Università
Ca'Foscari
Venezia

Master's Degree programme – Second Cycle
(*D.M. 270/2004*)
in INFORMATICA – COMPUTER SCIENCE

–

Final Thesis

Ca' Foscari
Dorsoduro 3246
30123 Venezia

Intent Flow Analysis of Android Applications

Supervisor
Prof. Agostino Cortesi

Assistant Supervisors
Prof. Fausto Spoto
Ph.D. Pietro Ferrara

Graduand
Rocco Salvia
Matriculation Number 838737

Academic Year
2015 / 2016

Abstract

This work is focused on the implementation of an intent analysis, integrated into the Julia static analyzer, to discover intent flows inside Android applications. An intent is a message that can be sent inside or outside the application, passing through various components, and finally reach a destination. Our analysis warns when an intent carries sensitive information, or even worse, when tainted information are retrieved from it. It tracks intent flows, from the starting point in any component of the application, and exploit Julia's flow analysis in order to know about the taintedness of the information carried around. This tool is first tested on a well-known open tests suite (DroidBench), widely used to evaluate the effectiveness of taint-analysis tools for any Android application, and it is finally tested over a real world open source application (Telegram), where two dangerous flows of sensitive data towards an email service application are discovered.

Abstract

L'analisi implementata è un'integrazione dell'analizzatore statico Julia, con lo scopo di individuare flussi di intent in applicazioni Android. Un' intent è un messaggio al quale possono essere allegati dei dati e può viaggiare all'interno del dispositivo mobile, attraversando diverse componenti, raggiungendo infine una destinazione. L'analisi genera degli allarmi quando le intent trasportano informazioni riservate, o nel caso peggiore quando queste vengono estratte nel corso del flusso. Sono state prima analizzate delle applicazioni Android di una libreria di test dedicata agli analizzatori statici (DroidBench) ed infine una applicazione open source individuata nel marketplace (Telegram), nella quale sono stati identificati due flussi di informazioni sensibili verso una nota applicazione di posta elettronica.

Contents

Chapter 1

Introduction

In this thesis we developed an intent analysis for Android applications. The analysis is integrated into the Julia[39] static analyzer that applies abstract interpretation[30][36] to the analysis and verification of Java bytecode. The static analysis offers compile time techniques to infer an overapproximation of the program execution. In our case the static analysis is adopted to get all possible trajectories that any intent could traverse during the lifecycle of the application under analysis. Before the presentation of the tool we get the basis of Android OS and intent communication mechanism.

1.1 Context

Nowadays Julia[39] static analyzer is able to detect only a part of the intent's flow. It correctly detects the initialization point of an intent, all the filling operations (in particular those involving taintedness), and finally all the sink methods allowing the intent to flow outside the detected component. The missing part of the analysis includes the receiving phase of the intent, as a matter of fact nowadays there are only two available possibilities regarding flow analysis during the receiving procedure of an intent: to consider all incoming intent as tainted, or to consider all intent as clean. This thesis is focused onto improving this part of the analysis in order to be precise when an intent is received. The goal is to detect only those intents that were filled with tainted data by the sender, in order to propagate the flow when they are received. There are a lot of information inside a device that are considered sensible. They are divided in: information related to the device (name, manufacturing company, operating system) or related to some action performed by the user (GPS position, telephone numbers, browser history). The operations allowing the developer to retrieve those informations must be marked as sources, in order to track how the application under analysis managed them, and if it leaks those information outside.

1.2 Goals

The goal of this thesis is to study the problem of intent flows inside an Android application, starting from general knowledge about Android applications, in-

tents communication mechanisms, and finally the integration inside Julia static analyser of a tracking analysis for the suspicious parts involved in a communication performed through intents, in order to mark potential tainted data flow. This is the building-block of a wider work that expands the analysis to an inter-application intent flow. The final aim and main future work, is to give in input to the analysis a bunch of applications and get back results in order to discover how and if those applications will move sensible data among them, and between the application and the operating system.

1.3 Work Strategy

The development of the analysis was done through the completion of the following steps:

- Get the basics concepts of Android operating system working mechanism. The development of the analysis starts in the 2016 summer during the release of Android 7.0 Nougat. Intent mechanism comprehends: creation, management and sending processes, they were an Android's basis since its origin. Infact most of methods and classes handled belongs to API1.
- Get familiarity with AndroidStudio[5], the official IDE for developing Android applications. It involves Gradle[20], Android virtual device Manager and SDK Manager.
- Get the basics concepts of JuliaDexConverter[25] which is composed by two tools:
  - Dex2Jar[13] allows the reverse of the APK into JAR in order to perform the analysis. The adopted version is modified in order to preserve debug information.
  - APKTool[6] performs the XML transformation: from binary format to a readable representation of the Android Manifest.
- Get Julia[39][24] basics concepts, available checkers, annotations, run configuration and parameters.
- Design and development of the Android flow oracle, inside the framework section of the Julia analyzer. The analysis is based on sources and sinks annotation mechanism.
- Test of the analyzer on DroidBench[15][14].
- Empirical evaluation of the analysis on the messaging Android application Telegram[27][18].

1.4 Results

The developed analysis has been tested over the known suite test cases DroidBench[15][14], and then over Telegram[27][18] messaging application. In particular in Telegram are found two leakages of sensitive device information toward an application sender. In our analysis the involved application is Gmail,

but could be any application satisfying the action `android.intent.action.SEND`. All results are described, also by a graphical representation using Graphviz Dot[21]. Most important leakages are showed in source code and triggered during the execution of the application on a mobile device.

1.5 Overview of the Thesis

The following chapters are organized as follow:

- Chapter II describes the background of intents mechanism in Android operating system, Android manifest structure, and annotations system based on sources and sinks definitions. An overview about Java bytecode with examples is given, then the basic blocks definition used into the Julia analyzer and the Injection checker used for the analysis are described.
- Chapter III contains the definition of the intent model built and adopted in the analysis.
- Chapter IV explains how to correctly configure and execute the tools needed by the analysis, it means how to get the JAR archive from the source code in AndroidStudio, how to run Julia analyzer, how to run JuliaDexConverter to reverse the APK.
- Chapter V is about the results section, mainly focused on DroidBench suite test cases and Telegram analysis results.
- Chapter VI contains related works.
- Chapter VII is about limitation, future works and considerations about the implemented analysis.

Chapter 2

Background

This section describes the Android communication mechanism based on intents and the set of components that form the bases of the communication on intents. Then it is described the purpuse of Android manifest inside an application, and how to get possible communication flows from it. All the constructors of an intent are described, togheter with modifiers that alter the destination address and the contents of the intent. It is explained what kind of methods are considered as sources, how to control them, and warning when sensitive information sinks out. Then are discussed main features of Java bytecode, and how Julia translates it into basic blocks. Finally we discuss about Injection Checker and the taintedness analysis computed through Boolean Formulas.

2.1 Android & Intent

An Android application is formed by components. Those are the building blocks of the application, and each of them exists on its own and could be an entry point of the application. Entry point means that the application is startable from it. An intent is a bearer of information that connect two entities, called components, involved in the comunication.
Components are divided in:

- Activities: an `Activity`[7] is a single screen of the application. A parallel can be done with the MVC paradigm, when the View is tuned by the Control. An application can be composed by many activities each of them with its scopes, and with the ability to communicate with other components inside or outside the application.

- Services[4]: a `Service` is a set of batch processes with no interaction with the user. They are usually labeled as background processes since they can work without need of users interaction. For instance a service component can be adopted in order to perform a downloading operation.

- Broadcast receivers: a `BroadcastReceiver`[9] is a listener of events, triggered by the operating system or by other components. The BroadcastReceiver is registered in order to intercept specifics events in order to perform the associate action.

- Content providers: a `ContentProvider`[3] is an interface between the application and the memorization layer, where information can be stored into databases, files or over the network. It can perform inter-operability between applications and memorization layer (consider all the applications that access to the same address book for managing it).

The communication involving the components is based on messages called intents.
An intent can be implicit or explicit:

- Implicit: the receiver is not defined when the intent is created, but it is chosen by the operating system together with the user. First the operating system is consulted about who can holds the operations requested by the intent. If there are more than one match, the user will be asked in order to takes the decision. On the other hand if only one match occurs, the operating system will automatically takes the decision (extremely dangerous).

- Explicit: the receiver of the message is uniquely determined. The intent contains the fully class name of the receiver. This solution is usually adopted for intra-communication application, since is well known which operations the application can performs.

2.1.1 Application Manifest

When an implicit intent is sent to the operating system in order to find all matching receivers, the dispatcher researches all the applications that can handle the requested operation. This is done through comparison between, the request carried by the intent, called action, and what is offered by the application's components. Any component that satisfies the intent is collected by the operating system. In case of several satisfactory components, the OS asks to the user which one better suits the request, and on the other hand the operating system selects as receiver the only one avaiable. The application's offer is described by the `AndroidManifest.xml` and in particular in the intent filter tag. Each component can define its intent filter, where all the actions that it can performs are described. The manifest informs the operating system about the application capabilities. An intent filter allows the application to be selected to perform the operation described by the action tag, that must be contained in every intent filter. If an application does not specify any intent filter than it can be reached only by explicit intents, since an explicit intent is always delivered to the component's destination regardless if the intent filter is defined. An implicit intent sent to the dispatcher of the operating system can't reach a component that did not declare exactly that precise action in its intent filter. In case of services it is considered dangerous to attach an intent filter to a service, since unknown external entity could start it. (This is why usually the service is not exported, `exported=false` in the manifest). If a component is not exported it means that it can be started only by components of the same application.

```java
public class OutFlowActivity extends Activity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      TelephonyManager telephonyManager = (TelephonyManager)
          getSystemService(Context.TELEPHONY_SERVICE);
      String imei = telephonyManager.getDeviceId(); //source
      Intent i = new Intent();
      i.setAction("lu.uni.serval.iac_sendbroadcast1.ACTION");
      i.putExtra("Secret-IMEI", imei);
      sendBroadcast(i);
   }
}
```

Listing 2.1: The activity creates a new implicit intent and sets the action through the method `setAction`. It appends an extra field mapped to the key `Secret-IMEI`. The IMEI is the identification number of the device, and is considered a sensitive information.

```java
public class InFlowReceiver extends BroadcastReceiver {
   @Override
   public void onReceive(Context ctx, Intent intent) {
      String imei = intent.getStringExtra("Secret-IMEI");
      SmsManager smsManager = SmsManager.getDefault();
      smsManager.sendTextMessage("+39347...", null, imei, null, null);
   }
}
```

Listing 2.2: The broadcast receiver is an intents listener. It receives the intent sent by `OutFlowActivity` and it leaks out the IMEI code through an sms message.

```xml
<receiver
    android:name="lu.uni.serval.iac_sendbroadcast1_sink.InFlowReceiver"
    android:label="@string/app_name" >
  <intent-filter >
    <action
        android:name="lu.uni.serval.iac_sendbroadcast1.ACTION"/>
    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
</receiver>
```

Listing 2.3: The receiver can listen for the intent since it defines an intent filter in the manifest. It is easy to see that the action is the same sent by the activity.

2.1.2 Intent Constructors

The purpose of the intent is to activate the receiver and if needed to pass additional information. Since the intent mechanism is managed by the operating system, the path traced by the intent is hidden to the application since the beginning, when the intent left the sender, until when it reaches the destination. The intent creator is the set of instructions that produces a new intent and allocates it to the heap. The creation can be performed by six constructors, which are divided in two set: five are simple, and one is a copy constructor.

Simple:

- `new Intent()` creates an empty intent.
- `new Intent(String action)` creates an intent with a given action.
- `new Intent(String action, Uri uri)` creates an intent with a given action and for a given data url.
- `new Intent(Context packageContext, Class cls)` creates an intent for a specific context, and require a particular destination class.
- `new Intent(String action, Uri uri, Context packageContext, Class cls)` creates an intent as explained it the previous bullet point but defines also a specified action and data parameter action.

Copy:

- `new Intent(Intent o)` clone the attributes of the intent given in input. The main difference between simple and copy constructors is that the first set contains only starting points of an intent. Any intent defined using simple constructors exists from the moment in which the new bytecode instruction is founded. For copy constructor what is contained into the new intent could be copied from an intent coming from the rest of the world. Copy constructor cannot be considered as starting point of an intent since it is not clear from where the intent attributes coming from.

2.1.3 Intent Modifiers

Given an intent, the goal of the analysis is focused on two topics: the information attached to the intent and its destinations. The first ones are called extra, the second ones can be tuned thanks to intent modifiers. As explained before, intents contains both the destination address and the action that the sender desires to perform. The destination can be given in input to the constructor, but can also be added to the intent after its creation. After the creation the intent tuning is done through a set of methods:

- `Intent setAction(String action)` defines the action to be performed.
- `Intent setClass(Context packageContext, Class<?> cls)` defines the destination component's name and the class receiving the intent.

- `Intent setClassName(String packageName, String className)` defines the name of the receiver class, with the explicit declaration of application's package name and the class name.
- `Intent setClassName(Context packageContext, String className)` defines the same as before but in this case the first parameter is the instance of the context.

2.1.4 Intent Extra

Any intent can contain additional information beyond actions and addresses. These informations could be additional parameters to the request that the receiver should perform, and are called extras. Any extra is contained in a map that associate the information to a string called key. This information can be both a primary type or an object, and it is inserted in the map before the intent is sent. It could be retrieved by the receiver when the intent is get. Once the intent's flow in the application under analysis is tracked, it is not hard to propagate extra values from the source to all possible destinations, since any time a component extracts an information from the received intent, it means that it is one of intent's consumer, and not only a communication channel between a source and a future receiver. The core feature of the tool is to find if the information extracted from an intent are tainted. In order to answar to this question it is necessary to mark taints or secret values when they are inserted into the intents in order to scan the application and look for any methods that fills the intent, finally checking if the value associated to the key is taint.

