interface flows to. This can be one or more of an output-interface or an internal data sink. Data can be handed over to another component via Android's intent mechanism. So, the information whether an input interface acts as a data source can be extracted from the source code.

An input-interface is formally defined as a 4-tuple of a set of intent-filters (*filters*), a set of required permissions (perms), a set of all directly reachable sinks (*sinks*), and some metadata (*meta*):

(filters, perms, sinks, meta) with
filters $\in$ "Intent-Filters", *perms* $\in \mathcal{P}$("Permissions"),
sinks $\in \mathcal{P}$("Internal Data Sinks" $\cup$ "Output-Interfaces"),
meta $\in$ "Metadata"

Output-interface: Component output-interfaces are data exit points from application components. Similar to internal data sinks, output-interfaces are certain pre-defined API calls. The difference is that the data-flow target is not necessarily known for an output-interface. For Android component exit points are where an intent is created and sent to the operating system to start a component instance. So, a target component is unknown when an output-interface represents source code using an implicit intent and the called successive component is chosen by the user during runtime.

At an output-interface, we must know which components, more precisely which input-interfaces, can be accessed. Therefore, we create a description of the target input-interfaces. We must distinguish between explicit and implicit intents. For explicit intents, we need to extract the concrete name of the target component from the source code. For implicit intents, we must extract the Action and Category specifications for the potential target input-interfaces.

As for input-interfaces, output-interfaces deal with inter-component communication. Therefore, Android's permission system is taken into account. For output-interfaces, it relevant which permissions are granted to an application. We use the permission information stored in the application manifest file.

Output-interfaces implement a special kind of data sink. However, the security label of an output-interface is not necessarily known before analysing. This is in particular the case when data comes from an input-interface for which the security labels are also still not known during Phase 1.

An output-interface is formally defined as 3-tuple of an sent intent (*intent*), a set of granted permissions (*perms*), and metadata (*meta*):

(intent, perms, meta) with *intent* $\in$ "Explicit Intents" $\cup$ "Implicit Intents",
perms $\in \mathcal{P}$("Permissions"), *meta* $\in$ "Metadata"

6.1.3. Example

We will use the same set of application components for the examples of all three phases. The goal is to illustrate the results of the respective phases while

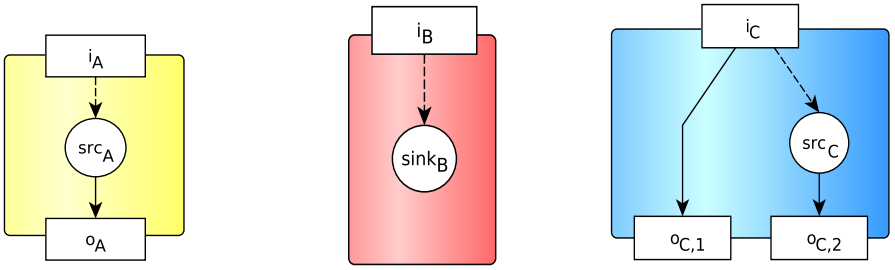
keeping the examples simple enough to easily understand them. Therefore, the used set contains only three components (A, B, and C) of which only one is launchable (component A). Though the example set seems small, we will see that the resulting graphs can grow quite rapidly (especially for examples without merging). This also the reason why we do not use a real world example which are usually much bigger and more complex. Also, we omit some aspects such as the implications of the component access restrictions in applications to fully concentrate on the core functionality of the respective phases. We will now introduce the properties of each component and the structure of its black box representation:

Component A: Component A is the only launchable component in the example set. It can be launched through code represented by the input-interface i_A. There is no direct data-flow from i_A to any other black box element. The component creates an implicit intent with the action flag set to "a". The intent contains data originating from an internal data source src_A. The creation and sending of the intent is represented by the output-interface o_A. There are no security label annotations in component A. The component is shown in Figure 6.1a.

Component B: The second component black box consists of an input-interface i_B and an internal data sink $sink_B$. All data incoming with the intent accessing i_B flows to $sink_B$ which is annotated with the security label *public*. $sink_B$ could, e.g., represent an Android API call to send a message to another device. Component B can be started through an implicit intent with the action flag set to "a". Thus, it can called from the output-interfaces of components A and C. Component B is shown in Figure 6.1b.

Component C: The last component can be started through an implicit intent with action "a". Data sent with the incoming intent at the only input-interface i_C can directly flow to the output-interface $o_{C,1}$. During this flow the data is modified such that when it reaches $o_{C,1}$ it adds the label *private* to the data. Component C has an internal data source src_C which produces data that flows to output-interface $o_{C,2}$. Both of C's output-interfaces represent source code creating an implicit intent with action "a". Thus, from $o_{C,1}$ and $o_{C,2}$ components B and C can be called. The call of a component instance of C is rather unrealistic for real world

applications but allows us to show the issue of loops in the component black box call graphs. Component C is shown in Figure 6.1c.



(a) Component A (b) Component B (c) Component C

Figure 6.1.: Example component black boxes used to illustrate results of all phases

6.2. Phase 2: Creation of Component Call-graph

The second phase of our approach uses the black boxes created during Phase 1 and generates relations between component black box instances. In the following, we introduce an algorithm which creates transitions between output- and input-interfaces to construct a component black box call graph which is needed for Phase 3.

6.2.1. Phase Description

As the initial component set for the algorithm, we use all components, which can be directly launched from the system by a user through clicking the application icon in the application menu. In Android a component is launchable by a user if the application's manifest file contains the code shown in Figure 6.2 in its component specification.

```
1 <intent-filter>
2     <action android:name="android.intent.action.MAIN" />
3     <category android:name="android.intent.category.LAUNCHER" />
4 </intent-filter>
```

Figure 6.2.: Intent-filter XML code to mark an Activity component as launchable

For the black box representation the launch of a component means entering a black box at one of its input-interfaces. As a first step we, must therefore identify these input-interfaces in our initial component set. We store the intent-filters as the description information of the input-interfaces which makes it easy to extract them from the black box set. In practice, the launchable components must be activity components. The launch entry point of an activity is always the *onCreate* method.

Launchable components are in fact not the only components which can be used to start an application. Recall that broadcast receiver is the component type which listens for system-wide messages. As an example, Android sends a broadcast message after it finished the booting process. There are many applications which use this message for a self-launch, e.g., Google's Gmail app[2]. Thus, these broadcast receivers, or at least the components started through them as a reaction on broadcast messages, could also be included in the initial component set. Nevertheless, we will only consider launchable components as the initial set throughout the thesis. We believe that activities provide a sufficient basis for a first prototype implementation and its evaluation.

Starting with the launchable input-interfaces, we perform a breadth first search (BFS) to find first all reachable output-interfaces. From the output-interfaces we continue the search to find all from a directly callable components, more precisely their respectively accessible input-interfaces. Basically, the search for successive components could be done with a depth first search, too. In Phase 3, we will use the principle of BFS to obtain better performance for state merging. Thus, using BFS in the second phase allows us to easier combine both phases as an optimisation of our approach later in Section 6.4. Through the optimisation, the successive components of a component are found on the fly, reducing memory usage and execution time.

[2]https://play.google.com/store/apps/details?id=com.google.android.gm

```
1  connectComponents():
2    Components startComponents = {c|c.isComponent() ∧ c.isLaunchable()}
3    int depth = 0
4    InputInterfaces inputs = ∅
5    foreach c in startComponents do
6      inputs = intputs ∪ c.startInputInterface()
7    od
8    startInputs = inputs
9    while depth < MAX_DEPTH do
10     inputs = findSuccessors(inputs)
11     depth = depth + 1
12   od
13   return startInputs
```

Figure 6.3.: Initial function for constructing the component call graph up to a given call depth

Figure 6.3 shows the function *connectComponents* initialising Phase 2 as pseudocode. In line 2, the set of launchable components is created. The loop in lines 5-7 extracts the set of input-interfaces which are used for launch from the initial component set. In the loop in lines 9-12, the actual search for successors in the component call graphs is executed. For a current set of input-interfaces the function *findSuccessors* determines the respective successor component black boxes, more precisely the input-interfaces through which the respective black boxes are accessed. Each set *inputs* in the loop contains all input-interfaces reachable in a certain common call graph depth. In phase 2, we need the maximum call depth to overcome the existence of loops in the black box call graph and to guarantee termination for our algorithm. Let's assume we have two components A and B where A can call B and B can call A. Without the maximum call depth limitation function *connectComponents* would be stuck in an infinite loop alternating A and B. The concrete value for the maximum call depth is purely heuristic and configuring the search with a sufficient value is not a trivial task. In Phase 3 we will introduce a fix-point detection which obviates the need of the maximum call depth value. However, in Phase 2 we do not yet have the notion of states and therefore do not yet apply fix-point detection. In Section 6.4 we get this feature for Phase 2 as a side effect of the optimisation of combining Phase 2 and 3. Function *findSuccessors* stores the references to the respectively reachable

```
1  findSuccessors(InputInterfaces inputs):
2    InputInterfaces succInputs = ∅
3    foreach InputInterface i in inputs do
4      Component c = i.getComponent()
5      Sources sources = {s|s.isSourceIn(c)} ∪ {i}
6      foreach Source s in sources do
7        foreach OutputInterface o in {o|o.isDirectlyReachableFrom(s)} do
8          InputInterfaces successors = {succ|succ.isInputInterface()
9                                         ∧ succ.isDirectlyReachableFrom(o)}
10         o.successors = successors
11         succInputs = succInputs ∪ successors
12       od
13     od
14   od
15   return succInputs
```

Figure 6.4.: Function for finding and connecting all from a component directly callable components

input-interfaces of the successor component black box instances in the reached output-interfaces. Through this, *findSuccessors* creates the component black box call graph structure with all dependencies. The initial input-interfaces set is returned as the entry points to the graph.

Function *findSuccessors* (shown in Figure 6.4 as pseudocode) receives a set of input-interfaces reached at a certain call depth in the black box call graph. For each of these input-interfaces we retrieve the set of data sources in its black box instance (line 5). The set contains all component internal data sources. The idea is that data-flows originating at these internal sources can be triggered through, e.g., a button click event as soon as the component is started and must therefore be considered in the call graph. We also add the input-interface through which the black box was reached to the data source set. Remember that intents cannot only be used to call other components but also as a container object to transfer data to another component. This data might flow to a data sink, i.e. an internal data sink or an output-interface. So, the transferred data can directly influence the data-flow and the accessed input-interface must therefore be part of the black box call graph. Consequently, data-flows from other input-interfaces of the same component which were not reached must not be considered and are not added to the data source set.

In line 6-13 of Figure 6.4, we process the output-interfaces reachable from a data source of the set created in line 5. For every output-interface, we create the set of directly reachable input-interfaces. In case of an explicit intent, the provided identifier can be directly used to find the matching target input-interface. In case of an implicit intent, we must first look up all input-interfaces matching the provided description. In either case, we additionally check the granted and required permissions to see if access is allow. If the permission check fails, the input-interface is not reachable for the currently handled output-interface. The set of reachable input-interfaces is stored with the respective output-interface as its successors (line 10), and added to the function result set in line 11. The result set forms the set of input-interfaces of the next call depth in the call graph. As described above, the result is returned to function *connectComponents* which decides whether to call *findSuccessors* again, depending on the current call depth.

6.2.2. Example

We will now demonstrate the workflow of Phase 2 by using the example component black boxes from Section 6.1.3. Basically, the example comprises three component black boxes A, B, and C. A represents the only launchable component which creates an implicit intent for calling components B or C. Component black box B has an internal data sink but no output-interfaces. C has an internal data source and two output-interfaces. From both of C's output-interfaces the input-interface B or C can be reached. We will use a maximal call depth of 3 in the example of this section.

In the following, we will explain the basic steps of the execution of Phase 2. The resulting component black box call graph is shown in Figure 6.5.

The very first step of Phase 2 is to identify the set of launchable components and to identify all input-interfaces which are accessed when launched. In our example, this set contains only a single input-interface, the input-interface i_A of component black box A.

Based on the current input-interface set, the next step is to construct the set of relevant data sources for the current call depth. The set of relevant data sources is the currently reached input-interface united with the set of all internal data sources of the input-interface's component black box. In our example this results in the set $\{i_A, src_A\}$. From the set of relevant data sources only one output-interface o_A is directly reachable (reachable through src_A).

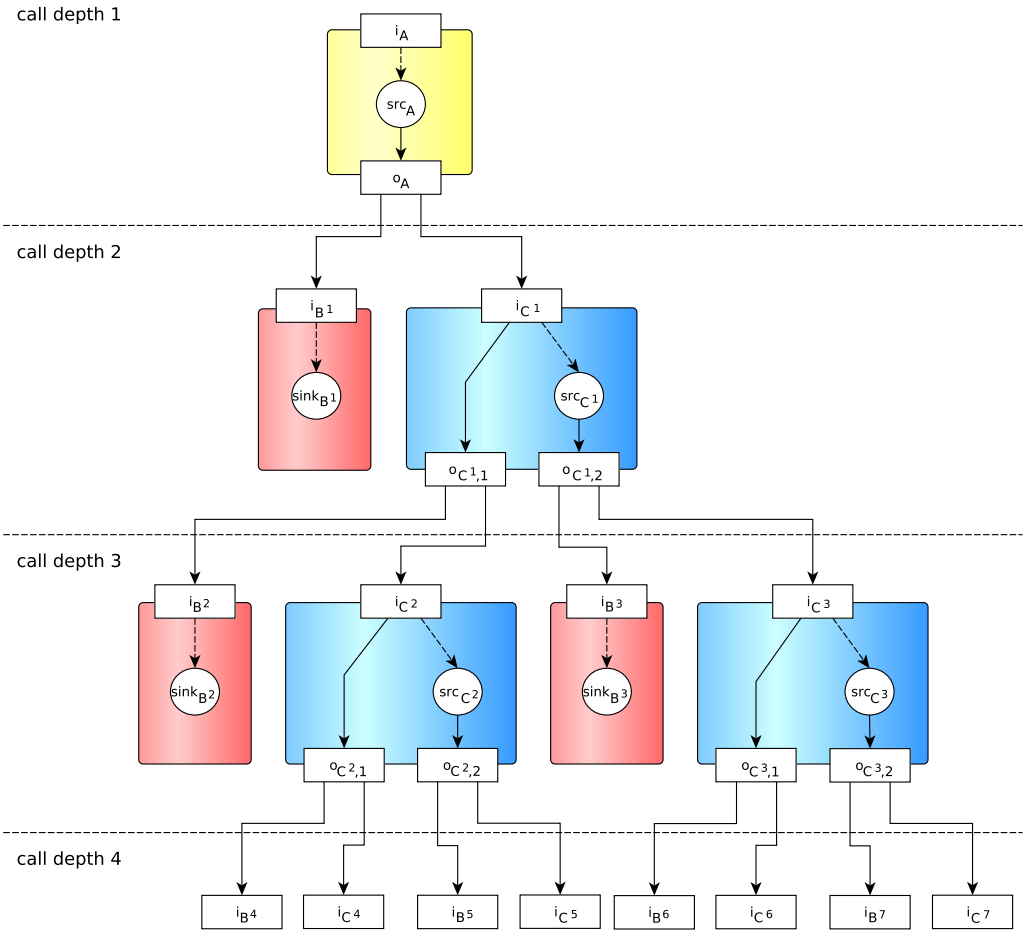


Figure 6.5.: Component black box call graph resulting from executing function *connectComponents* of Phase 2.

Now that we have all reachable output-interfaces, we search for all reachable input-interfaces. This search returns us the set of the input-interfaces of B and C, respectively, i.e. the set $\{i_{B^1}, i_{C^1}\}$. We create new black box instances B^1 and C^1 and connect the output-interface o_A to the input-interfaces of the new black box instances. The numbers in the exponent of the black box identifiers represent the instance identifier.

With the now increased current call depth of 1 we have not yet reached the maximal call depth and we therefore continue constructing the call graph. The

set of relevant data sources is now $\{i_{B^1}, i_{C^1}, src_{C^1}\}$. With this set we retrieve the set of reachable output-interfaces $\{o_{C^1,1}, o_{C^1,2}\}$. From each of the output-interfaces we can both times reach the input-interfaces of component black boxes B and C. For $o_{C^1,1}$ and $o_{C^1,2}$ we create each new successor instances of B and C and get $\{i_{B^2}, i_{C^2}\}$ for $o_{C^1,1}$ and $\{i_{B^3}, i_{C^3}\}$ for $o_{C^1,2}$, respectively. So, the new set of reached input-interfaces is $\{i_{B^2}, i_{C^2}, i_{B^3}, i_{C^3}\}$. Last, we increase the call depth to 2.

We have now a similar input-interface set as with call depth 1. The only difference is that for black boxes B and C we have each two instances instead of a single one. Thus, the computation steps are the exact same ones as for call depth 1 but with a double-sized set. This results in the new input-interface set $\{i_{B^4}, i_{C^4}, i_{B^5}, i_{C^5}, i_{B^6}, i_{C^6}, i_{B^7}, i_{C^7}\}$. The new current call depth is 3 which is also the maximal call depth. So, the algorithm of Phase 2 terminates and we are done constructing the component black box call graph.

6.3. Phase 3: Propagation of Security Labels

The third and last phase uses the security label information collected during Phase 1 to propagate them through the component black box call graph returned by Phase 2. During the propagation of the security labels we will use merging and branching of nodes to reduce the actual graph size. The result of Phase 3 is the set of detected security conflicts as well as the directed graph which was used for the detection. So, basically we transform the black box call graph into a state graph while checking for security conflicts.

In this section, we will first describe which data is observed (Section 6.3.1), define the notion of states for our approach in Section 6.3.2, and finally discuss the algorithms for the security label propagation and security conflict detection in Sections 6.3.3 - 6.3.5.

6.3.1. Propagated Data

The result of our analysis is to know where and with which security properties data might flow beyond application and component boundaries. To be able to achieve this goal, we first need to know from which sources to which sinks data-flows exist. These relations are extracted from source code in Phase 1. Based on this information, Phase 2 then generates the global connection between

components, more precisely the connections of the black boxes representing components. For computing the security properties of component instances and detecting conflicts, we need to propagate the security labels annotated with API calls through the black box call graph. So, we do not observe the flowing data itself but only the security labels associated with them. In fact, we are in no way interested in any of the actual data values but only use their flows as an indicator of their relations. The flows allow us to correctly propagate the security labels.

Though the algorithm presented in this section is fairly independent from the concrete structure of the security labels (only the security label update and conflict detection functions are affected) we assume the definition from Chapter 5 for the rest of this work.

6.3.2. States

In our work, a state is a set of cumulated black box elements with the same security label and call depth. Formally, a state is defined as a tuple of a security label *label* and a set of black box elements *elems*:

$$\textit{(label, elems)} \text{ with } \textit{label} \in \text{"Security Labels"}, \textit{elems} \in \mathcal{P}(\text{"Elements"}) \\ \text{and } \forall e_1, e_2 \in \textit{elems} : \text{callDepth}(e_1) = \text{callDepth}(e_2)$$

We aggregate black box elements with the same security label. So, as soon as we detect a security conflict for a state we automatically know that there must be a conflict for all other elements within the same state, too.

The definition of state requires for all black box elements to have the same call depth. However, the concrete call depth value is not relevant but rather the fact that it is the same for all elements. This implies that after handling a certain call depth, no element can be added to the state. The unchangeability of states allows to have fixed points which are used for a natural termination of Phase 3.

Neither predecessors nor successors are included in the definition of state. The first allows adding of predecessors at any time without changing the sate itself and thus a fixed point analysis through detection of and linking to already existing states. Successors are not included because they are directly derivable from the set of a state's black box elements. Since the set of black box elements can never change after creation, the set of successors can't either.

In our definition of state, we do not require the black box elements to be of the same type. However, all states will have the same element type in practice due to the work flow of our algorithms which handle the black box elements of each type separately.

We define the equivalence of states as follows:

State Equivalence

Two states are equivalent if and only if their security label and their sets of black box elements are equivalent.

Note that for the equivalence of states we do not demand the black box elements to have the same call depth. This allows to globally search for equivalent states within the call graph and to find already reached states, i.e. global fixed points.