2.1.5 Source, Flow and Sink

The analysis reports undesired sinks, starting from a dataset sources containing those methods marked as dangerous. In any device there are a lot of private information, like the smartphone's IMEI number [12]. Those data are retrived by the developer from the OS through methods call, which returns the desired information. In order to tail them in the flow, that means to control how the application under analysis managed them, the methods returned value are marked as dangerous, untrusted, in order to recognize them any time. Taint analysis requires a preliminary procedure before starting the analysis. This operation is manually performed, and consists into signing the operations considered as sources of tainted data 2.5. Moreover a set of methods that allow data to sink out exists, as example the `sendTextMessage(...)`[11](Listing 2.5) method of the class `SmsManager`[10], that allows to send an SMS text message. In a similar way the methods where the taintedness has not to flow in must be marked (Listing 2.5). Those methods should receives only secure information. When the IMEI of the telephone flows into the text of an SMS (Listing 2.2) an Injection is matched, and Julia warns those (dangerous) operations if requested, and if the procedure or marking sources and sinks is done. The same is true for the method `sendBroadcast(Intent i)` reported in Listing 2.1.
Julia performs the flow analysis over the application in order to get the tainted data model of the application. As later described, the flow analysis is performed through boolean formulas abstraction, in order to get a binary answer about the objects. The resulting analysis is consulted like an oracle when a question

about the taintedness of an entity is raised. The tool described in this work produces an Android intent flow analysis, with the main goal of depicting intent flow taintedness. The idea is that intent extra keys are based on string, which are usually constant in Java. Most of the times intents are filled with values associated to a key of type string. This is one of the limit of our analysis: it recognized constant strings used as destination addresses of an intent, both for constructors and modifiers, or keys used during filling operations. The implemented analysis finds constant strings and collect them, but when a non constant value is found it is approximated with a top value("***"). Now suppose that a getter operation extracts a key from an intent, but this key is not a constant: the approximation done by the analysis is to collect all tainted key contained into the intent, as potentially requested by the getter.
When an intent filler instruction is founded the analysis asks to the oracle if the value inserted is trustable. If not, the key is a label of taintedness. When during the intent flow an entity performs a getting operations, like for the filler, it extracts information associated to a determined key. Thanks to the model build during the filling phase, it is asked to the Android oracle about the key, and at the end the edge between the sender and receiver is built.

```
void sendTextMessage(String destinationAddress, String scAddress,
    @Trusted String text, PendingIntent sentIntent, PendingIntent
    deliveryIntent)
```

Listing 2.4: sink method, the string given in input as the message text must be trusted.

```
@UntrustedEnvironment String getDeviceId();
```

Listing 2.5: source method, the value returned by the method is sensitive information. Is marked as untrusted since it is dangerous, moreover it is considered as flow source.

In order to performs intent analysis must be built the same model. Any methods returning sensible information like for instance IMEI code, or the GPS position, have to be tainted. The idea is that when one taint information flows inside a `putExtra(key,value)`, the analysis produces a warning. The flow analysis is already performed in Julia, since the leak consists in an Injection sink.
The problem come out when the intent is received, since it is impossible to go back from the receiver to the sender. The intent is serialized after it is sent and deserialized before it is retrieved. Moreover the reception of the intent is done in Android's core. The only available information are the set of keys used to retrieve information. The adopted solution consists into tracks intents before they are sent and connects every matching components through intent, in order to have a graph of intents to consult when a getting operation is founded.

2.2 Bytecode Java

Since Julia is able to analyse Java bytecode, it is necessary to revert the APK of the Android application into JAR archive. Existing tools to do that, in particular Dex2Jar[13] are described in the Configuration Settings chapter. The Java Virtual Machine[35] is divided into four main components:

- Class: where there are resident classes code and constants. Methods implementation are stored in a space called methods area, instead constants are kept in a constants pool. The class definition, that is the code building the class, is stored as template for any instances. Class properties (superclass, interfaces, fields, methods, constants) are immutable, meaning that after they are defined there is no way to modify them. In case of fields, if static field exists then a single copy of the field is considered for all classes of that type, if not static, there is a reserved copy for each object.

- Stack: where it is kept track of any calls to method and data associated to it. Every time a method is called a new data space is reserved for the method, it can be called stack frame. Collectively all stack frames are called Java frames. Each stack frame has an operand stack, an array of local variables, and the program counter (PC). When a method is invoked, a new stack frame is created and then pushed on the top of the stack. The new stack frame initializes the program counter to the first instruction of the method, and then performs its body. When the method returns, the active frame disappear and the following stack frame becomes the active one. The program counter is setted to the instruction after the call to the method.

- Heap: where objects are kept. Each object is associated with a class in the class area. Each object has a number of slots for storing fields, one for each field.

- Native method stacks: this tracks methods that are implemented in another language.

2.2.1 Methods Call

We are particularly focused on how Java operates with methods[33]. Since we are going to inspect any calls to intent's methods, it is essential to know how methods are invoked, the parameters pushed into the stack and the returned value. When some particular methods are called, our goal is to recognize the parameters given in input to them, save them in an apposite structure, in order to match them in the future and tracks a possible flow of information. Constants are the only set of information that the analysis can recognize with full precision, since they are known at compiling time. It is useful to consider constants in general with any referenced string constant being particularly relevant. The idea is that since they are immutable, when one of them is linked to an interesting information is possible to recognized it during all application's life cycle.

```
aload_2
ldc "DroidBench"
invokevirtual <android/content/Intent.getStringExtra>
astore_3
```

Listing 2.6: Bytecode instructions corresponding to the Java code reported in Listing 2.7

```
String imei = intent.getStringExtra("DroidBench");
```

Listing 2.7: Java call method `getStringExtra(String s)`

In Listing 2.6 is reported an example of bytecode program, results of the java code in Listing 2.7
Brief explanation of the bytecode instructions:

- `ldc "MyString"`: is the instruction used to push a constant value on the top of the stack.
- `aload_2`: load a reference to the stack from local variable 2.
- `astore_3`: store a reference to local variable 3.

When a method `m` of the class `x` is invoked through a $x.m(p_1, p_2, ..., p_n)$ the following steps are performed:

1. The value of the variable x is pushed to the stack.
2. Then all input parameters are pushed as well.
3. The method to be called is determined basing on the dynamic type of x.
4. A new stack and a new local variables array is created.
5. The top k+1 elements of the stack are copied onto l_0^*, l_1,..., l_k local variables, where l_0^* contain the referenced object, and from l_1 to l_k the parameters given in input to the function.
6. Control is transferred to the called method.

When return instruction is reached the return value is pushed onto the stack of the caller. It resumes the execution from the instruction after the method call.

2.2.2 Local Variable

Every method has a pool of locals variables. Local variables in Java bytecode are similar to Java variables. Variables are referenced by number starting from 0. As for stack entries, long and double takes two consecutive local variables. The pair of instructions that manage local variables are `load` and `store`. In case of `load` the value contained in the variable is loaded into the stack, instead when `store` pushes the value of a precise local variable to the operand stack. Local variables in our analysis are particular interesting since they are used to manage methods arguments, with the goal of finding those methods with intent parameter. Each method input parameter is stored into a local variable. In case of return value no local variable is used.

2.2.3 Objects Creation

Objects are created by `NEW`. When the instruction terminates, there is a new object on the top of the stack. Then usually constructor is called. The bytecode sequence instructions that creates an empty intent is the following ones:

```
new android/content/Intent
dup
invokespecial android/content/Intent.<Init>
```

Listing 2.8: Bytecode instructions creating a new intent using empty constructor.

`dup` is used in order to duplicate the reference to the earlier created object onto the stack. This is done since the constructor `<Init>` takes the reference to the object and does not return it once modified. This is why `dup` is performed in order to get the object after the `INVOKESPECIAL`.

2.2.4 Fields

To get the value of a field, `getfield` is used, that place the reference on the top of the stack (in case of reference type) of the value contained into the field. The instruction for changing the value in a field is `putfield` that expects both the object containing the field to modify and the new field values.

2.3 Elements of Julia

The Julia static analyser applies the theory of abstract interpretation to the Java Bytecode's analysis[39]. Julia is a library for static analysis, composed mainly by checkers, each of which verify the absence of a family of errors in the software under analysis. The analysis performed by Julia returns a list of warnings, that is a reasonable list of problems founded in the analysed code.

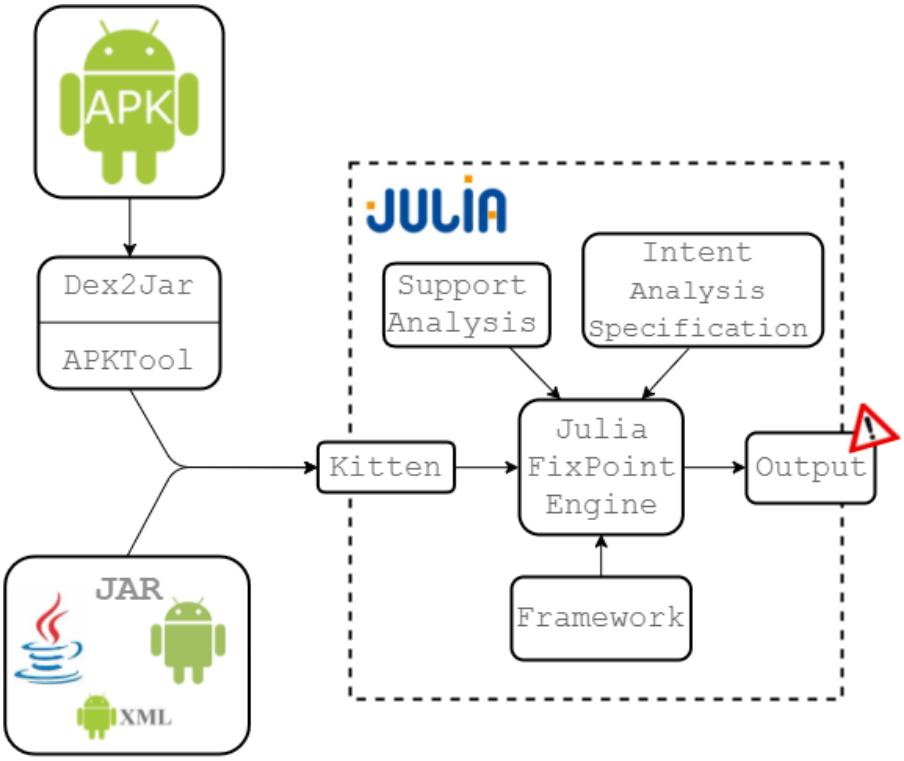


Figure 2.3.0.1: Main entities involved in the project: Julia analyzer and reverting tools. It is showed a basic representation of the Julia internal architecture. Note that Kitten is a parser that reads Java bytecode and optimize it, grouping togheter the instructions producing the same effects in terms of analysis purposes. Support analysis are used and consulted while performing our intent analysis.

2.3.1 Transformation of Java Bytecode into Basic Blocks

A Java bytecode program is composed by a set of classes, each defining methods and fields. A method contains a sequence of Java bytecode statements. The problem of Java bytecode is that it lacks an explicit scope structure and uses an operand stack to hold intermediate computational results. The solution consists in splitting the code into chunks of contiguous bytecodes called basic blocks[33]. Jumps can occur only at the end of a chunk and the target of the jump can only be the first bytecode of another chunk. A control-flow graph is build by connecting blocks with directed edges which reflect the transfers of control into the program. Basic blocks are linked through edges representing transfer of control. The Java bytecode is grouped into blocks and a graph of that blocks is

build. In the graph there are reported also exceptional path, that could occur due to a runtime exception, when an instruction call is performed. Resolution of methods is dynamic in Java Bytecode and Julia applies an algorithm called class analysis in order to approximate it. The analysis of Java bytecode is done also thanks to the bytecode extractor that tracks the boundary of the code that Julia analyses. In order to correctly start the analysis is necessary to define the entry points of the application. Those can be all public methods founded in the analysed application, or they could be manually annotated by the user only over those method that the user desire as entry points of the analysis. The bytecode instruction `if_cmple` e `if_cmpgt` are used in order to select the correct execution path, then is not needed another kind of node for conditions. Our representation of Java bytecode through basic blocks is such that each bytecode which might raise an exception, as for instance `call` instruction, is at the end of a basic block with at least two successors, one for normal flow and the other for exceptional continuation. The latter might further route the computation to the appropriate exception handler, if any. Consider for instance the bytecode in Listing 2.9. It is transformed into the graph of basic blocks shown in the same figure.

```
public void intentReceive(){
  Intent t=getIntent();
  t.setAction("NewAction");
  startActivity(t);
}
```

Listing 2.9: Intent reception through a call to `getIntent()` method.

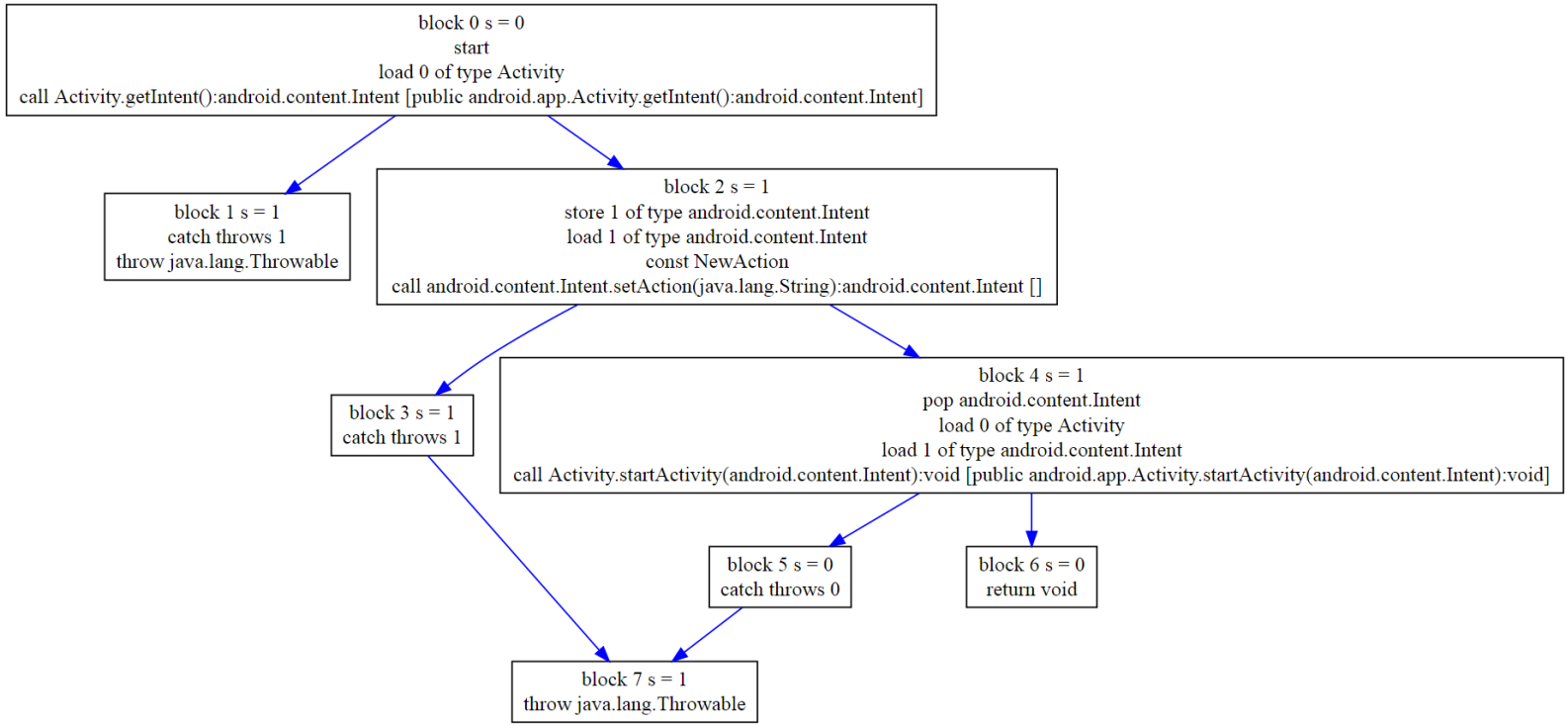


Figure 2.3.1.1: Basic blocks rapresentation of the Java code in Listing 2.9. The instruction `CALL` *is introduced by Kitten parser to group toghether the four Java bytecode instructions:* `INVOKESPECIAL`, `INVOKEVIRTUAL`, `INVOKESTATIC`, `INVOKEINTERFACE` *[35]*

```java
public String randGreat5(){
  double i=Math.random();
  int test=(int) i*10;
  String s="";
  if (test>=5)
    s="Greater than 5";
  else
    s="Smaller than 5";
  return s;
}
```

Listing 2.10: the source code of Java method `randGreat5()`

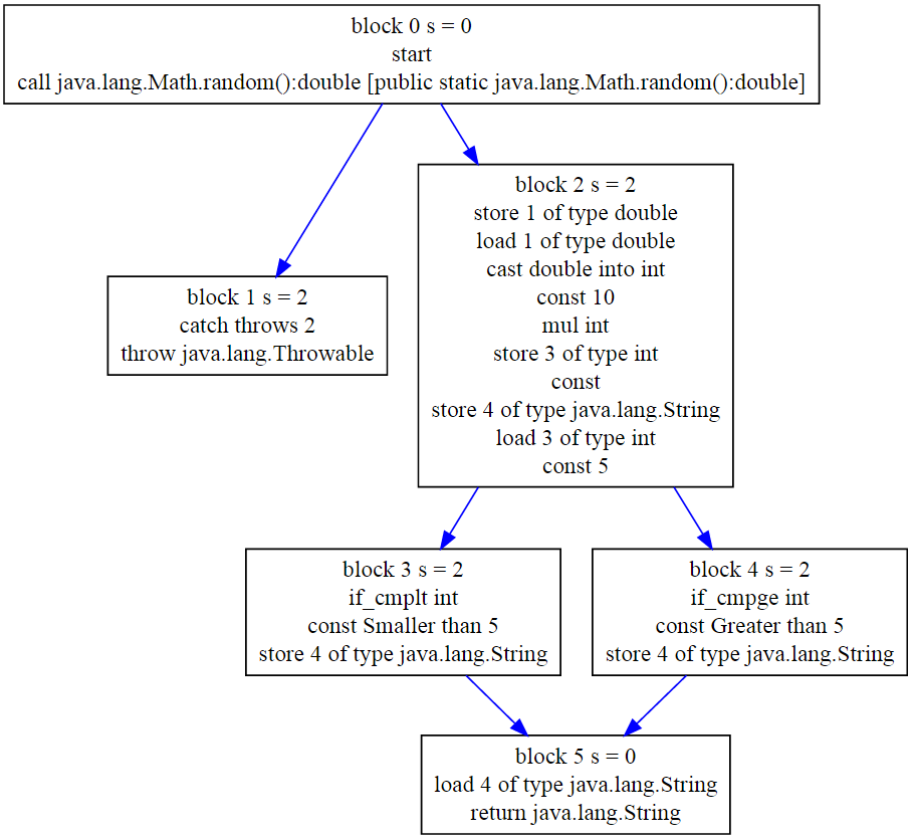


Figure 2.3.1.2: Basic blocks rapresentation of the Java code in Listing 2.10. At the beginning of the block is reported the block number and the stack height.

2.4 Injection Checker and Taintedness Analysis

Julia's taintedness analysis is performed through the interpretation of Java bytecode to Boolean Formula, that express all possible explicit tainted data flows. Julia, starting from a set of tainted input of the user, establishes if those tainted values flows into sensible sinks. In this section we will give an overview of taintedness analysis in Julia[39][32].

2.4.1 Background

Partial Order

A relation R on a set S is called a partial order if it is:

1. Reflexive
$$\forall s \in S,\ sRs$$
2. Antisymmetric
$$\forall a, b \in S,\ (aRb \wedge bRa)\ \Rightarrow\ a = b$$
3. Transitive
$$\forall a, b, c \in S,\ (aRb \wedge bRc)\ \Rightarrow aRc$$

A set S together with a partial ordering R is called a partially ordered set (poset, for short) and is denote (S, R) [29]. Partial orderings are used to give an order to sets that may not have a natural one.

Denotational Semantics of Java Bytecode

We assume a Java bytecode program P as a collection of basic blocks of code, one for each method. In the following definitions we consider only integer primary type and classes as reference types.

Classes

The set of classes $\mathbb{K}$ is partially ordered w.r.t. the subclass relation $\leq$. A type is an element of $\mathbb{K} \cup \{int\}$. Each class define instance fields $k.f : t$ where t is the type of the field and instance methods $k.m(t_1, .., t_n) : t$ where $t_1, .., t_n$ are the parameters of the method and t is the return type.

2.4.2 Abstract State

In our model a value can be an integer $\mathbb{Z}$, a tainted integer $\boxed{\mathbb{Z}}$, or a location $l \in \mathbb{L}$. A state is composed by values, and it is defined by $\langle l||s||\mu\rangle$ where l contains the values of local variables, s the values of the stack elements and μ a set of memory locations bounded to objects. An empty stack is denoted by ϵ, and the concatenation operator is :: . An object o is an instance of the class $o.k \in \mathbb{K}$ and maps identifiers (field f of $o.k$) into $o.f$. The set of states is denoted by Ξ, and $\Xi_{i,j}$ means that the number of stack elements and local variables is fixed to i and j respectively.
A value v has type t in a state $\langle l||s||\mu\rangle$ if:

- t is *int* and $v \in \mathbb{Z} \cup \boxed{\mathbb{Z}}$.

- $v = null$ and $t \in K$, meaning that if v is $null$, t can have any reference type.
- v $\in L$ and $\mu(v).k \leq t$, meaning that if v is a location and $t \in \mathbb{K}$, then instance $\mu(v).k$ must be of type t or a subclass of t.

Example. Let state σ be $\langle[4, \boxed{3}, l]||\boxed{5} :: l' :: l''||\mu\rangle \in \Xi_{3,3}$ with a set of locations $\mu = [l \to o, l' \to o', l'' \to o'']$. Local variables are: variable number zero contains value 4 clean, variable one contains value 3 tainted, variable two contains a location type. The stack contains number 5 taint, a location l' and a location l''. The memory maps location l to object o, location l' to object o', and l'' to object o''. The stack is composed by number 5 coming from an untrustable source, and therefore is tainted, and a reference to location l and another to location l''. The memory is formed by tree locations l, l', l'' that are related respectively to o, o', o''. Where $o.f = l'$ and $o'.g = \boxed{15}$ and $o''.g = \boxed{4}$. The field f of object o refer to l' and the field g of object o' contains tainted number $\boxed{15}$. Field g of object o'' containt tainted value 4.

Exceptions are coded, in particular they are separated from normal flow and are denoted by: $\underline{\sigma} \in \underline{\Xi}$ that appears exactly after a bytecode throwing an exception. After the exception the stack is composed by only one element, that is the location where the exception happened. A bytecode instruction can be represented as a flow from an initial to a final state.
Given the previous definitions, the set of JVM states from now is denoted by $\Sigma_{i,j} = \Xi_{i,j} \cup \underline{\Xi_{i,1}}$.
A denotation is a transaction between an initial to a final state, in notation: $\Delta_{i_1,j_1 \to i_2,j_2}$.
The sequential composition of denotations is $\delta_1, \delta_2 \in \Delta$ is $\delta_1; \delta_2 = \lambda\sigma.\delta_2(\delta_1(\sigma))$: the meaning is that $\delta_1; \delta_2$ represents the transaction from the starting point of $instruction_1$ represented by δ_1 until the end of $instruction_2$ represented by δ_2.
Abstract interpretation defines only four operators:

1. `;` concatenation between states, as demonstrated before.
2. $\cup$ union between the set obtained by two denotations.
3. `extend` operator, reserved for call instruction, where the called method is inserted into the caller context.
4. $\zeta(\Delta)$ denotation of each single bytecode instruction except `call`.

2.4.3 Taintedness Analysis

In this section we defined the abstraction of the concrete semantics. The abstract domain is made of Boolean Formulas that describe all possibles way to propagate taintedness in the program. The concrete semantics works over $\zeta(\Delta)$ and is built from one singleton (sets made of a single $\delta \in \Delta$ where δ describes the behaviour of a single instruction) for each bytecode, with the operators defined before.

Taintedness

Let $v \in \mathbb{Z} \cup \boxed{\mathbb{Z}} \cup \mathbb{L} \cup null$ be a value and μ a memory. The property of being tainted for v in μ is defined recursively as:

- $v \in \boxed{\mathbb{Z}}$
- v is a memory location in L and $\mu(v) = o$ where o is an object with a field f such that $o.f$ is taint in μ.

Tainted Variables

Let $\sigma \in \Sigma_{i,j}$. Its tainted variables are:

- $\{l_k | l[k]$ is tainted in $\mu, 0 \leq k \leq i$ where i is the number of locals$\} \cup \{ s_k | v_k$ is tainted in $\mu, 0 \leq k \leq j\}$ if $\sigma = \langle l || v_{j-1} :: ... :: v_0 || \mu \rangle$
- $\{l_k | l[k]$ is tainted in $\mu, 0 \leq k \leq i$ where i is the number of locals$\} \cup \{e\}$ with an exception thrown on the unique element on the stack v_0 if $\sigma = \underline{\langle l || v_0 || \mu \rangle}$ and v_0 is tainted in μ
- $\{l_k | l[k]$ is tainted in $\mu, 0 \leq k \leq i$ where i is the number of locals$\}$ if $\sigma = \underline{\langle l || v_0 || \mu \rangle}$ and v_0 is not tainted in μ.

The meaning of this latter part is that if a field in an object is tainted, the container object is tainted, and every location that refer to this object is considered tainted. Moreover to make the analysis flow sensitive, distinct variables abstract the input (marked with $\check{}$) and the output of a denotation (marked with $\hat{}$). For instance $\check{l_2} \rightarrow \hat{s_2}$ means that the taintedness in output of $\hat{s_2}$ depend of the origin taintedness in input to local variable $\check{l_2}$.

Taintedness Abstract Domain $\mathbb{T}$

Let $i_1, j_1, i_2, j_2 \in \mathbb{N}$. The taintedness abstract domain $\mathbb{T}_{i_1,j_1 \rightarrow i_2,j_2}$ is the set of Boolean Formulas over $\{\check{e}, \hat{e}\} \cup \{\check{l_k} | 0 \leq k \leq i_1\} \cup \{\check{s_k} | 0 \leq k \leq j_1\} \cup \{\hat{l_k}, 0 \leq k \leq i_2\} \cup \{\hat{s_k} | 0 \leq k \leq j_2\}$ (modulo logical equivalence).

Abstract Interpretation

$\mathbb{T}_{i_1,j_1 \rightarrow i_2,j_2}$ is an abstract interpretation of $\zeta(\Delta_{i_1,j_1 \rightarrow i_2,j_2})$ with concretization operator $\gamma : \mathbb{T}_{i_1,j_1 \rightarrow i_2,j_2} \rightarrow \zeta(\Delta_{i_1,j_1})$ given by:

$$\gamma(\phi) = \left\{ \delta \in \Delta_{i_1,j_1 \rightarrow i_2,j_2} \;\middle|\; \begin{array}{l} \text{for all } \sigma \in \Sigma_{i_1,j_1} \text{ s.t } \delta(\sigma) \text{ is defined} \\ \check{tainted}(\sigma) \cup \hat{tainted}(\sigma) \models \phi \end{array} \right\}$$

The meaning is that given a $\phi \in \mathbb{T}_{i_1,j_1}$ corresponding to an abstraction of $\zeta(\Delta_{i_1,j_1})$ its reverse corresponds to all the bytecode instructions such that: for all the states in $\Sigma_{i,j}$ set of starting points states s.t. $\delta(\sigma)$ is defined, then the input set of taintedness of σ union the output set of the instruction computed over σ makes true the formula ϕ.

Example. $(store\ k\ t)^{\mathbb{T}} = U \wedge \neg\check{e} \wedge \neg\hat{e} \wedge (\check{s}_{j-1} \leftrightarrow \hat{l}_k)$ means that a store bytecode leaves untouched the exception taintedness and if the value on the stack that

is going to be stored is tainted, also the k-th local is going to be tainted, and vice-versa. U stands for unchanged and means that the output of the bytecode is the same given in input, both for locals variables and for the stack. Of course in case of exception the output stack is formed only by the exception thrown. Formula $\phi = ((\check{l_1} \leftrightarrow \hat{l_1}) \wedge (\check{l_2} \leftrightarrow \hat{l_2}) \wedge (\check{l_3} \leftrightarrow \hat{l_3}) \wedge \neg\check{e} \wedge \neg\hat{e} \wedge (\check{s_0} \leftrightarrow \hat{l_0})) \in \mathbb{T}_{4,1\rightarrow 4,0}$ is true by the bytecode *store* 0 since it does not modify l_1, l_2, l_3 as requested by ϕ, no exceptions are threw before and during the execution of the bytecode, and the output l_0 is coming from the topmost value on the stack. These are the base concepts for the theory, starting from them all other bytecode instructions are defined, and the abstraction of the operators `;`, $\cup$ and `extends` for method calls.