6.3.3. Basic Security Label Propagation Algorithm

Phase 3 implements the propagation of security labels through the black box call graph and detects possible security conflicts. It uses the entry points to the graph resulting from Phase 2 as the initial set of states. The main function of Phase 3 is shown in Figure 6.6. After performing an intra-component analysis, it runs a loop alternately performing an inter-component and an intra-component analysis. The loop stops when either there are no more security labels to propagate or a maximal call depth was reached. Security conflicts are detected and reported in the intra- and inter component analysis steps.

The intra-component analysis function propagates security labels from data sources to data sinks, i.e. propagates security labels within component boundaries. Data sources are input-interfaces and internal data sources. Data sinks are internal data sinks and output-interfaces. Since the entry points to the graph of Phase 2 are data sources, the intra-component analysis is called first. We discuss the intra-component analysis function in more detail in Section 6.3.4.

The inter-component analysis function propagates security labels from output-interfaces to the respectively directly reachable input-interfaces. So, the inter-component analysis propagates security labels beyond component boundaries. The function is discussed in more detail in Section 6.3.5.

```
1 securityLabelPropagation(States startInputs):
2   int depth = 1
3   States outputs = intraComponentAnalysis(startInputs)
4   while ((not outputs.isEmpty()) ∧ (depth <= MAX_DEPTH)) do
5     States inputs = interComponentAnalysis(outputs)
6     outputs = intraComponentAnalysis(inputs)
7     depth = depth + 1
8   od
```

Figure 6.6.: Initial function for global security label propagation with max. call depth

The main difference between the intra- and inter-component analysis functions is that in the intra-component analysis function new data-flows can be introduced through internal data sources. When a component is started, a flow from an internal data sources may be triggered through, e.g., clicking a button click event to open a file. These data-flows do not directly depend on the data-flow originating in the input-interface through which the component was reached but are assumed to be startable whenever the surrounding component is accessed. Opposed to that, the inter-component analysis propagates security labels from output-interfaces to input-interfaces which may never introduce new data-flows to the analysis.

The algorithms in Phase 3 use a breadth-first search strategy (BFS) to traverse the call graph. Though basically a depth-first search strategy (DFS) would also be applicable, Rhein et al. [25] argue that in practice BFS performs better than DFS. Based on their claim, we decided to design our algorithms in a BFS fashion.

6.3.4. Intra-component Security Label Propagation

The intra-component analysis function propagates security labels between elements of the same black box component and detects possible security conflicts. The basic idea is to first identify all relevant data sources for each reached component and then to propagate the security labels to the reachable data sinks. During these steps, merges are applied where appropriate.

Relevant data sources are the input-interface through which the component was accessed and all internal data sources of the same component. Any other

input-interfaces are irrelevant since the component was not accessed through them. Thus, all possible in Phase 1 identified data-flows passing these interfaces do not exits in practice for the particular component call. Any data-flow originating from an internal source however is assumed to be potentially started through events such as user interactions, e.g., an button click event. Therefore, all internal sources are included in the set of relevant data sources.

The next step is the propagation of the security labels to all reached data sinks. Data sinks are either internal sinks or output-interfaces. Internal sinks do not have successors. So, whenever an internal sink is reached its security label is updated but the sink itself does not influence the further analysis. Nevertheless, security conflicts can occur at internal data sinks and they must therefore be processed as any other black box element. The security labels of output-interfaces are updated, the resulting states merged, and if an equivalent state never occurred before in the global graph, the state is added to the function output set for further processing.

In Figure 6.7 we show the function *intraComponentAnalysis* implementing the above computation steps. Basically, the function iterates over all input-interfaces of each state. For each through an input-interface reached component black box, the set of relevant data sources is created and iterated over in line 7. All reachable data sinks are updated accordingly in the loop in lines 9 - 17. The implementation of the security label update function (called in line 10) depends on the definition of security labels. Section 5.2 defines security labels as a set of security tokens. Hence, an update of security labels is in our case a union of security token sets. The function *mergeStatesOnLevel* called in line 12 and 15 is the state constructing function. With every iteration of the loop from line 9 - 17 a black box element is added to some state. This means in particular that states constantly change. Therefore the states are still kept local at this point until every data sink was updated. When done so, the states are fully constructed and will never change again. This allows to now check for already existing equivalent states in the global graph, i.e. for fixed points (function calls *mergeStatesGlobally* in lines 21 and 22). Function *mergeStatesGlobally* returns the set of states for which no equivalent already existing state could be found. These states are the actual new ones and are therefore further processed. In line 23 we finally call the function *detectConflicts* which checks the new states for security conflicts. The function depends on the actually used conflict specification which is discussed in Section 5.2. Function *detectConflicts* does not interrupt the analysis but only records security conflicts.

The set of output states is the return value of *intraComponentAnalysis*.

```
1  intraComponentAnalysis(InputStates inputs):
2    OutputStates outputs = ∅
3    InternalDataSinkStates sinks = ∅
4    foreach InputState state in inputs do
5      foreach InputInterface i in state do
6      Component c = i.getComponent()
7        foreach Source src in {src|src.isInternalSourceIn(c) ∨ (src=i)} do
8          Sinks sinkElems = {s|s.isDirectlyReachableFrom(src)}
9          foreach Sink s in sinkElems do
10           s.updateSecurityLabel(src.securityLabel)
11           if (s ofType OutputInterface) then
12             outputs = mergeStatesOnLevel(outputs, s)
13           fi
14           if (s ofType InternalDataSink) then
15             sinks = mergeStateOnLevel(sinks, s)
16           fi
17         od
18       od
19     od
20   od
21   outputs = mergeStatesGlobally(outputs)
22   sinks = mergeStatesGlobally(sinks)
23   detectConflicts(outputs ∪ sinks)
24   return outputs
```

Figure 6.7.: Function implementing the intra-component security label propagation including conflict detection

6.3.5. Inter-component Security Label Propagation

The inter-component analysis function propagates security labels between black boxes, i.e. from output-interfaces to input-interfaces. While propagating, it detects possible security conflicts.

The function *interComponentAnalysis* which implements the inter-component analysis is shown in Figure 6.8. It is similar though less complex as the pre-

viously discussed *intraComponentAnalysis* function. So first it iterates over the output-interfaces of each state. Each from an output-interface reachable input-interfaces is updated in line 6. As for *intraComponentAnalysis*, the update function depends on the definition of security labels is therefore a union of security token sets. Also, as long as input-interfaces are still update states are under constant change. Function *mergeStatesOnLevel* therefore constructs and updates the states locally. After updating all security labels, we globally look for equivalent states and merge them accordingly (function call *mergeStatesGlobally* line 11). The set of unmergeable states is then checked for security conflicts in line 12 before returning the set as the result of *interComponentAnalysis*. We already discussed functions *mergeStatesOnLevel*, *mergeStatesGlobally* and *detectConflicts* in the previous subsection, and therefore kept the explanations quite short to avoid redundancy.

```
1  interComponentAnalysis(OutputStates outputs):
2    InputInterfaces inputs = ∅
3    foreach OutputState state in outputs do
4      foreach OutputInterface out in state do
5        foreach InputInterface i in {i|i.isReachableFrom(o)} do
6          i.updateSecurityLabel(out.securityLabel)
7          inputs = mergeStatesOnLevel(inputs, i)
8        od
9      od
10   od
11   inputs = mergeStatesGlobally(inputs)
12   detectConflicts(inputs)
13   return inputs
```

Figure 6.8.: Function implementing the inter-component security label propagation including conflict detection

6.3.6. Example

We continue the example from Phase 1 and 2 to demonstrate the workflow of Phase 3. In the example of Phase 1 we introduced three fairly simple

component black boxes (Section 6.1.3). In the example of Phase 2 we re-used the black boxes to construct the black box call graph with a maximal call depth of 3 (Section 6.2.2). In the example of Phase 3 we will use this call graph to propagate security labels and to eventually detect security conflicts within the graph.

The annotations of the security labels were already integrated in the example black boxes in Phase 1. We used the security token set with elements *private* and *public*. Security conflicts were generally defined as a set of unwanted security labels (Section 5.2). In our example we want to detect all security labels which contain both, the *private* and the *public* token. Thus, we specify the security conflict set as follows $\{\{private, public\}\}$. As in the example of Phase 2, we again use a maximal call depth of 3. We will see however that this time the algorithm terminates by itself thanks to the fixed point detection.

We will now sketch the execution of the functions introduced for Phase 3 but do not cover the whole execution in full detail which would be far to extensive for the purpose of an example. The result of applying Phase 3 to the black box call graph from the example of Phase 2 is shown in Figure 6.9.

The main function of Phase 3 *securityLabelPropagation* (Figure 6.6) expects the entry points of the resulting call graph of Phase 2 as states. In our example, there is only the single graph entry point i_A and therefore also only a single start state. The first function call of *interComponentAnalysis* in line 5 in Figure 6.6 returns a set of states of input-interfaces. Though there are two input-interfaces i_C and i_B reachable at this point, the return set only contains a single state. However, this state encapsulates the two input-interfaces because both share the same security label. This is a situation in which VarDroid merges black box elements to a single state. VarDroid continues the analysis by propagating the security labels to the successors of i_C and i_B. An example for state splitting is given in the subsequent call of *intraComponentAnalysis* in line 6 of Figure 6.6. From i_C we reach the black box elements $o_{C,1}$ with security label $\{private\}$. Via the internal data source src_C in component C we reach element $o_{C,2}$ with both times empty security label. From i_B we reach the internal data sink $sink_B$ with security label $\{public\}$. So, all black box elements have a different security label or are of a different element type. We must therefore create four separate states during the execution of *intraComponentAnalysis*. Phase 3 also features a fixed point detection which for our approach means that already existing equivalent states are detected. From the just created state containing output-interface $o_{C,2}$ we can reach the input-

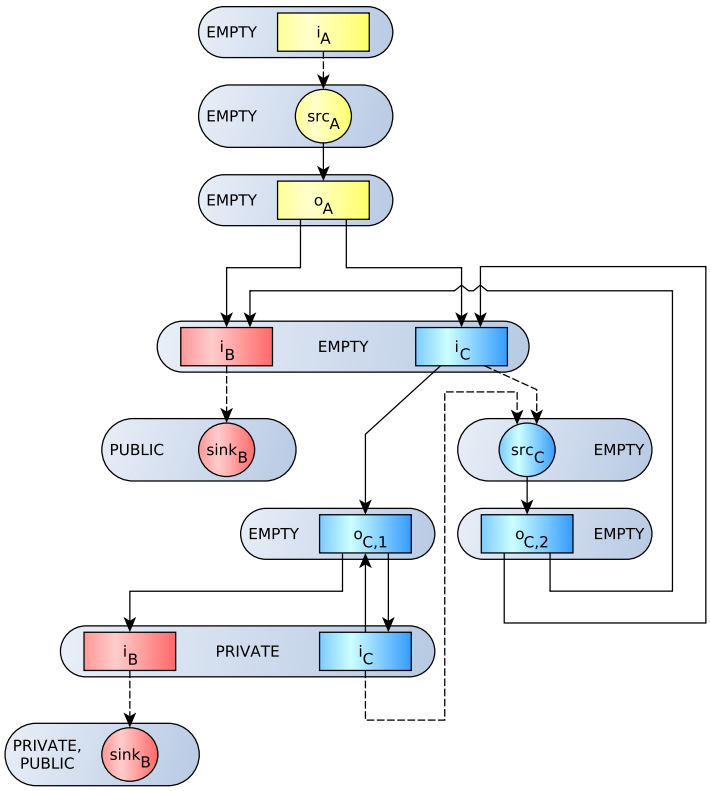


Figure 6.9.: State Graph resulting from applying functions of Phase 3 to black box call graph of example in Phase 2 (Section 6.2.2)

interfaces i_C and i_B during execution of the *interComponentAnalysis* function with both times empty security label. The for the input-interfaces created common state was already created when calling *interComponentAnalysis* for the very first time. Thus, the analysis reached a fixed point and the state must not be considered in the remaining analysis.

6.4. Optimisation

In this section we describe how we can combine Phase 2 (Section 6.2) and Phase 3 (6.3) to a single pass analysis to improve the overall performance. In the following we first motivate the optimisation and then show its practical

work flow.

6.4.1. Motivation

The purpose of Phase 2 is to find all from an application component callable components. More precisely, Phase 2 finds all from the output-interfaces of a component black box instance reachable input-interfaces of other component black box instances. Thus, the result of Phase 2 is a black box call graph with up to exponential size, limited only by the maximal call depth. The maximal call depth was introduced to guarantee termination.

Proof. As a simplification we assume a single input-interface and a single output-interface per component black box. The number of different black boxes n is fixed for an analysis run. If from every output-interface all other black boxes can be called, every output-interface has n successors. The parameter for the height of the black box call graph is the maximum call depth d. Therefore, the size of the resulting call graph is $\mathcal{O}(\sum_{i=1}^{d} n^i) = \mathcal{O}(n^d)$. □

Through merging of black box elements in Phase 3, we reduce the analysis space, i.e. the black box call graph, whenever possible. Everytime two black box elements are merged, the subsequent analysis only has to traverse a single subgraph instead of two. Thus, the redundant subgraph was created unnecessarily during Phase 2. Note that the construction of states in Phase 3 does not actually reduce the analysis space since all black box elements are kept in the states and must be individually checked for successors. The state representation only reduces the number of nodes in the resulting graph.

Another implication of separate Phases 2 and 3 is the fact that the black box call graph is traversed twice: once for the graph creation, once for the security label propagation. Especially in the first case the graph with up to exponential size is traversed at any rate. For the latter, the graph is potentially smaller due to merging (though the exponential size is still possible). Obviously, traversing a graph twice is less performant as a single traverse.

Given the above issues, we will combine Phase 2 and 3 to improve the runtime performance of our approach. For this, we will integrate Phase 2 in Phase 3 to only search for black box element successors when needed.

6.4.2. Optimisation Description

The idea of combining Phase 2 and 3 is to execute an on-demand successor search. The successors for input-interfaces and internal sources directly depend on the respective component they belong to and are therefore directly known through the black box representation. Trivially, internal sinks do not have successors at all. The successor black box elements of output-interfaces on the other hand are the respectively callable input-interfaces. The goal of Phase 2 is to find these input-interfaces. Therefore, we must search for successors only when continuing the analysis after reaching output-interfaces.

Technically spoken, we extend the base algorithm of Phase 3 (Figure 6.6) such that before every execution of the inter-component analysis function the successive input-interfaces of the reached output-interfaces are found. Also, Phase 2 identifies the set of launchable components and filters for their start-up input-interfaces (see algorithm in Figure 6.3). This functionality is added to the beginning of the combined algorithm.

```
1  securityLabelPropagation():
2    Components startComponents = {c|c.isComponent() ∧ c.isLaunchable()}
3    InputInterfaces startInputs = ∅
4    foreach c in startComponents do
5      startInputs = startInputs ∪ c.startInputInterface()
6    od
7    int depth = 0
8    States outputs = intraComponentAnalysis(startInputs)
9    while ((not outputs.isEmpty()) ∧ (depth < MAX_DEPTH)) do
10     outputs = findInputInterfaces(outputs)
11     States inputs = interComponentAnalysis(outputs)
12     outputs = intraComponentAnalysis(inputs)
13     depth = depth + 1
14   od
15   return startInputs
```

Figure 6.10.: Optimised initial function for global security label propagation with max. call depth

We show the new algorithm in Figure 6.10. Lines 2 - 6 implement the

search for all start-up input-interface of the launchable components. Function *findInputInterfaces* which handles the successor search is inserted in line 10. The rest of the function is the same as in the original function shown in Figure 6.6.

7. Implementation

In this chapter we present the current prototype implementation of the phases introduced in Chapter 6. We will first describe the tool *ComponentGenerator* for randomly generating component black boxes (Section 7.1. With this tool we simulate the analysis of application components and the black box generation. Until we replace the ComponentGenerator with analysis tools for real world applications, it enables us to execute first test and evaluation runs. Second, we describe in Section 7.2 the status of the actual prototype implementation of our approach called *VarDroid*. We explain its basic architecture and how it can be configured for running in different execution modes.

7.1. ComponentGenerator

The *ComponentGenerator* is a tool for generating random application component black boxes and storing them in an XML file. In the following we will describe in more detail how the black boxes are created, which configuration options exist, and the used XML output format.

7.1.1. Component Generation

ComponentGenerator creates sets of application component black boxes with random internal structure. The produced sets are meant for testing VarDroid and getting performance results without having to analyse real world applications. Therefore, the black boxes only reflect the basic structure of real components and do not contain every feature possible for real components. For example, when creating an action tag for an input-interface as can be defined in the AndroidManifest.xml, ComponentGenerator uses generic names such as *Action1*, but does not use real action names, e.g., the in Android predefined action *ACTION_CALL* for intending to make phone calls. The only exception is when creating an input-interface which can be directly started by the

user through clicking the application icon. Here, the actual predefined action *ACTION_MAIN* is used. The same applies for categories.

Other component related features are not included at all. For example, there are no associations for components which would group components to an application. This implies that the generated test sets do not allow to fully test for security aspects related to the application sandbox. Another missing feature is, e.g., the Extra tag which can be defined for intent-filters in the manifest file. However, ComponentGenerator is meant as a test set generator and hence does not need to generate all aspects of real world application components.

Besides the existing limitations, ComponentGenerator allows to quickly generate huge test sets in which the internal structure of all component black boxes depends on certain probabilities. These probabilities can be set in a configuration file. Amongst others it can be specified how many elements of each type a component contains and with which likelihood there is a data-flow from one element to another. Generally, there can be only data-flow transitions from data sources to data sinks. However, there is no limit on how many outgoing or incoming transitions exist, respectively (though there can be maximal only one of each possible data source-data sink pair)

7.1.2. Program Arguments

ComponentGenerator expects two mandatory runtime arguments: the path to the configuration file, and the path to the output file. The configuration file path is denoted with a preceding *-config*, the output file path with a preceding *-o*. The structure of the configuration and output file are described in the following sections.

7.1.3. Configuration Options

We will now introduce all configuration options for ComponentGenerator. Each of the options are mandatory for definition in the configuration file but the order is arbitrary. The tool expects a new line for each option and every line in the format "*option* = *value*". Empty lines are allowed. We use a uniform probability for the decisions and the selection of concrete values.

An example configuration file which we used for our evaluation is given in the Appendix A.

components: the total number of component black boxes. The value must be an integer and at least one.

minInputs: the minimum number of input-interfaces. The value must be an integer and at least one. It also must be less or equal *maxInputs*.

maxInputs: the maximum number of input-interfaces. The value must be an integer and at least one. It also must be equal or greater than *minInputs*.

minOutputs: the minimum number of output-interfaces. The value must be a non-negative integer and less or equal *maxOutputs*.

maxOutputs: the maximum number of output-interfaces. The value must be a non-negative integer and equal or greater than *minOutputs*.

minSources: the minimum number of internal data sources. The value must be a non-negative integer and less or equal *maxSources*.

maxSources: the maximum number of internal data sources. The value must be a non-negative integer and equal or greater than *minSources*.

minSinks: the minimum number of internal data sinks. The value must be a non-negative integer and less or equal *maxSinks*.

maxSinks: the maximum number of internal data sinks. The value must be a non-negative integer and equal or greater than *minSinks*.

probInputSink: the probability that a transition from an input-interface to an internal data sink is created. The value must be a floating point number in the interval from zero to one, while allowing the values zero and one.

probInputOutput: the probability that a transition from an input-interface to an output-interface is created. The value must be a floating point number in the interval from zero to one, while allowing the values zero and one.

probSourceOutput: the probability that a transition from an internal data source to an output-interface is created. The value must be a floating point number in the interval from zero to one, while allowing the values zero and one.

numActions: the total number of different actions generally available for intent-filters. Actions can only be set for implicit intents. The value must be a non-negative integer.

probExplicitIntent: the probability that an output-interface represents source code of an explicit intent. The probability for an implicit intent is automatically set to "1−*propExplicitIntent*". The value must be a floating point number in the interval from zero to one, while allowing the values zero and one.

numPermissions: the total number of different permissions generally available which can be required for the access to input-interfaces or granted to black box interfaces, respectively. The value must be a non-negative integer.

probGrantPermission: the probability that a permission is required at an input-interface. The value must be a floating point number in the interval from zero to one, while allowing the values zero and one.

probRequirePermission: the probability that a permission was granted to a component. The value must be a floating point number in the interval from zero to one, while allowing the values zero and one.

numSecurityLabels: the total number of generally available security labels. The value must be a non-negative integer.

probInputSecLabel: the probability that a security label is annotated to an input-interface. The value must be a floating point number in the interval from zero to one, while allowing the values zero and one.

probSourceSecLabel: the probability that a security label is annotated to an internal data source. The value must be a floating point number in the interval from zero to one, while allowing the values zero and one.

probSinkSecLabel: the probability that a security label is annotated to an internal data sink. The value must be a floating point number in the interval from zero to one, while allowing the values zero and one.

probOutputSecLabel: the probability that a security label is annotated to an output-interface. The value must be a floating point number in the interval from zero to one, while allowing the values zero and one.