Chapter 3

Intent Analysis

3.1 Sending and Receiving Intents

Depending on the component an intent could be sent to a desired destination. This could happen explicitly or implicitly. No matter which mode is chosen, the parts involved in the communication are: the sender, the intent and the receiver. The sender is the component where the intent is created, or, as we will see later, where the intent comes from. The intent contains the destination address of the receiver if it is explicit, or the action to perform if implicit. The creation of an intent is the same for explicit or implicit intent, except that in the first case the destination address is clearly reported, in the other, the action is reported. When the intent is composed, it is sent to the receiver. The instruction that allow the intent to be sent is different for each component, but produce the same effect. Otherwise the procedure done by the receiver, is different from one component to another.
Furthermore in case of activity this is performed through a method of class `Activity`, that is `getIntent()`. The method needs no parameters and then cannot be tuned. The intents received by any activity performing the `getIntent()` could be sent by:

- A source that clearly sent the intent to the destination activity. This can be done through method `setClass(Context c, Class<> c)` of the class `Intent`. This method is used to built an explicit intent by setting the destination class as the second parameter of the method.
- Any source asking for an action that the receiver can perform. Here the action can be attached to an intent using `setAction(String s)` method of the class `Intent`.

```
class MyActivity extends Activity {
   public void onCreate(...){
      Intent i=new Intent();
      i.setClass(getContext(),MysecondActivity.class)
      startActivity(i);
   }
}

class MySecondActivity extends Activity{
   public void onCreate(...){
      Intent i=getIntent();
   }
}
```

Listing 3.1: Case Activity: the intent is received through the invocation of `getIntent()`. The intent is sent by `MyActivity` and received by `MySecondActivity`

In case of services, the intent receiving process belongs completely to the OS. There are two classes for creating a Service: `Service` and `IntentService` (the difference is that only the first one supports multi-threading). Those two classes can be extended to create a starting service. In order to get the base of the analysis is not necessary to understand completely the entire life-cycle of a service, but we can focus only on how the communication is performed. In case of `IntentService` is necessary to implement `protected void onHandleIntent(Intent intent)` that will be called every time the service is requested. In case of `Service` there is a more complex structure, but when the `Service` is requested, the OS calls first time `onCreate()` method if was not already done, and then
`public int onStartCommand(Intent intent, int flags, int startId)` .
In both cases the initialization of the service is done by passing an intent to the method. This intent is created outside the `Service` and from the moment when it is received is not possible to go back from the receiver (the service) to the sender, since the intent when it is sent is serialized, and then deserialized when the receiver method is called. The same is for `BroadcastReceiver`.

```java
class MyActivity extends Activity {
   public void onCreate(...){
      Intent i=new Intent();
      i.setClass(getContext(),MyIntentService.class)
      startSarvice(i);
   }
}

class MyIntentService extends IntentService{
   protected void onHandleIntent(Intent intent) {
      \\avaiable intent
   }
}
```

Listing 3.2: Case Service: the explicit intent is received by MyIntentService.

```java
public class MyReceiver extends BroadcastReceiver {
   public MyReceiver() {
   }

   @Override
   public void onReceive(Context context, Intent intent) {
      // This method is called when MyReceiver is receiving a broadcast
          intent.
   }
}
```

Listing 3.3: Case BroadcastReceiver: the intent is sent by the OS.

```xml
<receiver android:name="MyReceiver">
  <intent-filter>
    <action
        android:name="android.intent.action.BOOT_COMPLETED">
    </action>
  </intent-filter>
</receiver>
```

Listing 3.4: Manifest of the component `MyReceiver` (Listing 3.3).

Since in the manifest it is declares that the BroadcastReceiver `MyReceiver` could manage the action `"android.intent.action.BOOT_COMPLETED"`, when the telephone completes the boot process, the operating system sent an intent to `MyReceiver` since it is registered to listen this particular action.

3.2 Intents Creation

The first procedure performed by the analysis is to build a model containing any intent founded in the analysed application. This process consist of identifying every intent constructor in the application, and retrieving any related information. An application can manage many different flows of intents. An intent could be created in the heap using one of the constructors defined in the class `Intent`, both by the application under analysis or by the operating system. After that the intent can be filled and sent. Based on the receiving component the intent is retrieved through the receiving method. Intents received through `getIntent()` are called intent from method. At the end, like in Services, intent received through parameters are considered.

3.2.1 New Intent

The first step of the tool is to collect any creation point of a simple intent done by the new operator. Anyone is considered as starting point both for the communication, and for the flow analysis. It means that they cannot receive information by any other intent. It is not possible to assert the same for the copy constructor, since its definition is derived from a previous intent because it receives an intent as an input parameter and it retrieves all information from it. The analysis starts with researching in the application all the simply constructors. As said before, they are considered starting point of the analysis since the intent they are going to built is not related to any previous one. For each collected constructor, thanks to Julia and to forward analysis, is possible to discover any bytecode call to methods that receives as input parameter the intent. All intent modifiers of the intent are discovered and collected in order to obtain all possible destinations of the intent.

3.2.2 Intent from Method

Once simple constructors are collected, the next step is to analyse `getIntent()` method call [8]. Communications between activities happens through intents. An intent is filled by the sender activity, then the destination activity address is set, and finally the intent is sent. From the receiver side, a `getIntent()` call is performed. The explicit or implicit intent is received and data is retrieved. In the receiver activity, the `getIntent()` method is considered as the constructor of the intent. It is considered as the starting point of an intent, but unlike before, the intent have to collect all the information carried by the sender and add them to itself. After that again the research of modifiers is performed. Communication between activities is performed through `startActivities(Intent i)` method of the class `Activity`, and `getIntent()` method of the same class.

3.2.3 Intent from Parameter

The intents that belong to services and background receivers are analysed. Here by default, the intent is received implicitly as one parameter of a method. All methods of the application where the list of parameters contains one intent are collected. Those kind of intent are generic, since they are simply received. Those kinds of intents, like intents from method, have no initialization point,

they cannot exist without an intent created by a simple or copy constructor, and this is why they are analysed after simple intents.

3.2.4 Derived Intent

At the end we considered derived intent. As said before derived intents are created only through constructor `Intent i=new Intent(Intent j)`. When one of these constructors is found thanks to backward analysis, it is possible to retrieve all the bytecode instructions that produced the first parameter of the constructor. Three possibilities are considered by the tool, in order to collect all information about the parameter intent:

1. The intent given in input is produced by a `new` instruction.
2. It is produced by an intent coming from a method, in particular `getIntent()` method.
3. The tool assumes that the intent must be produced by an intent passed as parameter, if the method where the copied constructor is founded receives as input parameter at least an intent. This is considered the last option, since is not sure that it happens. If there is no intent in input of the method, the tool lost information given by the intent parameter and then it is not fully sound.

3.3 Intent Model

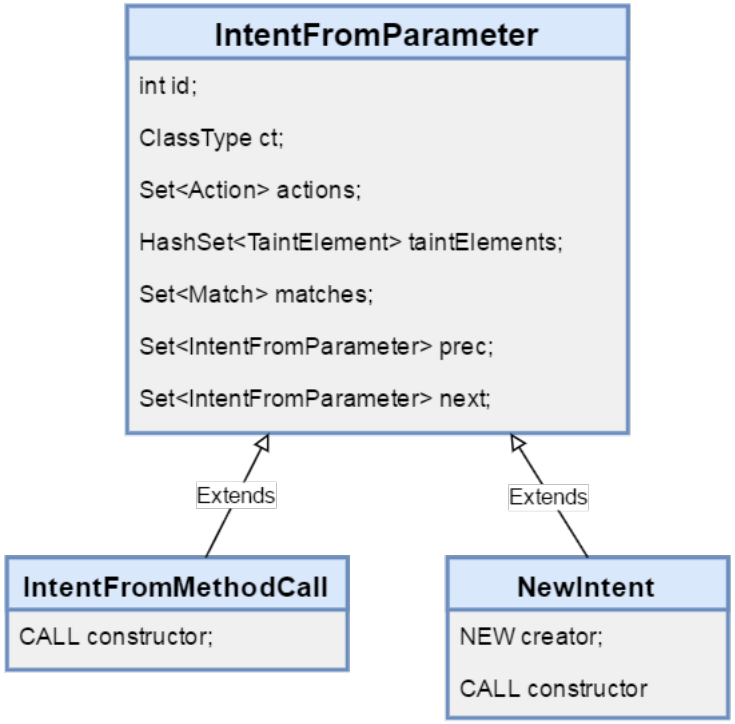


Figure 3.3.0.1: The intent model reflects the three kinds of intent analysed. In particular any intent own a reference to a list of destinations of the outflow, and receives data from a list of incoming intents.

Figure 3.3.0.1 describes the model used in the tool to implement the intent mechanism described before. The attributes that are common to all intents are:

- Any intent is founded inside an application class. Despite its creation an intent can be connected to the application class where it is found, or more precise to the method.
- The application class, alias component, can perform some actions defined by its intent filter in the Android manifest. Any action that the component can handle must be exposed in the Android manifest through the intent filter tag. For new intents this set is always empty.

Both attributes are used in the model and correspond to the `ClassType`, and to the `Set<Action>` reported in Figure 3.3.0.1.

3.3.1 Modifiers in the Model

We have to collect all modifiers of any intent, in order to track every possible path that the intent can traverse.

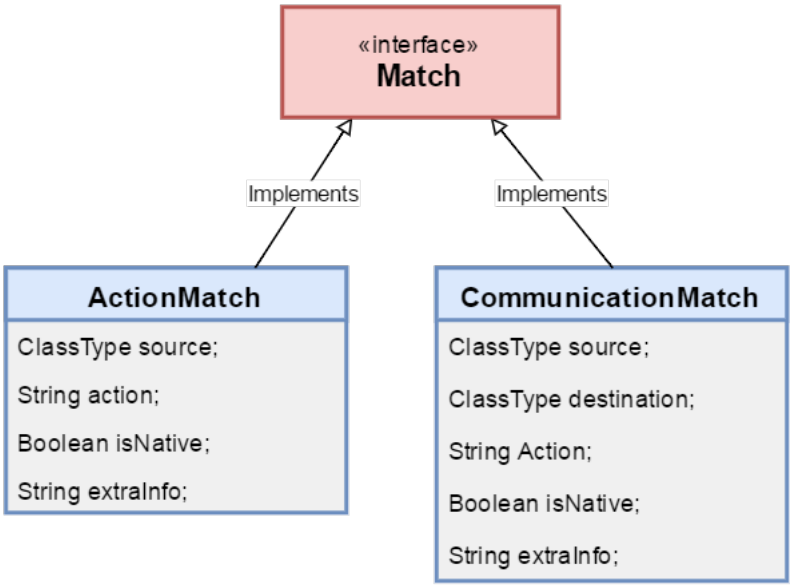


Figure 3.3.1.1: Model representation of the matches, divided in: implicit(ActionMatch) and explicit(communicationMatch)

The model mirrors the intent definition described in Android applications: the definition is splitted in explicit and implicit intent.
ActionMatch (alias implicit intent) are composed by:

- `source`: the source Class, where the modifier method is found.
- `action`: the action requested to the receiver of the intent.
- `isNative`: if the intent is the original owner of the modifiers, or if it has been propagated during the flow and then is propagated.
- `extraInfo`: if some information are not constant, the tool memorized a string information about what is found instead of a constant.

The `CommunicationMatch` contains also the destination class of the intent, since it is the alias of explicit intent.

3.3.2 Intent Flow

Once all intents are collected, the connection between them is inferred. Remember that only intents from parameters and method calls can be connected to a sender, since they are the only ones without creation point. The connection is performed through the method reported below in Listing 3.5:

```
public boolean isSenderTo(IntentFromParameter j){
   if (j.getCreator()!=null)
      throw new IllegalArgumentException();
   for(Match m: matches){
      if (m.getActionDestination()!=null &&
          m.getActionDestination().equals(genericAction) &&
          j.getActions().size()!=0)
         return true;
      if (j.getActions().contains(m.getActionDestination()))
         return true;
      if (m.isGenericDestinationClass())
         return true;
      if (m.getClassDestination()!=null && j.getClassType()!=null &&
          m.getClassDestination().equals(j.getClassType()))
         return true;
   }
   return false;
}
```

Listing 3.5: Connection method used to create intents network. The target intent asks if it could comunicate with intent given in parameter input.

First of all, the method (Listing 3.5) checks the stability: if the intent is from parameter or method, then could be the receiver of the communication. For any match owned by the referred intent it is asked if:

- The referred intent owns an action that was over-approximate as the generic destination, and the destination intent is registered in the Android manifest to hold at least one action. This is an approximation derived from the imprecision of the address used in the match. This approximation produces an huge amount of false communication, since any activity in order to perform a work, should define an action in its manifest.
- The action filtered by the destination intent contains the sender action, that is a perfect match.
- The source intent owns an explicit destination address but it is unreachable. Then generic destination address is considered, meaning that the sender could write to any component inside the application, is the same concept defined before. Both the adopted approximations were improved to test on real software.
- The communication is explicit between the parts.

If one of these cases happens than the referenced intent is a sender. Then a new edge connects the two parts. When the match happens, based on the

configuration, all matches from the sender should be copied into the receiver. If a connection happens, basically two operations are performed:

- Match copy: since an intent can contain many actions and class destination addresses, it should share with the receiver those addresses, since from now the receiver can perform any operation carried by the sender. Then, all matches carried by the sender are copied into the receiver. The tool keeps track that those copied data are not native.
- Taintedness copy: all taints information carried by the intent are copied inside the receiver intent. This is the taintedness flow of the tool. No matter if the receiver extract from the intent the taintedness, but in order to expand the analysis taintedness are copied.

Once the connections are made, copied constructor are updated. The update differs from the connection since in does not connect intents but it performs only coping operations, that involves matches and taintedness.

3.3.3 Intent Extra

In parallel to the connection procedure we perform the research of fillers and getters. This is the top of the iceberg of the analysis, since it is performed over a network of intents that is an over approximation of the real communication. Fillers set is composed by all the methods that attach additional information to the intent. In the analysis we defined a new annotation used to recognize any method of this set: `@WriteIntentExtra`. The filler procedure reasearch is composed by two main operations:

- It looks for every bytecode, that is a call to a method marked as `@WriteIntentExtra`. This method is considered dangerous, since it fills the intent with additional information.
- Once a method is found, the Julia flow analysis is enquired about the data being filled. If taint information is involved the tool keeps track of the intent being filled and the key set used to do that.

The getters are all the methods that extract information from an intent. For those methods we introduce a new notation: `@ReadIntentExtra`. The getter procedure research is composed by:

- The research of any method call marked as `@ReadIntentExtra`.
- If the get operation is performed through a key, it looks if the intent holds this key. Intents in the model hold only tainted keys. Any time a match between keys occur the retriever memorize itself into the destinations set 3.3.3.1. In this way at the end of the analysis can be tracked the flow to get how the intent travers the components. In particular, it is memorized also the mode in which data are extracted.

As shown in Figure 3.3.0.1 any intent, is in a one to many relation towards `TaintElement` class. In the model fillers and getters are memorized in the structure in Figure 3.3.3.1.

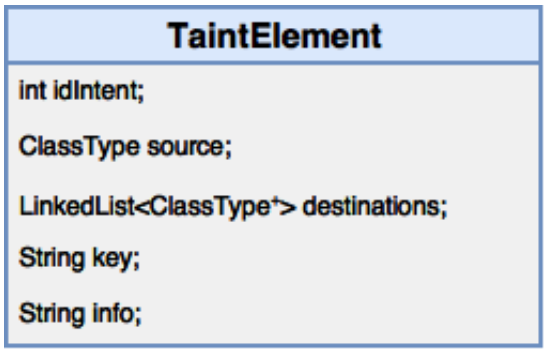


Figure 3.3.3.1: Any time a method annotated as `@WriteIntentExtra` adds to an intent tainted data, a new `TaintElement` is created, storing the source class where the addition is performed. Moreover the key represent the label associated to the taintedness. When a method annotated as `@ReadIntentExtra` extract tainted data from an intent, it adds itself to the destination list. It memorizes if data are extracted clearly or not, in fact any destination is composed by class name and boolean checker.