7.1.4. XML Output Format

The output created by ComponentGenerator is a file in XML format. An example snippet from an output file is shown in Figure 7.1. The example contains all allowed XML elements though in some cases arbitrary many are allowed, i.e. elements could be left out or multiple elements are valid.

Overall, the file includes at least one element *component*, representing a component black box. The component name acts as the identifier and thus must be unique within the component set. A component can have various elements: *input-interfaces*, *internal-sources*, *internal-sinks*, *output-interfaces*, *relations*, and *granted-permissions*. The order of these elements is arbitrary. Except input-interfaces, all other elements are optional.

The XML element *input-interfaces* is mandatory and inlcudes at least one *input* element with an component internal unique identifier and the flag if the intput-interface represents the entry point to the component launching the application. The input element can have a *description* element representing the Android intent-filter, and an element *required-permissions* listing the for this input-interface used permissions.

The elements *internal-sources* and *internal-sinks* are fairly similar. The first one contains all internal source definitions (*source* elements), whereas the second contains all internal sink definitions (*sink* elements). The source and sink elements have each a component internal unique identifier and optionally specifies the annotated security label (*label* element).

The *output-interface* has a component internal unique identifier. Additionally, it must specify if the target component(s) are called implicitly or explicitly. The example in Figure 7.1 shows an explicit call. An implicit call can be defined in the same way as the description for input-interface "in1" in the example.

The *relations* element lists the transitions between black box elements in the *relation* elements. Recall, that only transitions from data sources to data sinks are valid.

Last, the *granted-permissions* element lists the permissions granted to an Android component as specified in the manifest file (*permission* elements).

```
<components>
  <component name="Comp42">
    <input-interfaces>
      <input name="in1" startup="false">
        <descriptions>
          <implicit>
            <action name="android.intent.action.MAIN"/>
            <category name="android.intent.category.LAUNCHER"/>
          </implicit>
        </descriptions>
        <required-permissions>
          <permission name="p2"/>
        </required-permissions>
      </input>
    </input-interfaces>
    <internal-sources>
      <source name="src1">
        <label conf="L2" />
      </source>
    </internal-sources>
    <internal-sinks>
      <sink name="sink2">
        <label conf="L0" />
      </sink>
    </internal-sinks>
    <output-interfaces>
      <output name="out1">
        <descriptions>
          <explicit target="Comp23"/>
        </descriptions>
      </output>
    </output-interfaces>
    <relations>
      <relation   input="in1"   sink="sink0"/>
      <relation   input="in2"   sink="sink0"/>
    </relations>
    <granted-permissions>
      <permission name="p1"/>
    </granted-permissions>
  </component>
  ...
</components
```

Figure 7.1.: Example XML output snippet created by ComponentGenerator

7.2. VarDroid

In this section we introduce VarDroid. VarDroid is the prototype implementation of the overall approach presented in this master thesis. Its basic design is shown in Figure 7.2. Its core module is the so called Propagation Handler which implements the security label propagation functions of Phase 3. The Propagation Handler retrieves the black box component call graph from the Component Connector which implements the functions of Phase 2. The call graph can either be requested completely or stepwise as described in Section 6.4. The Propagation Handler also calls the Conflict Detector at certain points to check for security conflicts. All three modules can be configured in many ways. A user of VarDroid can specify the Component Connoector's source file for the black boxes, the file containing the security conflict specification, and the execution modes of the Propagation Handler.

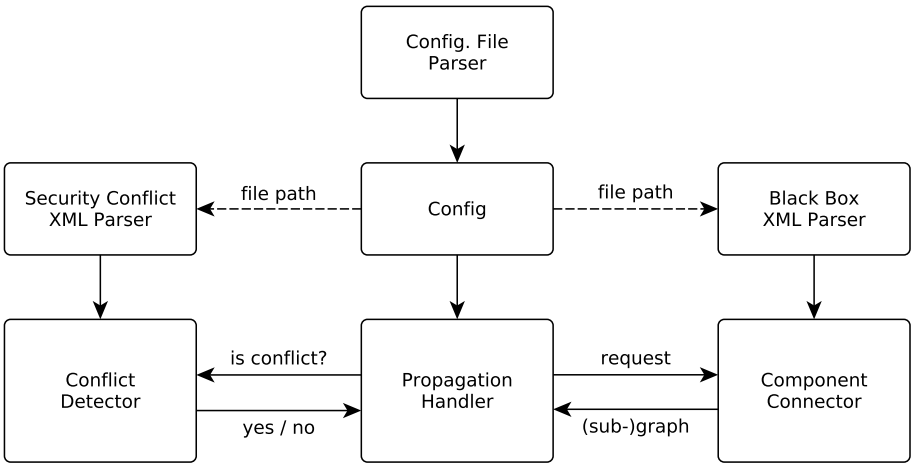


Figure 7.2.: VarDroid implementation architecture

In the remainder of this chapter, we first describe the available execution modes and configuration options (Section 7.2.1). Afterwards, we give an overview about the implementations of the three computation phases (Sections 7.2.2 - 7.2.4). Last, in Section 7.2.5, we explain the implementation of the

conflict detection mechanism.

7.2.1. Execution Modes and Configurations

One of the design goals was to make VarDroid as easily evaluable and comparable in the different execution modes as possible. Therefore, VarDroid provides various configuration options which are specified in a configuration file. All options are mandatory. VarDroid uses a single configuration file which location must be given as the only runtime argument. The command for running the VarDroid jar-file is:

java -jar "path/to/VarDroid.jar" "path/to/configuration/file"

The order of the configuration options in the file is arbitrary though there must be a new line for each option. Space lines are allowed. Each configuration option must be defined in the following format:

"option name" = "value"

In the following, we will explain all execution modes and show how to practically configure them.

Component Call Graph Generation There are two ways of creating the component call graph. The first one as described in Section 6.2 is to separately construct the full component call graph up to a certain call depth and then propagate the security labels based on this graph. The other way is the combination of the second and third phase and to construct the call graph on demand as described in the optimisation in Section 6.4. VarDroid allows to switch between the two strategies to find the successive component black boxes. Further details on the concrete implementation of the component call graph generation can be found in Section 7.2.2.

The desired construction behaviour is specified through the configuration option *findSuccessor*. The only valid values are either *separate* for the unoptimised or *combined* for the optimised component call graph creation.

Maximal Call Depth If the separate component call graph construction is chosen it is possible that call loops occur which lead to non-terminating behaviour. We overcome this issue by introducing a maximal component call

depth, i.e. artificially limiting the size of the graph. If the value *combined* is set for the configuration option *findSuccessor*, the maximal call depth simply ignored.

The maximal call depth is defined through the option *callDepth*. The value must be a non-negative integer. If the call depth is 0, VarDroid never searches for black box successors. If it is set to 1, VarDroid performs the search once and so forth.

Merge Modes VarDroid implements different merging strategies for better comparison of the practical performance. Besides the global merging of states as described in Section 6.3, VarDroid also allows to run the analysis without globally merging states. Other modes only focus on merging black box elements either globally or only on level-basis. It is also to not merge at all. More details on the concrete implementations of the different merge modes can be found in Section 7.2.4.

The configuration option for selecting the merge mode is *mergeMode*. It expects one of the following values for the respective above mentioned merge modes: *state-global, state-level, elem-global, elem-level, none*

Component Black Box Definition File As stated earlier in Section 6.1, VarDroid does not implement the component black box generation itself but rather relies on specialised external tools to perform this task. The integration of a data-flow analysis tool is currently not done yet. Therefore, we use the tool ComponentGenerator which produces a file with XML-formatted random component black boxes. ComponentGenerator is introduced in Section 7.1.

The path to the concretely used XML file containing the black boxes must be given as the value of the configuration option *input*.

Conflict Definition File The purpose of VarDroid is to detect conflicts in the security labels occurring in the component call graph. However, it must be defined what conflicts actually are. The path to the XML formatted file containing this definition is given as the value of the configuration option *conflicts*. The concrete implementation of conflict detection is discussed in Section 7.2.5.

Logging There are three logging types available for VarDroid: debugging, warning and error. Log output is produces in various situations in the source code. The output can be enabled or disabled for each logging type individually.

The configuration option *log* specifies for which logging type output is produced. Valid values are the characters *d, w* and *e*. To enable logging types, the respective characters are concatenated in arbitrary order, i.e. *log = dew* is equivalent to *log = wde*. If a logging output is unwanted, the respective character is left out. To fully disable logging, use *none* instead of the characters.

Graphical Output VarDroid provides the feature to create a graphical representation of the constructed component call graph of Phase 2 and the merged graph after Phase 3. If not present, a folder named "output" is automatically created in the current working directory. The graphs are written to separate files in the "output" folder. The files are created in the graph description language DOT [20]. Be aware that the size of the graphs can grow rapidly. Therefore, it is recommended to only use this feature when very small component black box sets are analysed.

The graphical output can be enabled or disabled by setting either a 1 or a 0 as the value of the configuration option *dot*, respectively. Additionally, it is possible to enable or disable transition labels. The values 1 enables and 0 disables the option *labels*, respectively. If the graphical output is disabled, the value for *labels* is ignored.

7.2.2. Blackbox Generation (Phase 1)

As explained in Section 6.1, VarDroid does not implement the component black box generation itself but relies on specialised tools, e.g. FlowDroid[3], to fulfil this task. However, in this master thesis we want to focus on the algorithms for security label propagation and conflict detection as described in Sections 6.2 and 6.3, respectively. Therefore, we use the tool ComponentGenerator, introduced in Section 7.1, to generate random component black boxes. ComponentGenerator writes its results to an XML formatted file. So, at this point VarDroid only needs to parse the component black boxes from file and to create the black boxes in their Java object representation for further processing.

7.2.3. ComponentConnector (Phase 2)

VarDroid implements Phase 2 of our approach (Section 6.2) in the abstract and generic ComponentConnector Java class. It defines all methods for either executing Phase 2 separately or in combination with Phase 3 (cf. Section 6.4). Thus, it allows to either construct the full component call-graph up to a specified call depth or to only find successive component black boxes stepwise.

ComponentConnector also supports the different merging modes which is either black box element based or state based. More precisely, we inherited two subclasses from ComponentConnector, namely ElementConnector and StateConnector, which implement the modes. However, the ComponentConnector implementations do not differentiate between level or global based merging. This is due to the purpose of finding successive black boxes which is the same for level and global merging modes.

A ComponentConnector is a passive module meaning that it never triggers any actions itself but rather waits for a PropagationHandler instance (see Section 7.2.4) to call it respecting the current execution mode.

7.2.4. PropagationHandler (Phase 3)

The propagation handler implements the propagation of security labels and the different merging modes. It is responsible for calling the component connector accordingly, i.e. either for a separate or combined construction of the black box call graph (Section 6.4). The propagation handler also calls the conflict detector during the propagation process to allow on-the-fly conflict detection. The conflict detector is explained in Section 7.2.5.

The different merging modes are in fact implemented separately. So, we provide an implementation for all of the following merge modes:

None This implementation provides no merging at all. It uses the full black box call graph with the up to exponential size. This mode was implemented to demonstrate the basic problem motivating this work.

Black Box Elements (level merge) In this mode, the Propagation Handler propagates security labels while only merging black box elements with the same call depth. This means in particular that there is no fixed point detection and hence, we need a maximal call depth to ensure termination. Also, there is no notion of state.

Black Box Elements (global merge) Similar to the level-based element merging, this mode only merges black box elements but this time on the overall call graph. Therefore, we have a fixed point detection implemented in this mode. Again, there is no notion of state.

States (level merge) This mode is based on the level based black box element merge mode. However, this implementation introduces states. So, additionally to the merge of black box elements, the resulting elements are grouped to states complying the definition of Section 5.1. Since there is no fixed point detection, a maximal call depth is required to ensure termination.

States (global merge) Eventually, the last mode implements Phase 3 exactly as described in Section 6.3. Thus, it extends the previously introduced mode by additionally merging states globally, i.e. detecting whether the exact same state occurred before.

VarDroid can be configured to run in one of the above merging modes with either separate or combined construction of the black box call graph. We give more details on the configuration options in Section 7.2.1. We tried to implement the propagation handlers as similar as possible to allow a expressive comparison of the different modes. The evaluation and comparison is presented in Chapter 8.

7.2.5. ConflictDetector

The conflict detector implements the decision point that checks security labels for certain security conflicts. Besides the merging functions, it is the only module in VarDroid that depends on the definition of security labels and security conflicts. Currently, the conflict detector implements the definition given in Section 5.2.

The concrete security conflicts must be pre-defined in an XML formatted file. The path to this file is specified in the runtime configuration file as described in Section 7.2.1. Security conflicts are basically certain unwanted security labels. This representation is reflected in the XML structure of the specification file. Figure 7.3 shows an example set of security conflicts specifying security labels $\{L0, L1\}$ and $\{L0, L2\}$ as unwanted. Similar to the given example, arbitrarily many security conflicts of arbitrary size can be specified in general.

```
<conflicts>
    <conflict>
        <label conf="L0" />
        <label conf="L1" />
    </conflict>
    <conflict>
        <label conf="L0" />
        <label conf="L2" />
    </conflict>
</conflicts>
```

Figure 7.3.: Exemplary security conflict specification in XML format

The conflict detector implements only the decision point which implies it does not know anything about the analysis algorithms itself. Therefore, the propagation handler is responsible to call the conflict detector in appropriate situations. Currently, this is whenever a security conflict is updated. However, the separation of the conflict detector from the actual analysis algorithms allows to keep the propagation handler independent from the definition of security labels and security conflicts.

8. Evaluation

The (though yet far away) goal of our approach is to be able to analyse a whole application store. Therefore, it is important to measure the performance of VarDroid right from the beginning. We executed several experiments to see how VarDroid scales in general, and how it performs compared to other merging strategies, in particular to the naive execution with no merging at all and to component element merging, i.e. merging without the notion of states. In our experiments we focused mostly on execution times.

In this chapter we will discuss the evaluation of our prototype implementation VarDroid. We first describe the execution environment and configuration and then report the evaluation results. We conclude this chapter by interpreting the results and derive implications and consequences to improve our approach in future work.

8.1. Execution configuration

We performed all experiments with an Ubuntu 12.04 operating system on a Dell PowerEdge R715 with 2.8 GHz and 128 GB RAM. However, we limited all JMVs to a maximum of 32 GB which was only exceeded twice[1]. VarDroid implements five analysis modes as described in Section 7.2.4: no merging, level-based black box element merging, global black box element merging, level-based state merging, and global state merging. For each of these modes we executed analyses on component black box sets in the range from 100 to 2000 components. We increased the set sizes stepwise with 100 black boxes. For every set size we repeated each experiment ten times (ten different sets for each size) to get more expressive mean values, e.g., for the execution times.

[1]For the memory exceedances we did not count runs without any kind of merging since they all ran out of memory after only few call depths

The black box component sets were all randomly generated by the ComponentGenerator introduced in Section 7.1. The used configuration file is provided in the appendix [2].

In our experiments there are three different security tokens *L0, L1, L2*. Though we propagate these labels through the data-flow graph and update the black box elements appropriately, security conflicts play a secondary role in our experiments. Therefore, we used a rather simple security conflict set shown in Figure 8.1. Instead of evaluating the set of detected conflicts, we focus on the overall execution performance of VarDroid. All experiments were

```
1 <conflicts>
2   <conflict>
3     <label conf="L0" />
4     <label conf="L1" />
5     <label conf="L2" />
6   </conflict>
7 </conflicts>
```

Figure 8.1.: Security conflict set used for the experiments

exclusively executed with the optimisation as described in Section 6.4, i.e. with combined black box connecting and security label propagation.

8.2. Evaluation Results

In this section we will show the evaluation results. We evaluated the approach regarding the needed maximum call depths, and the graph sizes, and the overall execution times. Though we intended to compare our results with a naive analysis, i.e. without any kind of merging at all, we excluded it from the resulting statistics. The reason is that with a component black box set of only 100 we always ran out of memory after a call depth of only 5 even though we

[2]The appendix shows the configuration file for 2000 components. Though this number varies for the different experiments, the rest of the file stays the same for all experiments.

doubled the JVM's available memory to 64 GB. Though this fact is interesting by itself the obtained numbers are not expressive enough for any statistic especially when it comes to comparing them to the other execution modes.

Besides for the no-merging mode we also encountered two memory exceedances when performing analyses in the level-based element merge mode. Both times the black box set was of size 2000. Once it occurred at the call depth of 65 the second time at the call depth of 100. The reason for this is the immense graph size which almost linearly grows for the level-based element merging. We discuss the graph size evaluation in more detail in Section 8.2.2.

In VarDroid we use the hash values of states for the equivalence comparison. For the computation of the hash value we use the states security label and the contained black box elements which matches exactly the definition of state equivalence as given in Section 6.3.2. The usage of hash values is a simple trick to reduce the comparison of two states to a simple numeric one. However, there is the possibility of hash collisions, i.e. the situation in which the hash codes are equal without their states being equivalent. In all the evaluation runs we encountered in total only four collisions[3]. Though this is an acceptable rate, every collision distorts the final result and therefore in future work we must look into ways to reduce the chances for collisions even further.

8.2.1. Graph Call Depth

The evaluation of the needed call depths was motivated by the question how fast the algorithms terminate in terms of the graph size and fixed point detection. As we will see later, the call depth is directly related to the graph size and execution times. We show the result of the call depth evaluation in Figure 8.2.

For all performed evaluations we set the maximum call depth to 100. Since the level-based execution modes of VarDroid do not implement fixed point detection their maximum call depth is trivially always exactly 100. Because of the same graph depth, their behaviour is in general fairly similar for the black box element count as well as for the overall execution time which we both discuss in the following subsections.

The call depth of the two modes implementing global merging are relatively low with a mean of 4.25 for the global element merging and a mean of 31