3.3.4 Correctness of the Analysis

Main limitations of Julia analyzer are the following:

- When a method is called using reflection pattern the analysis discards the call.
- In case of multithreading computation, the analysis serializes the operations as they were done one after the other, without any concurrency.

Moreover are discussed the main limitations of our intent analysis:

- The `getfield` and `putfield` bytecode instructions produce the interruption of the intent analysis, mainly caused to the block of the forward and the backward analyses. This limitation is partially solved using Creation Point analysis, that overcome this limitation but introducing few approximation. It follows the limitation of creation point analysis, looking for the creation point of an object limited to the same method or classes.
- Again the analysis makes an assumption in case of intent from parameter. What happens is that when a modifier or a filler method is matched, the analysis tries to find the creation point of the target intent. If no creation points is found, the analysis check if the method where the modifier happens receive an intent as input parameter. If it is true, the modifier is connected to that intent 3.6.

```
class MyActivity extends Activity {
  Intent field;
  public void onCreate(...){
    field=getIntent();
  }
  public void myMethod(Intent j){
    TelephonyManager telephonyManager = (TelephonyManager)
        getSystemService(Context.TELEPHONY_SERVICE);
    String imei = telephonyManager.getDeviceId(); //source
    field.putExtra("KEY",imei);
  }
}
```

Listing 3.6: shows a situation where the analysis makes a mistake. What happens is that the call to `putExtra` *method is performed on the* `Intent field` *but the result is that* `Intent j` *is modified. This happens since* `Intent field` *has no creation point in the class* `MyActivity`*, moreover the method* `myMethod` *has a parameter of type* `Intent`*.*

Chapter 4

Configuration Settings

In this chapter we show the sources and sinks annotations adopted into the intents flow analysis. After that is showed how to execute Julia in order to perform the intent analysis, that consist into setting the parameters in input to the Julia analyzer. Following sections are more technical and can be used as user guides. There are considered both the cases where the source code is available, and then JARs is given in input to the analysis, or just the APK is available and then must be used an intermediate step for reverse it. Both the situation are described: there is a section about Android Studio[5] where are described the compile results produced by the IDE and the directory tree built after the compilation, and following another section about JuliaDexConverter[25] where are showed the modules composing it.

4.1 Julia

4.1.1 Annotations

Our analysis is based in particular on `InjectionChecker` and the following annotations:

- `@Untrusted`: it signs any method that returns a sensitive information. The method `String getDeviceId()` is marked as `@Untrusted` since it returns the IMEI code of the device.
- `@Trusted`: it is applied to any method parameter that has to receive only secure information. In the method `Intent putExtra(String key, @Trusted String value)` the parameter `String value` is marked as trusted, it means that only secure information has to be given in input. A warning will be raised by the analysis if an untrustable information is given in input to a trusted parameter.
- `@ReadIntentExtra` is an annotation attached to any method able to retrieve information from an intent. The method `@ReadIntentExtra char[] getCharArrayExtra(String name)` retrieves from an intent data associated to the string key `name`.

- `@WriteIntentExtra` is an annotation applied to any method able to fill information into an intent. The method `@WriteIntentExtra Intent putExtra(String key,@Trusted CharSequence value)` is a filler method.

4.1.2 Run Julia

Julia is executed from the core-project. The analyzer allows to set various parameters, in order to perform different kinds of analysis and checks different properties.
For our analysis the following configuration is adopted:

- `-framework androidAPI19` is the last Android framework supported by Julia. The framework is the mechanism that allows to tune up methods signatures in order to mark, add, and modify, additional information to return type and parameters. It is composed by a set of signatures of methods wrapped on the original ones.
- `-Injection` is the checker adopted in order to identify potential Injection attacks.
- `-mergeArrays` during the analysis merges the arrays of the same type.
- `-i android.` it includes into the analysis the Android classes but does not warn on them. They are included in order to perform correctly the analysis.
- `-i java.` it includes into the analysis the Java classes but does not warn on then. They are included to perform correctly the analysis.
- `-si ..\Telegram\Telegram.jar` is the jar to analyse.
- `-Warnings` returns the warning produced by the analysis.

4.2 Android Studio

Android Studio is the official IDE for developing and debugging Android applications[5]. In order to perform the analysis we need the Java bytecode, or from the APK of the application a reverse tool like Dex2Jar [13]. The source code of the bytecode under analysis allow us to check results given by the analyzer. The first step was then to study how Android Studio builds a project in order to retrieve `.class` files before the APK is made.
Android Studio is based on Gradle that is a build tool. Gradle allows to be extremely customized, indeed each project has its private configuration, that first of all set the `compileSDKVersion` and the `buildToolsVersion`. It is possible to install Gradle locally on the machine, and then compile the project using it, but each time should be adapted to the project to build and moreover it is not backward compatible. It is better to use the wrapper `Gradlew` included into each project, since it is made in order to be suited to the application to built.
After building, the folder `ProjectName\app\build\intermediates` is created in the root of the project.

Inside it can be found:

- `classes` contains all the `.class` files of the project, libraries and application code.
- `exploded-aar` contains all the JAR archives of the library needed to the application.
- `manifest` contains the application manifest.
- `packaged` contains the JAR of the application, including app classes and library classes. The analysis is performed on this JAR, where in addition was included the manifest contained in the folder `manifest`.

4.3 JuliaDexConverter

JuliadDexConverter[25] is a tool that performs the following operations:

- It reverts `.dex` file into Java `.class` files. This is done thanks to Dex2Jar[13] tool that re-build `.class` files from the `.dex`. The adopted Dex2Jar is modified[25] with respect to the standard version, since it tries to preserve debug information. Those are useful to get precise results from the analysis.
- It transforms binary representation of the XML in a readable version of the AndroidManifest.xml. ApkTool [6] performs this.

At the end two JAR archives are produced and while the first one containing the application the second one contains the libraries. Thanks to that the Julia's parameters configuration runs can be splitted in `-si` for the application JAR and `-i` for the library JAR, allowing to discard warnings retrieved from the libraries.

4.3.1 Dex2Jar

Any APK downloaded from the Google Play Store contains just one `.dex` file that includes all classes of the application, the manifest of the application and resources. Dalvik Virtual Machine is a service implemented into the Android OS that executes dalvik code memorized into .dex files. Basically the Dalvik Virtual Machine is used to reduce memory spaces occupy by standard JVM. Dex2Jar is a reverse engineering tool that revert an Android application APK. Dex2Jar is adopted in our analysis since Julia can analyse only Java bytecode, and thanks to this tool, we are able to analyse any Android application downloaded from the store. In general the translation is not perfect, few approximations are introduced, but during the tests we noticed that the approximation never produces new false positive tainted data.

Chapter 5

Results

This section is about the results produced by the intent analysis. It starts with the hardware configuration used to execute the analysis and the parameters values given as input to Julia. Results are showed thanks to dot representation through Graphviz[21]. There is a description of the results, where for each analyzed application it is reported the Android manifest, the significant part of source code, and it is then described the resulting flows reported by the analysis. The analysis produces both the dot and the verbose representation of the flow, specifying for each intent the communication in which is involved. Any false positives and missing leaks not returned by the analysis are later discussed.

5.1 Hardware Adopted

Test are executed on a HP EliteDesk 800 G2 SFF with the following features: Intel Core i5-6600 CPU @3.30 GHz, 16GB RAM,
running Operating System Windows 7 Professional 64 bit.
Julia is runned from the core project, using the IDE Eclipse Mars.2 Release (4.5.2).
Telegram leaks are reproduced on the smartphone device HUAWEI P9 Lite with the following features: Kirin650 Quad-Core 2.0GHz 64-bit, 3GB RAM
running Operating System Android 6.0 + EMUI 4.1.

5.2 Dot Representation and Results Interpretation

The graphical results are obtained using dot representation, through which we can represent the application's intent flow. Each shape describes an intent with known identificator and the application class where it is defined. The meaning of the shapes is explained in the following:

Figure 5.2.0.1: Shows a yellow square shape. It means that the intent receives taint data but the component holding the intent does not extract those data. If it has any out edges it means that the intent is a carrier, otherwise the intents communication stops here.

The intents are represented by yellow rectangles when they are carriers of taintedness. It means that it receives taintedness from another intent and it carries them.

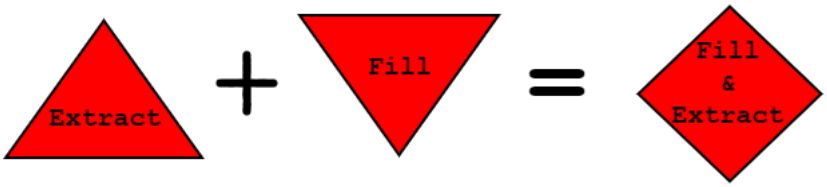


Figure 5.2.0.2: Shows the graphical representation of tainted data extracted from an intent (labelled as Extract), filled into an intent (labelled as Fill) and both operations performed on the same intent (result).

Red shapes correspond to the most dangerous intents: the shape signed as `Extract` means that taint information are extracted from the intent, while the shape labelled as `Fill` means that taintedness is added to the intent. An intent performing both operations is represented by the resulting square in Figure 5.2.0.2.
An edge connecting two nodes means that:

- The components holding the two intents satisfies the rules of communication. The communication involves only taint data, any arrow between two entities means that a sensitive information moves from the sender intent to the destination ones.
- The label reported on the edge is the constant string key used both during the `putExtra` and `getExtra`. It means that taint data are associated to the reported string key. In case of unreachable key, the generic key is reported.

5.3 DroidBench

Droidbench[14] benchmark is used to tests our analysis. It is composed by many applications each one reproducing a security issues that should be identified by the analyser. In our analysis we focused on intent communication, then only few applications are considered. Applications from both the branches `master` and `Epicc` are chosen to test our analysis. We start with the applications of the branch `master`.

5.3.1 ActivityCommunication2

This application is composed by three Activities (Figure 5.3 5.4 5.5):

- The first one called `OutFlowActivity` 5.3 creates a new implicit intent and sets the action parameter. The tool is not able to retrieve the full constant since it is a dynamic string, and then approximates it. The match is reported here:
  ```
  Imp.Com.:OutFlowActivity-->:(ACTION) ***
  Native:true
  Destination action not reachable:
  call java.lang.String.substring(int) :java.lang.String
  public java.lang.String.substring(int):java.lang.String
  ```
  The meaning is the following: it is founded an implicit communication between the component `OutFlowActivity` and an action. The analysis collects a string representation of what is found instead of a constant.

- `InFlowActivity` receives the intent sent by `OutFlowActivity` since it performs a `getIntent()` operation and in the manifest it declares at least one intent filter containing one action string: this means that the `genericAction(***)` sent by `OutFlowActivities` could be received by `InFlowActivities`.

```
<intent-filter>
  <action
      android:name="edu.mit.icc_action_string_operations.ACTION"
      />
  <category android:name="android.intent.category.DEFAULT" />
</intent-filter>
```

Listing 5.1: the InFlowActivity Manifest.

- `IsolatedActivity` performs a `getIntent()` and it declares an intent filter with at least an action in the manifest. Then could be the receiver of the generic action.

```
<activity
    android:name="...icc_action_string_operations.IsolateActivity"
    android:label="@string/app_name" >
  <intent-filter>
    <action
        android:name="edu.mit.icc_action_string_operations.EDIT"
        />
    <category android:name="android.intent.category.DEFAULT"
        />
  </intent-filter>
</activity>
```

Listing 5.2: the IsolatedActivity Manifest

```java
public class OutFlowActivity extends Activity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      //omitted details
      TelephonyManager telephonyManager = (TelephonyManager)
          getSystemService(Context.TELEPHONY_SERVICE);
      String imei = telephonyManager.getDeviceId(); //source
      Intent i = new
          Intent("...icc_action_string_operations.ACTION".substring(7));
      i.putExtra("DroidBench", imei);
      startActivity(i);
      //omitted details
   }
}
```

Listing 5.3: the OutFlowActivity source code

```java
   public class InFlowActivity extends Activity {
      @Override
      protected void onCreate(Bundle savedInstanceState) {
         //omitted details
         Intent i = getIntent();
         String imei = i.getStringExtra("DroidBench");
         Log.i("DroidBench", imei); //sink leak
      }
   }
```

Listing 5.4: the InFlowActivity source code.

```java
   public class IsolateActivity extends Activity {
      @Override
      protected void onCreate(Bundle savedInstanceState) {
         //omitted details
         Intent i = getIntent();
         String imei = i.getStringExtra("DroidBench");
         Log.i("DroidBench", imei);
      }
   }
```

Listing 5.5: the IsolateActivity source code

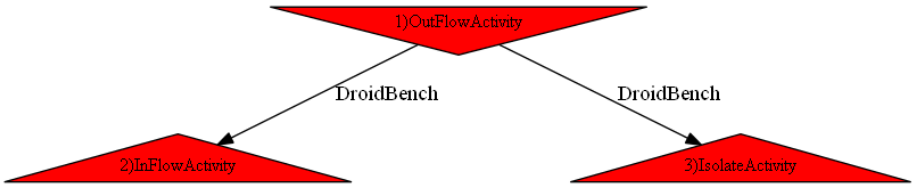


Figure 5.3.1.1: the dot representation of ActivityCommunication2

The Figure 5.3.1.1 is the result of the analysis performed. It underlines that the `OutflowActivity` fills the intent (id=1) with taint data associated to the constant string `"DroidBench"`. Both `InFlowActivity` and `IsolateActivity` get the intent and extract the taint information from it. The communication between `OutFlowActivity` and `IsolateActivity` is a false positive.

5.3.2 ActivityCommunication4

The application is composed by the same three activities described before in `ActivityCommunication2`. The only activity that has been changed is `OutflowActivity` that is reported here (Listing 5.6).
`OutFlowActivity` creates a new implicit intent and sets the action parameter as the concatenation beetwen two constant strings. The compiler optimizes the operation of concatenation. The result is that the string is correctly matched by the analysis and the result is correct.

```
public class OutFlowActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    //omitted details
    TelephonyManager telephonyManager = (TelephonyManager)
        getSystemService(Context.TELEPHONY_SERVICE);
    String imei = telephonyManager.getDeviceId(); //source
    Intent i = new Intent("edu.mit.icc_concat_action_string" +
        ".ACTION");
    i.putExtra("DroidBench", imei);
    startActivity(i);
    //omitted details
   }
}
```

Listing 5.6: the OutFlowActivity source code

In Listing 5.6 there is the source code of the class `OutFlowActivity`, then in Listing 5.7 there is the result of the compiler procedure.

```
26 new #9 <android/content/Intent>
29 dup
30 ldc #10 <edu.mit.icc_concat_action_string.ACTION>
32 invokespecial #11 <android/content/Intent.<init>>
```

Listing 5.7: the optimization done by the compiler, it merges together the two constants. `invokespecial` at offset 32 correspond to the call to intent's constructor.