[3]again without taking the no-merge execution mode into account

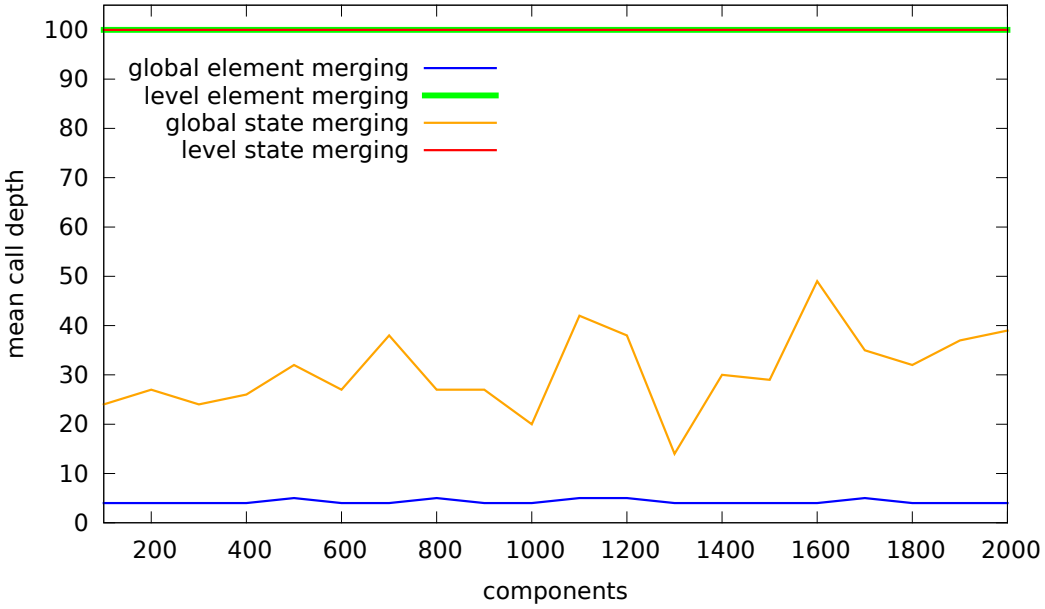


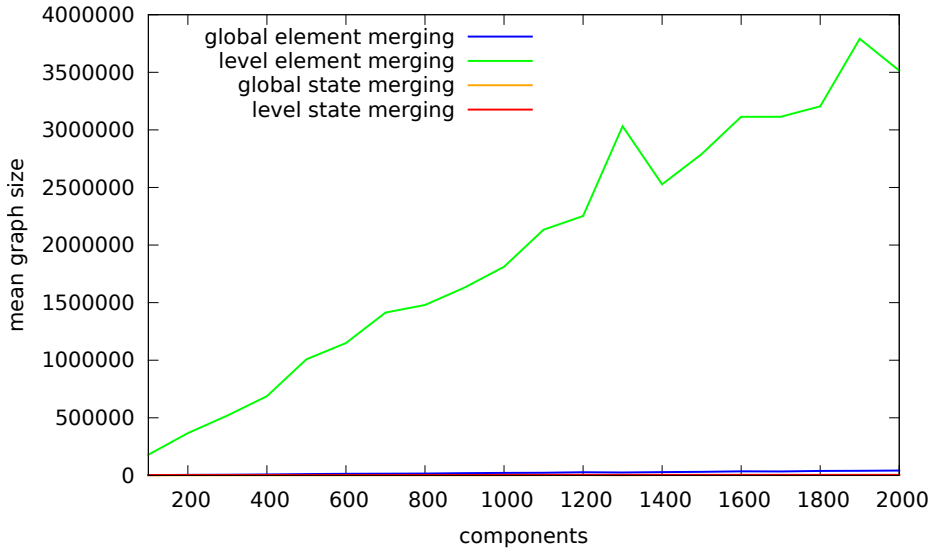
Figure 8.2.: Mean black box call depths using black box set in the size range from 100 to 2000.

for the global state merging. While the global element merging stays almost constant, the call depth of the global state merging deviates much more. Also for the latter, there is a slight increase in the needed maximum call depth from a mean 27.2 in the range of $100 - 1000$ black boxes to 34.8 in the range of $1100 - 2000$. As we will see, this correlates with the total number of states within the resulting graph.

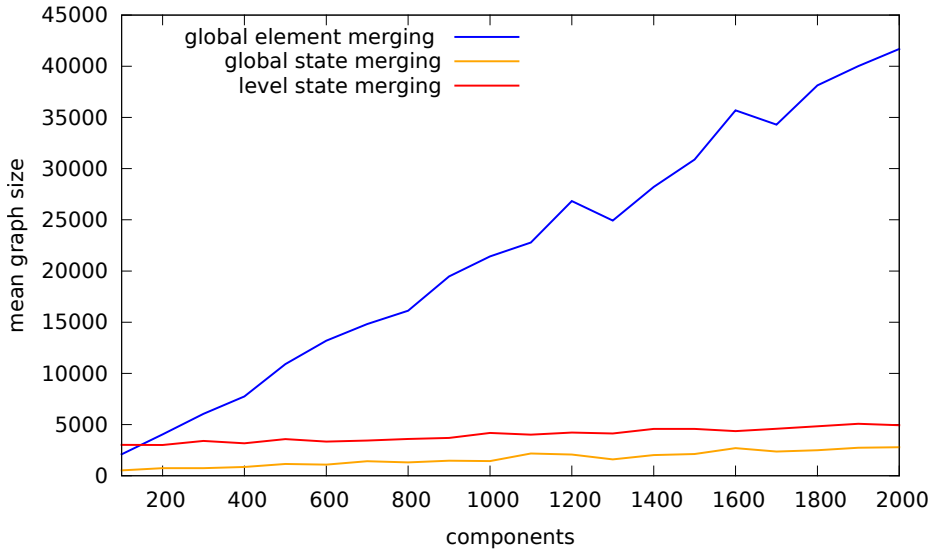
8.2.2. Graph Size

The evaluation of the graph sizes was motivated by the question how big the graphs can grow and how fast their sizes increase. We are also interested whether there is a coherence between the graph size and other behaviour such as the runtimes. Depending on the execution mode of VarDroid, the graph nodes can either be black box elements or states. For a better comparison we therefore created two statistics: one which considers the total numbers of graph nodes, i.e. regardless whether they are black box elements or states, the other shows the total numbers of black box elements in the overall graph. The

results are shown in Figure 8.3 and Figure 8.4.



(a) Total graph size including the level-based black box element merge.



(b) Total graph size without the level-based black box element merge.

Figure 8.3.: Mean graph sizes after applying Phase 3 while using black box set in the size range from 100 to 2000.

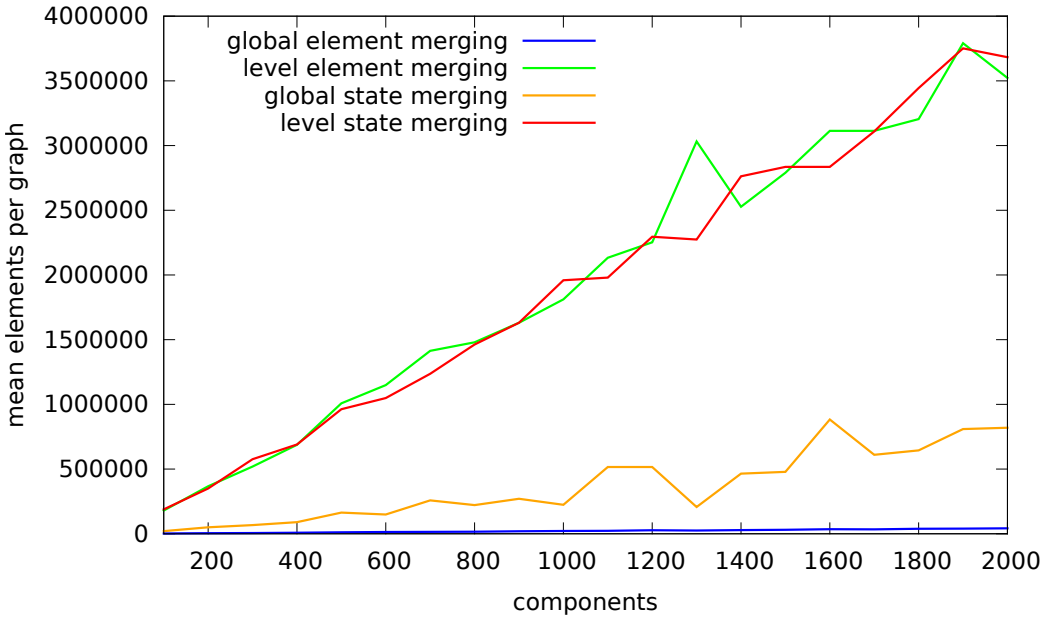


Figure 8.4.: Mean element count in graph after applying Phase 3 while using black box set in the size range from 100 to 2000.

We show the total graph sizes in two graphics 8.3a and 8.3b. While the first one shows all four execution modes, the second contains all but the level-based element merge mode. The reason is that the level-based merge mode contains far the most graph nodes which makes the other execution modes hardly visible in the first graphic. However, the high number of nodes is not surprising: firstly, it only merges black box elements per level and therefore as many redundant elements on different call depths, and secondly it the missing fixed point detection always leads to the full maximum call depth (cf. Section 8.2.1). Thought the number of total graph nodes is much lower for the level-based state merging, the number of total black box elements is almost the same, increasing linearly with the number of black boxes. The explanation of this behaviour is the same as for level-based element merging. Interestingly, the total number of graph nodes, i.e. the total number of states, increases only slightly compared to the more dramatic growth of the black box elements count. So, at this point we can already see the advantage of states to reduce the total graph size.

The global state merging shows the same behaviour for the total number

of graph nodes but is of course much lower for the total number black box elements. Compared to the global element merging, it has much fewer graph nodes. More importantly, the number of nodes growth far slower than for global element merging keeping the graph at a relatively small size. However when looking at the number of black box elements for the two global merging modes, the global element merging has much less elements. The reason is the definition of state in which it is possible that equivalent black box elements coexist in different states. However for the global element merging equivalent elements are always merged and thus can never coexist. Because of the chance of not merging the black box elements their number is higher in the global state merging mode than when merging elements globally.

8.2.3. Execution Time

The motivational questions for the evaluation of the runtime were how long the individual executions take, how the runtimes change with increasing black box sets, and how the different execution modes perform in comparison. We show the results in Figure 8.5.

As expected the runtime of the two level-based execution modes behave similar. This is a direct consequence of the other statistics we already handled, most significantly the total number of black box elements within the respective graphs. Also the reached call depth which is the maximum one for all level-based executions influences the similar behaviour. It is interesting that though the total number of black box elements grows quite linearly the execution times of the level-based modes increases faster and non-linear to size of the black box sets.