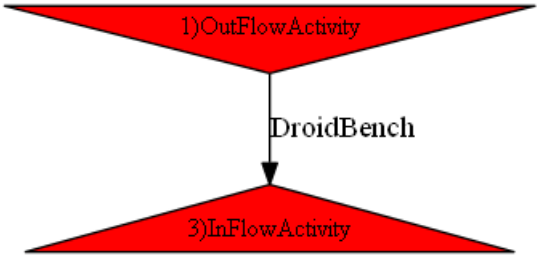


Figure 5.3.2.1: the dot representation of ActivityCommunication4

The Figure 5.3.2.1 is the result of the analysis performed. It underlines that the `OutFlowActivity` fills the intent (id=1) with taint data associated to constant string `"DroidBench"`. Intent (id=3) belonging to the class `InFlowActivity` receives it correctly and extracts the tainted data. Differently from `ActivityCommunication2` here the `IsolateActivity` is not involved in the flow since here the action is not overapproximated but is fully matched.
Same result is obtained when the following source code is analysed, corresponding to `ActivityCommunication5`:

```
public class OutFlowActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    TelephonyManager telephonyManager = (TelephonyManager)
        getSystemService(Context.TELEPHONY_SERVICE);
    String imei = telephonyManager.getDeviceId(); //source
    ComponentName cn = new ComponentName(this,
        "edu.mit.icc_intent_component_name.InFlowActivity");
    Intent i = new Intent();
    i.setComponent(cn);
    i.putExtra("DroidBench", imei);
    startActivity(i);
  }
}
```

Listing 5.8: in this application the destination class is initialized through the creation of an external item called `ComponentName`, which is applied over the intent. The result is exactly the same as Figure 5.3.2.1, furthermore the backward analysis is able to go back from the modifier `setComponent` until its creation point.

The dot representation is the same as reported in Figure 5.3.2.1.
The application is modified in order to integrate the `IsolatedActivity` in the dot result. This is done by adding an intent filter to the Android manifest, related to the correct component (Listing 5.9).

```
<activity
    android:name="...icc_concat_action_string.NoMoreIsolateActivity"
    android:label="@string/app_name" >
  <intent-filter>
    <action android:name="edu.mit.icc_concat_action_string.ACTION"
        />
    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
</activity>
```

Listing 5.9: `NoMoreIsolateActivity` adds an intent filter to its manifest.

The activity has been renamed as `NoMoreIsolateActivity`. The source code of the application is modified as follow: in `OutFlowActivity` a clean value is attached to the intent, in addition to the tainted one.

```
public class OutFlowActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    TelephonyManager telephonyManager = (TelephonyManager)
        getSystemService(Context.TELEPHONY_SERVICE);
    String imei = telephonyManager.getDeviceId(); //source
    Intent i = new Intent("edu.mit.icc_concat_action_string" +
        ".ACTION");
    i.putExtra("NotTainted ","Clean Value");
    i.putExtra("DroidBench", imei);
    startActivity(i);
  }
}
```

Listing 5.10: the modified version of OutFlowActivity. Intent i contains both tainted and clean keys.

In NoMoreIsolateActivity only the clean key is retrieved from the intent.

```
public class NoMoreIsolateActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Intent i = getIntent();
   String clean = i.getStringExtra("NotTainted");
   Log.i("NotTainted", clean);
  }
}
```

Listing 5.11: the source code of NoMoreIsolateActivity. Only clean key is retrieved from the intent.

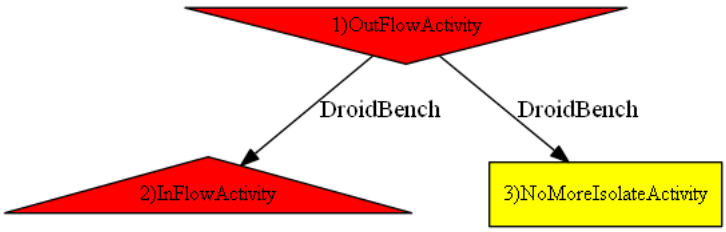


Figure 5.3.2.2: the dot representation of ActivityCommunication4-Modified

The dot in Figure 5.3.2.2 is changed since NoMoreIsolatedActivity has become part of the graph. Now it is involved in the communication due to the correction done in the manifest. NoMoreIsolatedActivity is now yellow and assumes rectangular shape, since it is not extracting or inserting tainted data into the received intent, but it is just carrying it.

5.3.3 IntentSink1

The application is composed by a single activity. It receives an intent by user interactions or by the OS.

```
public class IntentSink1 extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_intent_sink1);
    TelephonyManager telephonyManager = (TelephonyManager)
        getSystemService(Context.TELEPHONY_SERVICE);
    String imei = telephonyManager.getDeviceId(); //source
    Intent intent = this.getIntent();
    intent.putExtra("secret", imei);
    this.setResult(RESULT_OK, intent); //sink, leak
    finish();
  }
}
```

Listing 5.12: the source code of IntentSink1

Note that the intent is received through a call to `getIntent` method, and then is filled using a tainted information. The resulting intent is used as result value in a `setResult` call operation. This method is called exclusively in case of Activity communication, in particular as returning values of `startActivityForResult` method.

Figure 5.3.3.1: the dot representation of IntentSink1. Only one component is contained in the application.

Both the dot representation Figure 5.3.3.1 and the verbose Listing 5.13 output are reported. The intent is received through a `getIntent()` than it is an intent from method. Since the component can hold the action `android.intent.action.MAIN`, the intent inherit the action. The intent is then reported with empty `info` field, since the key `"secret"` is well known.

Textual representation:

```
Intent(getIntent()):1
Def.Class:de.ecspride.IntentSink1
Actions Handled: Act.:android.intent.action.MAIN

It is involved in the following taint elements:
Source: de.ecspride.IntentSink1
Destination: []
Keys: secret
Info:
```

Listing 5.13: the unique intent of `IntentSink1` is an intent from method.

5.3.4 IntentSource1

The `IntentSource1` application is composed by a single activity. There is no taintedness involved in the application as it is, then there is no resulting flow. The application is composed of two methods (Listing 5.14):

- `onCreate`: when the activity is created an implicit intent is sent to any component that can holds the reported action. The intent is sent using the method `startActivityForResult(Intent i,int n)` Figure 5.3.4.1. This method performs the same operation of `startActivity(Intent i)`, except that the receiver of the intent can send back to the sender additional information about the success or failure. The result can carry a result code and an intent for extra information. The backflow is not as normal, since it does not follow the normal rules of communication about actions or destination address. It means that the receiver of the `startActivityForResult` can send back an intent without properly setting the sender address.

- `onActivityResult`: this method is called by the receiver in order to get back to the sender how the operation was performed and its result.

```java
public class IntentSource1 extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_activity1);
    Intent intent = this.getIntent();
    intent.putExtra("KEY",imei); // TAINT
    intent.setAction("android.intent.action.MAIN");
    this.startActivityForResult(intent, 1);
  }
  @Override
  protected void onActivityResult(int requestCode, int resultCode,
      Intent data){
    if (requestCode == 1){
      Bundle b = data.getExtras();
      for (String key : b.keySet()){
        Log.i("SnT", "dump: " + b.get(key));
      }
    }
  }
}
```

Listing 5.14: the source code of `IntentSource1`. In the original version the intent does not contain tant values.

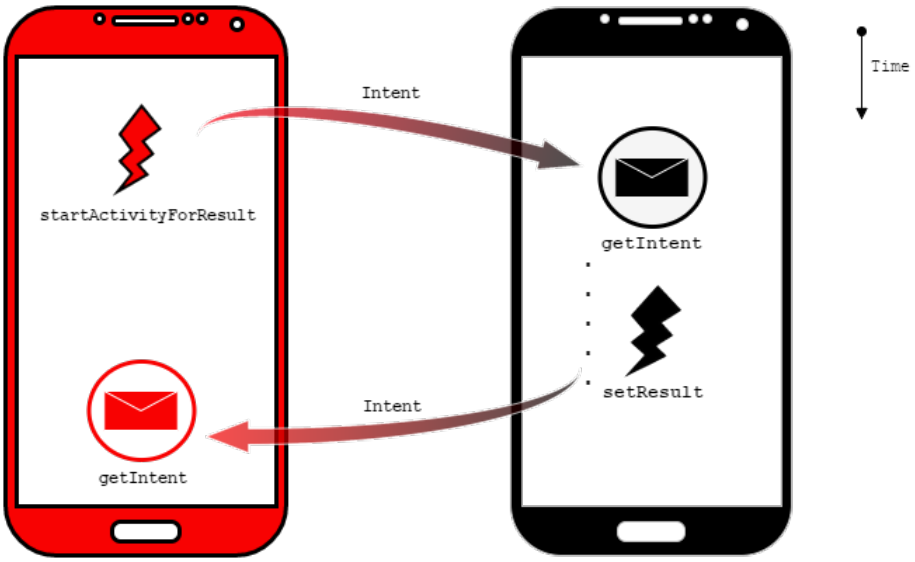


Figure 5.3.4.1: the graphical representation of `startActivityForResult` method. The backflow does not follow the standard communication rules.

Since there is no taintedness flow of information, it means that no dirty values flows between components, the analysis results are empty. Then the taint value is introduced into the intent thanks to a filler method (Listing 5.14). The tool analyses the code and returns the following: the activity desires to start an intent

holding action "android.intent.action.MAIN" and pretends a back result. The IntentSource1 activity can hold the action main, since it is the entry point of any Android application. It corresponds to the activity called when the user taps over the launch icon of an application. The analysis takes a look to the manifest and parses the actions that can be handled by IntentSource1 and matching the requested one. Then it creates the self reference in output from the IntentSource1 to IntentSource1 reported in Figure 5.3.4.2. Since the method onActivityResult receives in input an intent, could happens that the intent given in input is the same received by the application thanks to intent filter, and then is the same sent by onCreate method. This is why a flow between the intent sent in onCreate method and received by onActivityResult is created. The getIntent() method is retrieving an intent sent by the OS after the tap of the user over the launching icon. This input generates a setComponent modifier with destination the app tapped, in particular the activity that holds the action main.

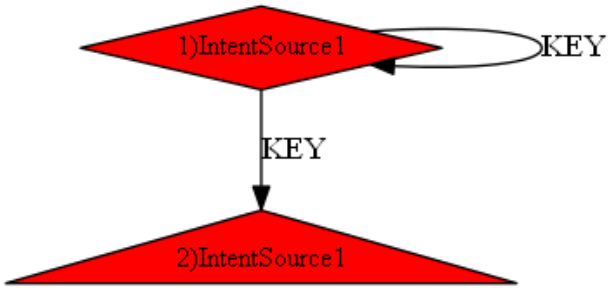


Figure 5.3.4.2: Dot representation of IntentSource1

In order to modify the destination address of the intent it is necessary to apply one of the modifiers both to the destination class and the action. Instead of do that a new intent is built. Only the method onCreate is modified Listing 5.15, while onActivityResult is exactly the same as before.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_activity1);
   Intent intent = new Intent()
   intent.putExtra("KEY",imei);
   intent.setAction("android.intent.action.BUG_REPORT");
   this.startActivityForResult(intent, 1);
}
```

Listing 5.15: the source code of onCreate method, modified to produce a new flow.

We introduced two new receivers of the intent both implementing an intent filter in order to managing the sender action (Listing 5.16).

```
<activity android:name="lu.uni.snt.serval.IntentReceiveExtract">
  <intent-filter>
    <action android:name="android.intent.action.BUG_REPORT"/>
    <category android:name="android.intent.category.DEFAULT"/>
  </intent-filter>
</activity>
<activity android:name="lu.uni.snt.serval.IntentReceivePassing">
  <intent-filter>
    <action android:name="android.intent.action.BUG_REPORT"/>
    <category android:name="android.intent.category.DEFAULT"/>
  </intent-filter>
</activity>
```

Listing 5.16: Here is the XML of the new `IntentReceiveExtract`

The first receiver is `IntentReceiveExtract`, it performs the following code:

```
public class IntentReceiveExtract extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_intent_receive2);
    Intent g=getIntent();
    Bundle b=g.getExtras();
    setResult(Activity.RESULT_OK,g);
    finish();
  }
}
```

Listing 5.17: the source code of `IntentReceiveExtract`. It receives the Intent through a call to `getIntent()`, it extracts the data contained thanks to a call to `getExtras()` that returns a `Bundle`. It calls the callback method `setResult()`, that is implicitly connected to the `onActivityResult`.

The second activity is `IntentReceivePassing`:

```
public class IntentReceivePassing extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_intent_receive_passing);
    Intent g=getIntent();
    setResult(Activity.RESULT_OK,g);
    finish();
  }
}
```

Listing 5.18: the source code of `IntentReceivePassing`. Differently from `IntentReceiveExtract`, the `IntentReceivePassing` receives the intent and carries it until the `setResult` method.

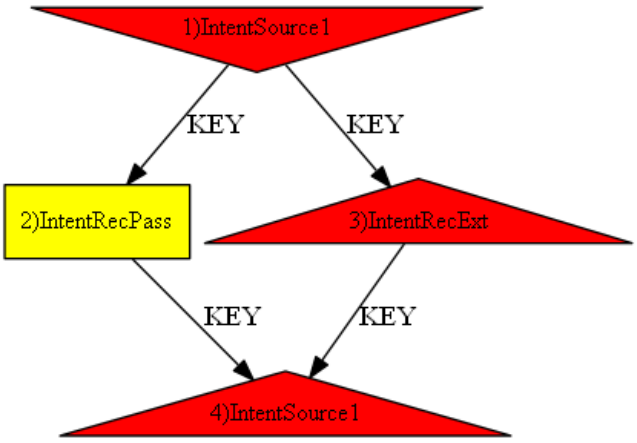


Figure 5.3.4.3: the dot result of IntentSource1(modified version).

`IntentSource1` fills a new intent with tainted information. Taintedness is associated with the key `"KEY"`. The intent is sent to both the activities `IntentReceiveExtract` and `IntentReceivePassing`. In one case the taintedness is simply received and spreaded to the sender using `setResultMethod`, in the other taintedness is also extracted from the intent. When both results go back to the sender, they are extracted and the flow finished.

5.3.5 startService1-Sink

The startService1-sink is an application taken from the branch Epicc[38] of DroidBench Git repository. This branch inspects in depth any situation involving inter communication through intents. This in particular is chosen in order to discuss also `Services` taint flow, and describes pros and cons of the analysis. In the original version the application was composed by two services and one activity. The Services are:

- `InflowService` is a component extending the class `Service`, and override methods `onStartCommand` and `onBind`, both receiving an intent from parameters. In the manifest it implements a valid intent filter.
- `IsolateService` is a component extending the class `Service`, and it is isolated since the manifest does not define any intent filter tag, then it is not reachable, except by explicit intent.

The activity calls the super of the parent and attaches itself to the view. This is clearly shown in the following code:

```
public class IsolateService extends Service {
  @Override
  public int onStartCommand(Intent intent, int flags, int startId) {
    String imei = intent.getStringExtra("Key");
    Log.i("Key", imei);
    return super.onStartCommand(intent, flags, startId);
```

```
  }
  @Override
  public IBinder onBind(Intent arg0) {
    return null;
  }
}
public class InFlowService extends Service {
  @Override
  public int onStartCommand(Intent intent, int flags, int startId) {
    String imei = intent.getStringExtra("Key");
    Log.i("Key", imei);
    return super.onStartCommand(intent, flags, startId);
  }
  @Override
  public IBinder onBind(Intent arg0) {
    return null;
  }
}
public class LaunchingActivity extends Activity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
   }
}
```

Listing 5.19: the source code of startService1-Sink.