The behaviour of the global execution modes is much more outstanding. While the execution times of global element merging took an average of 36 seconds with 2000 black box elements, the times for global state merging exploded to an average of 2 hours and 34 minutes. The good runtimes for the global element merge mode is clearly based on the previously elaborated facts: the relative small call depths paired with a reasonable growth in the number of graph nodes leads to a early termination of our algorithms. In contrast, the massive increase of the execution time of the global state merging algorithms was completely unexpected. Firstly, the mean call depth was indeed higher than the one for global element merging but within reasonable range. Also the only slight increase of the mean required call depth did not hint any dramatic

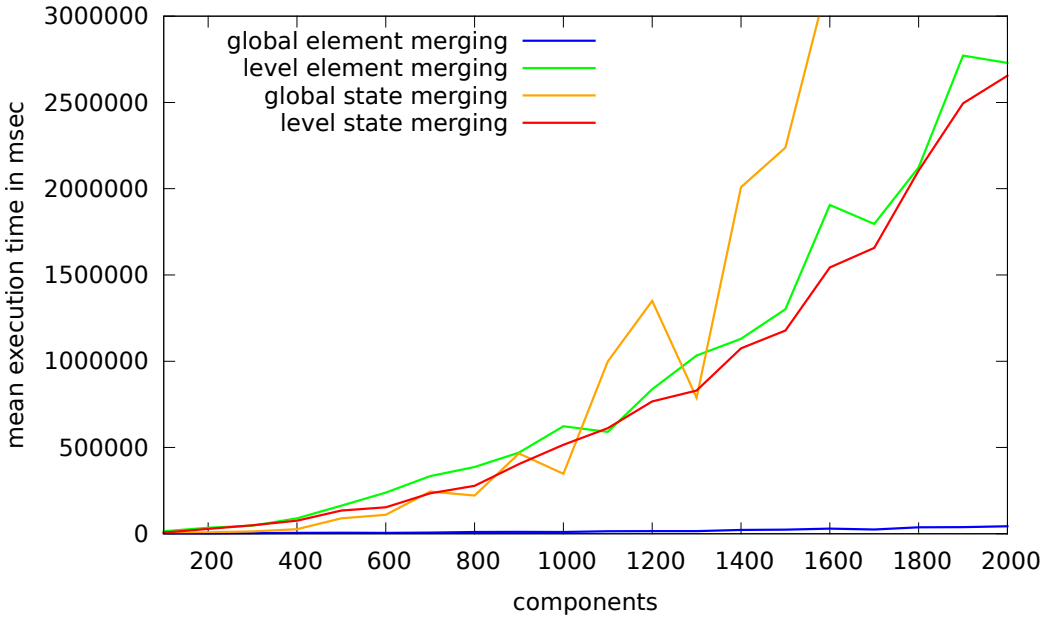


Figure 8.5.: Mean runtime in milliseconds using black box set in the size range from 100 to 2000.

increase of the execution times. Secondly, the resulting state graphs keep to a reasonable size with a mean of 2792 nodes and 819384 black box elements when processing 2000 black boxes (the graph of global element merging has 41689 nodes / black box elements). However, the most unexpected behaviour of global state merging was that it is even much slower than the level-based merging modes with roughly 45 minutes execution time.

8.3. Result Interpretation

There are two possible reasons for the outstandingly long execution times of the global state merging execution mode: either it is a design or an implementation problem. Indeed, when we look at the algorithms in Section 6.3 we can see that we use four nested loops in the *intraComponentAnalysis* function and three nested loops in the *interComponentAnalysis* function. However, every state and every black box element is only iterated over at most once per function call. Besides, the algorithms for the global element merging are exactly the same as

for the global state merging except for the state construction. So, the reason for the huge difference of the execution time is not on the conceptual level of the algorithms. A common reason for unexpected performance behaviour are bugs in the practical implementation. Like it is for the conceptual level, also VarDroid's implementations for the different merging modes share common code. In fact, the code implementing the global element merging is almost completely the same as the one of the global state merging. The difference of the source codes is the construction of states. This of course suggests that the reason for the long execution times must lie in the state construction code. However, we were not able to identify the exact location at the time of writing this master thesis.

9. General Approach Limitations

This master thesis presents the first basic developments for an analysis of a huge set of smartphone applications. The current approach still comes with various assumptions and limitations. In this chapter, we discuss the most significant ones while leaving the improvements to overcome the existing limitations to future work.

Concurrency and Side Effects Up to now, we only considered sequential calls of application components. As an implication, we do not have concurrency in program executions. This limitation affects several system features as, e.g., listeners, threads, but also the component types we can use for first evaluations. Implications on the latter are discussed below. In case there are multiple candidates for the next execution step, we assume an equal chance for all candidates as the next execution step. If for example an activity implements two button on-click event listeners l_1 and l_2, we follow the control flows if either l_1 or l_2 was triggered. Moreover, we assume only a single click, i.e. we do not consider click sequences as, e.g. $l_1 \rightarrow l_2 \rightarrow l_1$.

Component Types Though the in this master thesis provided approach handles components in general, the focus lies on activity components. The reason is closely related to the issue of concurrency. For example, the purpose of services is to provide a mechanism for background tasks, hence automatically comes with concurrency which we cannot handle appropriately. A possible first way to overcome this limitation is to make the assumption that the runtime environment waits for the termination of the, e.g., service component code before continuing with execution of the rest. However, this is not yet implemented.

Evaluation Due to the exclusion of concurrency and the implication on the handling of components, we currently cannot analyse arbitrary real world ap-

plications from, e.g., the Google Play-Store. Therefore, we only used randomly generated component black boxes as examples for our evaluations.

Non-intent Data-flows Besides intents, there are several other ways to transfer data from one component to another which we currently do not cover in our approach. For example, it is not possible to correctly track flows between two components of the same application using public static Java class variables. This is caused by the fact that our approach does not support component state preservation but instead assumes a new component instance on every call in the component call graph.

10. Future Work

There are improvements for our approach and practically for VarDroid we want to undertake. Throughout the master thesis we already stated some of the currently existing assumptions and limitations which we will try to remove in future versions to create a more powerful analysis tool. In this chapter we will address some of the most relevant topics which we will cover in future work.

VarDroid is designed to be fairly independent from the actual component black box generation. For first test runs and performance evaluations it was sufficient to use the randomly generated black boxes created by ComponentGenerator. However, we are of course interested how VarDroid performs with real world applications. Therefore, we need to undertake certain improvements: first, we will replace the ComponentGenerator used in Phase 1 with specialised Android application analysis tools. In fact, we are currently already investigating how to adjust and integrate tools such as FlowDroid [3] to produce the required component black box representations.

Android allows inter-component communication in many different ways. In VarDroid we currently focus on intent-based component calls. However, for example service components offer inter-process method calls via IBinder objects. Consequently, we need to extend the current black box model, in particular the input- and output-interfaces, to also allow other communication channels besides intents.

Application components dynamically created during runtime must not be explicitly handled by VarDroid. We see it as the task of the application analysis tool during Phase 1 to detect their existence and to create a suitable black box representing them.

Another topic we want to examine is the definition of security labels. VarDroid provides a configurable set of security tokens and security conflicts. The disadvantage of sets is, e.g. that it does not respect the order in which security tokens were added to a security label. For example, we get the same security

label when data originating from a private source flows to a public sink as for data from a public source flowing to a private sink. Though the first situation is correctly detected as a security leak, the latter is usually a permitted flow because no private data is leaked. We therefore plan to improve the definition of security labels to more accurately encode certain data-flow aspects.

A. Appendix

```
components = 2000

minInputs = 1
maxInputs = 3
minOutputs = 0
maxOutputs = 3
minSources = 0
maxSources = 3
minSinks = 0
maxSinks = 3

probInputSink = 0.5
probInputOutput = 0.5
probSourceOutput = 0.5

numActions = 4
probExplicitIntent = 0.5

numPermissions = 3
probGrantPermission = 0.3
probRequirePermission = 0.3

numSecurityLabels = 3
probInputSecLabel = 0.0
probSourceSecLabel = 1.0
probSinkSecLabel = 0.5
probOutputSecLabel = 0.25
```

Figure A.1.: Configuration file which was used for the evaluation runs in Chapter 8

Bibliography

[1] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Grösslinger, and Dirk Beyer. Strategies for product-line verification: case studies and experiments. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 482–491, Piscataway, NJ, USA, 2013. IEEE Press.

[2] Charles Arthur. Feature phones dwindle as android powers ahead in third quarter. http://www.guardian.co.uk/technology/2012/nov/15/smartphones-market-android-feature-phones, November 2012.

[3] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel. Highly precise taint analysis for android applications. Technical report, EC SPRIDE, May 2013.

[4] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[5] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22nd USENIX Security Symposium*, August 2013.

[6] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[7] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: an

information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[8] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 235–245, New York, NY, USA, 2009. ACM.

[9] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. Permission re-delegation: attacks and defenses. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association.

[10] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. Scandroid: Automated security certification of android applications, 2009.

[11] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, October 1988.

[12] Apple Inc. App store tops 40 billion downloads with almost half in 2012. http://www.apple.com/pr/library/2013/01/07App-Store-Tops-40-Billion-Downloads-with-Almost-Half-in-2012.html, January 2013.

[13] Google Inc. http://developer.android.com/develop/index.html, July 2013.

[14] Google Inc. Android main page. http://www.android.com/, July 2013.

[15] Julio Sincero, Horst Schirmeier, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Is The Linux Kernel a Software Product Line? In Frank van der Linden and Björn Lundell, editors, *Proceedings of the International Workshop on Open Source Software and Product Lines (SPLC-OSSPL 2007)*, 2007.

[16] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Large-scale variability-aware type checking and dataflow analysis. Technical Report MIP-1212, Department of

Informatics and Mathematics, University of Passau, Passau, Germany, 11 2012.

[17] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 229–240, New York, NY, USA, 2012. ACM.

[18] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: Extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, 2010.

[19] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically rich application-centric security in android. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 340–349, Washington, DC, USA, 2009. IEEE Computer Society.

[20] AT&T Labs Research. The dot language. http://graphviz.org/content/dot-language, Last visited: 16. August 2013.

[21] Jamie Rosenberg. Google play hits 25 billion downloads. http://officialandroid.blogspot.de/2012/09/google-play-hits-25-billion-downloads.html, September 2012.

[22] Stephen Shankland. Is android fragmentation over with jelly bean's rise? nope. http://news.cnet.com/8301-1035_3-57593500-94/is-android-fragmentation-over-with-jelly-beans-rise-nope/, July 2013.

[23] Thomas Thüm, Sven Apel, Christian Kästner, Martin Kuhlemann, Ina Schaefer, and Gunter Saake. Analysis strategies for software product lines. Technical Report FIN-004-2012, School of Computer Science, University of Magdeburg, April 2012.

[24] Iain Thomson. Report: Android malware up 614% as smartphone scams go industrial.

http://www.theregister.co.uk/2013/06/26/android_malware_bloom_security_updates/, June 2014.

[25] Alexander von Rhein, Sven Apel, and Franco Raimondi. Introducing binary decision diagrams in the explicit-state verification of java code, 2011.

[26] William Enck, Machigar Ongtang, and Patrick McDaniel. Understanding android security. *IEEE Security & Privacy*, 7:50 – 57, January/February 2009.