```
<activity android:name=".LaunchingActivity"
    android:label="@string/app_name" >
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
<service android:name=".InFlowService"
    android:label="@string/app_name" >
  <intent-filter>
    <action android:name="lu.uni.serval.iac_startservice1.ACTION" />
    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
</service>
<service android:name=".IsolateService"
    android:label="@string/app_name" >
</service>
```

Listing 5.20: the Android manifest of startService1-Sink

The result is that no flow is produced, and then the resulting graph is empty.

The activity component is modified as follows:

```
public class LaunchingActivity extends Activity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      TelephonyManager telephonyManager = (TelephonyManager)
          getApplicationContext().
          getSystemService(Context.TELEPHONY_SERVICE);
      String imei = telephonyManager.getDeviceId();
      Intent i=new Intent();
      i.setAction("lu.uni.serval.iac_startservice1.ACTION");
      i.putExtra("Key",imei);
      startService(i);
   }
}
```

Listing 5.21: the source code of `LaunchingActivity` where it is introduced a taint extra and an implicit match.

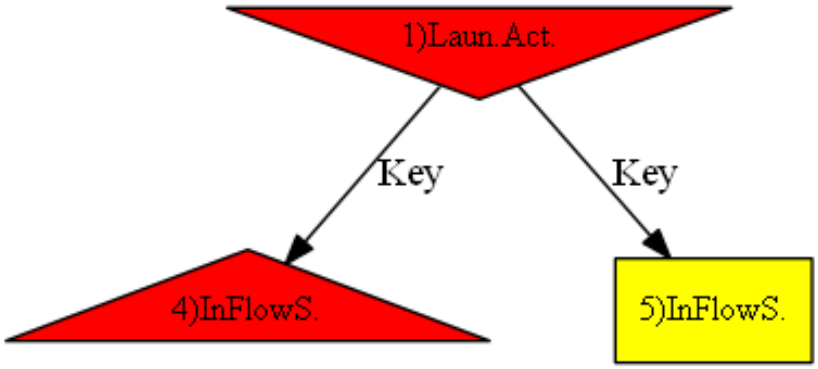


Figure 5.3.5.1: the dot representation of startService1-Sink. In the picture there are involved only two components: `LaunchingActivity` and `InFlowService`

In Figure 5.3.5.1 the communication between components is overapproximated. What happens is that the flow correctly excludes `IsolateService` from the results, then only two components are involved. Since `InFlowService` is composed by two methods both receiving an intent from parameter, the intent sent by `LaunchingActivity` could be received by both the methods. The difference is that `onStartCommand` extracts the taintedness from the intent, instead `onBind` interrupt the flow.

The verbose representation is reported, here there is the intent from parameter of the method `onStartCommand`:

```
Intent: 4
Def.Class: InFlowService
Def.Method: onStartCommand
Action Handle: Act.:...serval.iac_startservice1.ACTION

It is involved in the following taint elements:
Source: 1) LaunchingActivity
Destination: [4) InFlowService, extract generic:false]
Keys: Key
Info:

It is connected with the PREC Intents:
1 lu.uni.serval.iac_startservice1_sink.LaunchingActivity
It is connected with the SUCC Intents:
(Empty)
```

Listing 5.22: Verbose representation flow of the intent number four. The definition method discriminates the two intent from parameter. Generic extraction means that the tainted data are extracted through a well known key and not by a `getExtras` method calls.

The following is the verbose representation of the intent from parameter of the method `onBind`:

```
Intent: 5
Def.Class: InFlowService
Def.Method: onBind
Action Handle: Act.:...serval.iac_startservice1.ACTION

It is involved in the following taint elements:
Source: 1) LaunchingActivity
Destination: [4) InFlowService, extract generic:false]
Keys: Key
Info:

It is connected with the PREC Intents:
1 LaunchingActivity
It is connected with the SUCC Intents:
(Empty)
```

Listing 5.23: the verbose representation flow of the intent number five. Note that destination field contains intent number four.

5.3.6 Summary Table

Application	Lines	Time	Mts	TP	FN	FP	Prec.	Rec.
Act.Com. 2	32558	01:27	6094	1	0	1	0.5	1
Act.Com.4	32558	01:26.	6094	1	0	0	1	1
Act.Com.5	32559	01:29	6094	1	0	0	1	1
Int.Sink	32539	01:25	6088	0	0	0	/	/
Int.Source1	32575	01:26	6093	1	0	1	0.5	1
Int.Source1.1	32720	01:25	6113	4	0	0	1	1
Service1-Sink	48250	01:54	7955	1	0	1	0.5	1

Table 5.3.6.1: Summary table of DroidBench applications. The analysis always finds all the existing communication flows (Recall equals to one). `Lines` *stands for number of lines analysed,* `Mts` *stands for methods number. After that there are TP(true positive), FN(false negative), FP(false positive), Precision*($\frac{TP}{TP+FP}$) *and Recall*($\frac{TP}{TP+FN}$).

5.4 Real Case Study: Telegram

Telegram[27, 18] is a messaging application focused on speed and security features. You can use Telegram on all your devices at the same time your messages sync seamlessly across any number of your phones, tablets or computers [27]. With Telegram, you can send messages, photos, videos and files of any type (doc, zip, mp3, etc), as well as create groups for up to 5000 people or channels for broadcasting to unlimited audiences. You can write to your phone contacts and find people by their usernames. As a result, Telegram is like SMS and email combined and can take care of all your personal or business messaging needs[27]. The application at the date of the thesis is at the version v. 3.11 and its downloads number is between 100.000.000-500.000.000 units. It is quoted by 2.006.190 users with an average score of 4.2/5. The application has been chosen because it matches two useful main requirements in order to find some taintedness and checking it also in the code. Because the application is open source any output results given by the intent analysis can be checked in the source code and can be accessed from the git repository[18]. Moreover it is a well suited application for taintedness analysis due to its nature. In fact it could manages OS informations, telephone numbers, GPS positions and then there is an high probability that taintedness flows between components, for instance in order to share the location of the user, or the telephone number of a friend.

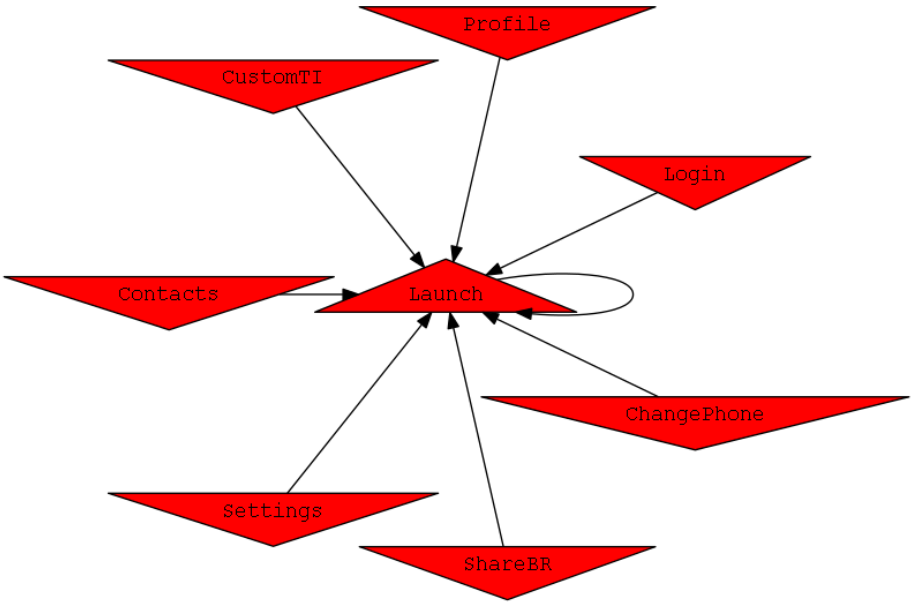


Figure 5.4.0.1: A simplified representation of Telegram intent taintedness flows

5.4.1 LoginActivity

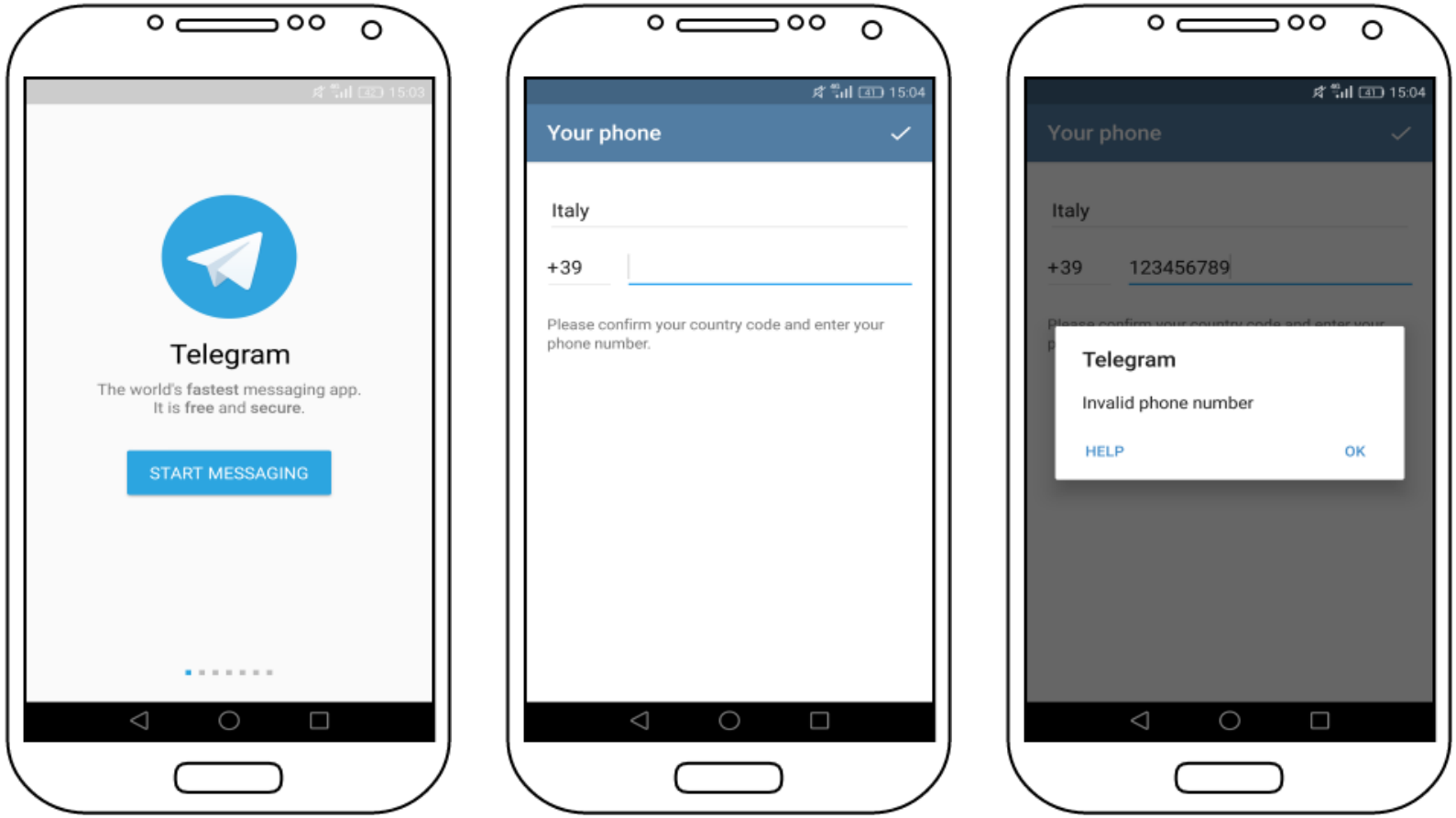
LoginActivity (represented in Figure 5.4.0.1 as "Login" labeled node) is showed to the user the first time when the application is started. It is the first step to complete to use the application, since it asks to the user the device's telephone number. If the telephone number is incorrect, meaning it does not satisfy length and format constraints, an error message its showed, alerting the user about the incorrectness and proposing him an help link. When the user taps on the help link the following happens:

- An implicit intent is created to satisfy the action: `"android.intent.action.SEND"`. This action alone is listened by many intent filters in the device. But the intent is addicted with the type, equals to `"message/rfc822"` that obliges the receiver to encapsulate messages, as in cases of emails.
- The application asks to the OS to collect those applications listening the requested actions, and the results are then shown.
- Once the selection is made, the application receives the intent and already fills the message to send. Selecting for instance Gmail[19], the full content of the email is showed and might be sent.

In the following piece of code the sources information are the devices sensitive data, like OS version and device manufacturer name.

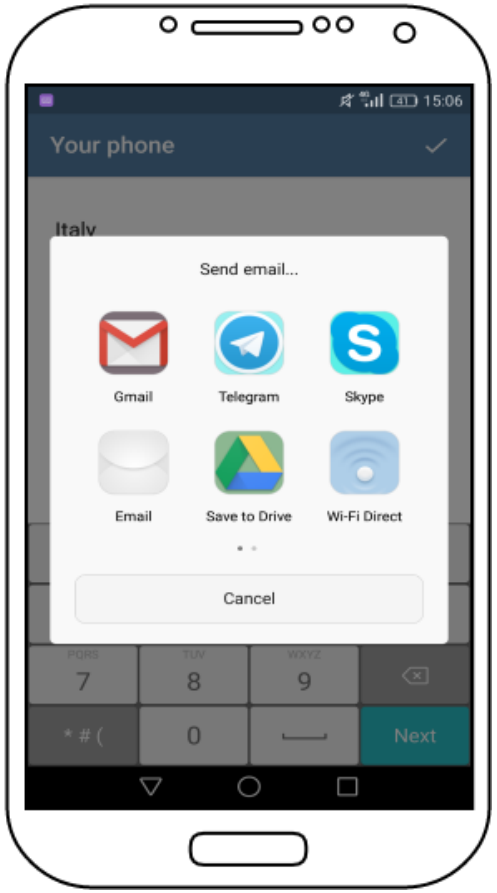
```
Intent mailer = new Intent(Intent.ACTION_SEND);
mailer.setType("message/rfc822");
mailer.putExtra(Intent.EXTRA_EMAIL, new String[]{"login@stel.com"});
mailer.putExtra(Intent.EXTRA_SUBJECT, "Invalid phone number: " +
    phoneNumber);
mailer.putExtra(Intent.EXTRA_TEXT, "I'm trying to use my mobile phone
    number: " + phoneNumber + "\nBut Telegram says it's invalid.
    Please help.\n\nApp version: " + version + "\nOS version: SDK " +
    Build.VERSION.SDK_INT + "\nDevice Name: " + Build.MANUFACTURER +
    Build.MODEL + "\nLocale: " + Locale.getDefault());
getParentActivity().startActivity(Intent.createChooser(mailer, "Send
    email..."));
```

Listing 5.24: the source code responsible of the intent creation and filling.

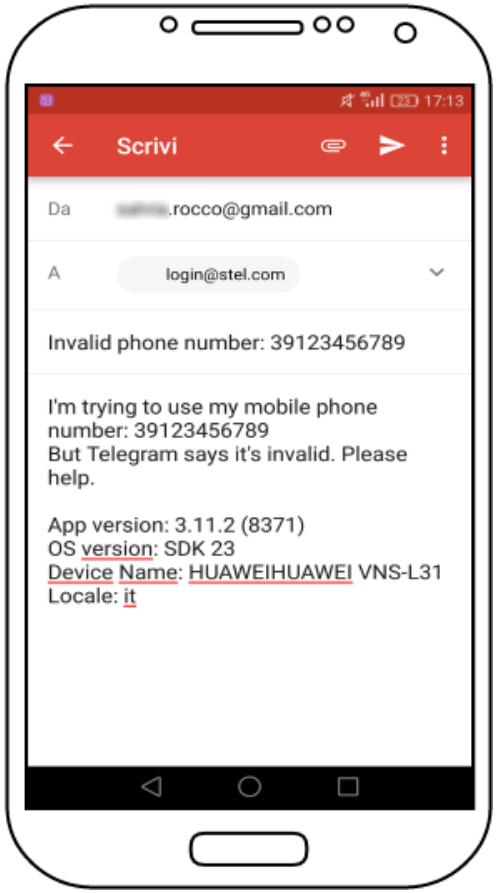


Screen-1 *Screen-2* *Screen-3*

Figure 5.4.1.1: In the Screen-1 is showed the first activity displayed to the user when it starts for the first time the application. The next step (Screen-2) asks to digit the telephone number. If the typed telephone number does not satisfy the application filter, telegram alerts the user with a message (Screen-3).



Screen-4



Screen-5

Figure 5.4.1.2: If the user clicks on the help link (Screen-3), the intent showed in Listing 5.24 is sended. The selection dialog is showed in Screen-4. Gmail application is clicked from the available list, and the result is that a pre-filled e-mail is displayed to the user. In particular the body tag is filled with sensitive data of the smartphone: OS version, device name and locale language (Screen-5).

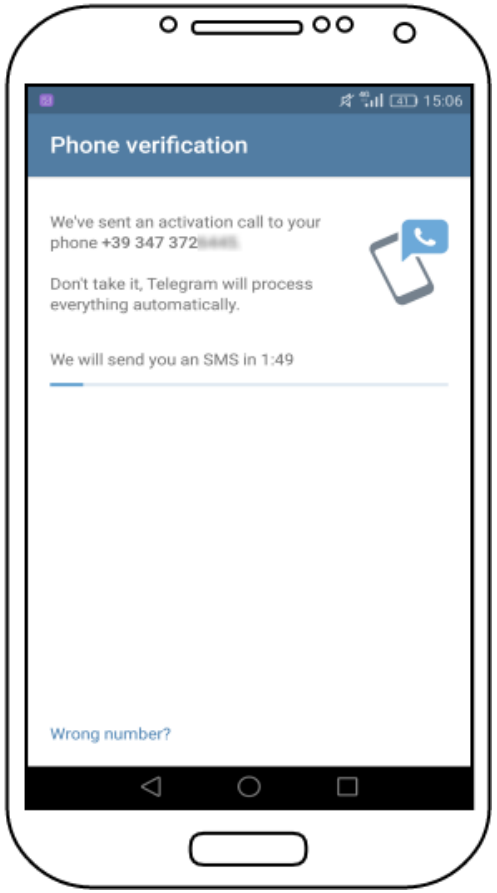
5.4.2 ChangePhoneActivity

Once the telephone number satisfies the format constraints, Telegram sends an activation phone call to the given telephone number, to verify the correspondence between the application activation request and the telephone number. If a mismatch occur the device does not receive any message, and the two minutes countdown completes. After it is shown another help link, asking about the correctness of the code. Once clicking the help link, again like in `LoginActivity`, an implicit intent requesting action `"android.intent.action.SEND"` is created, but now it is filled with another error message. Here there is the source code reporting the intent filling procedure and the error text:

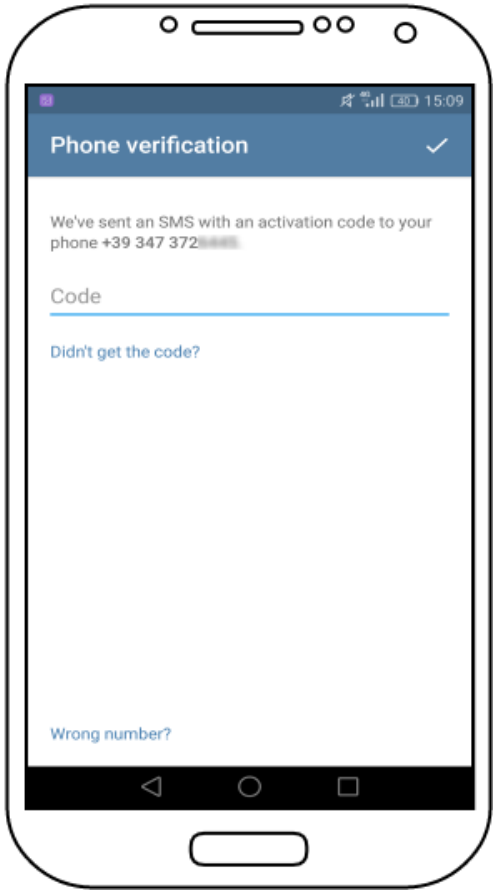
```
969 Intent mailer = new Intent(Intent.ACTION_SEND);
970 mailer.setType("message/rfc822");
971 mailer.putExtra(Intent.EXTRA_EMAIL, new String[]{"login@stel.com"});
972 mailer.putExtra(Intent.EXTRA_SUBJECT, "Invalid phone number: " +
    phoneNumber);
973 mailer.putExtra(Intent.EXTRA_TEXT, "I'm trying to use my mobile
    phone number: " + phoneNumber + "\nBut Telegram says it's invalid.
    Please help.\n\nApp version: " + version + "\nOS version: SDK " +
    Build.VERSION.SDK_INT + "\nDevice Name: " + Build.MANUFACTURER +
    Build.MODEL + "\nLocale: " + Locale.getDefault());
974 getParentActivity().startActivity(Intent.createChooser(mailer, "Send
    email..."));
```

Listing 5.25: the source code responsible of the intent creation and filling.

This sink could happens if the inserted number satisfy the requests of the constraints of the application filter, but it is not the telephone number holding the requests. The request is hold by Telegram server that replies with the security code, but the code will never reach the telephone since numbers are different. Once the countdown of 2 minutes finish, the application asks for the code but it shows an help link reporting the case where the code is never received.

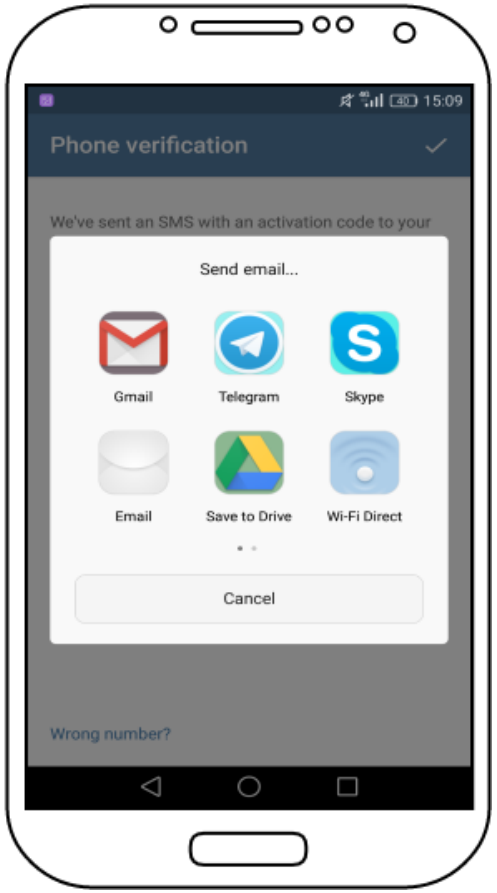


Screen-2.1

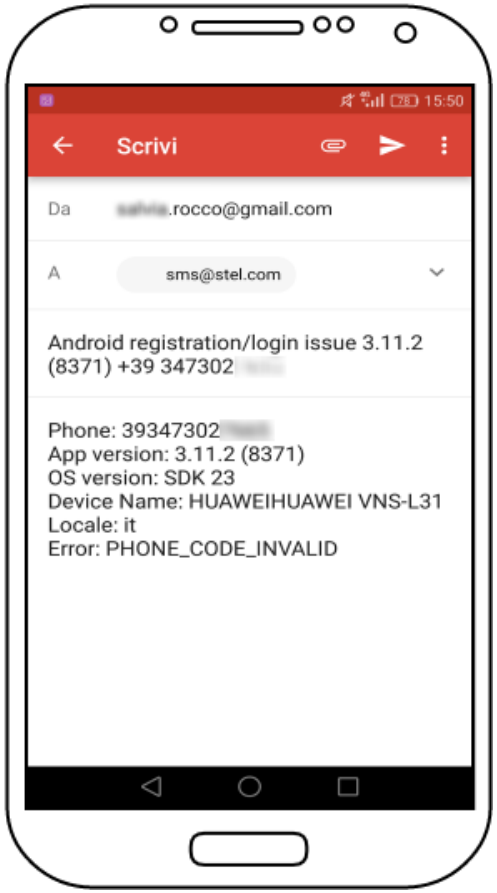


Screen-2.2

Figure 5.4.2.1: Once the telephone number satisfy the filters constraints, an activation calls is made towards the typed number. If a mismatch occurs, the call will be never receives by the device (Screen-2.1). Once the two minutes countdown finishes (Screen-2.2) an help link asks if the code was received. This link is clicked.



Screen-2.3



Screen-2.4

Figure 5.4.2.2: A selection dialog appears asking for the application holding the intent (Screen-2.3). If tapping on Gmail application, a pre-filled e-mail is displayed to the user. In particular the body tag is filled with sensitive data of the smartphone: OS version, device name and locale language.

Note that the dialog menu displayed both in Figure 5.4.1.2 and Figure 5.4.2.2 is deceiver. A more dangerous leakage could happen if just one intent filter satisfy the requested action. This is not the situation described since many applications could satisfy the action `SEND` due to its generality, but in case of exactly one match, the intent is automatically redirected to that component without any interaction by the user. This is extremely dangerous also for an expert user, since the choice is done automatically by OS.
Those are the main leaks discovered by our analysis. In the other situations reported in Figure 5.4.0.1 checking the taint flows in the source code it is not manually practicable since it is not easy to get how and with what, data filled into intents were tainted. They could be false positives, or maybe during the flow analysis they are tainted, but later overwritten again.

Chapter 6

Related Works

Other similar intent taint analyses have been built. Most of them handles only string constant like ours, but they focus the analysis on a limited set of components, for instance only activities, regardless of the different communication mechanisms involving other kinds of components. Few of them analyze also dynamic registration of `BroadcastReceiver` by analysing Java bytecode, others analysing the Dalvik instead of reverting it to Java bytecode or to an intermediate language.
In the following section we will collect the considered tools' main features.

- FlowDroid[16][17] performs a static taint analysis, by analysing directly the `.dex` file. It collects any possible sources and sinks using SuSi[1] that manages dangerous methods through machine learning. It is based on a mechanism that checks a list of properties of any method (name, return type, etc.)[1] in order to find sources and sinks. FlowDroid uses a call graph to ensure flow and context sensitivity. About inter-component communication it considers any method which sends intents as sinks, and calls to receiving methods as sources. It does not manage implicit intent communication, like implicit communication in activities through `startIntentForResults(Intent i)`. Parameters calls (modifiers) are collected only in case of string constant.

- Epicc[38] performs a sound static analysis of Android application. It is focused on inter-component communication, and most of the communication is based on intents. Analysis is performed over Java bytecode, and Dare[37] is used in order to revert Dalvik bytecode into Java bytecode. The string analysis computes the values of the arguments of the API calls `setAction()` and `addCategory()` of the class `Intent`. It performs a flow-sensitive analysis that can distinguish intent modifiers parameters at different program points. The analysis is able to connect components within single application and between different applications. The value of the intents at each exit point is precisely determined. For each application a list of entry points composed by all components listening for intent is built, and another of exit points containing those components that communicate by an outflow. All methods sending an intent are considered as exit points. Entry points are focused on the sinks of ICC, both in case

of dynamic registered broadcast receivers or intent filters found in the manifest.

- Iccta[34][23] is a static taint analyzer, that analyses privacy leaks as a path from source to a statement sending data outside the app or the device, called sink. Sources and sinks likes in FlowDroid are provided by SuSi[1]. Dalvik bytecode is first converted using Dexpler[28] that transform it to Jimle, an internal representation of Soot[26]. Soot was born as Java compiler but has evolved to perform also Java static analysis, in particular taint analysis, together with FlowDroid[16]. Soot analyses Java and Android applications, the second after a transformation in Jimple. Iccta modifies the Jimple representation in order to build the connection model between components and to enable dataflow analysis. Iccta leverages Epicc[38] to obtain the inter-component communication methods with parameters, then it applies FlowDroid to perform intra-component taint analysis. In addition to the analysis of Android manifest, in particular to intent filter tags, also dynamic registration of `BroadcastReceiver` is performed. The project contains a branch on DroidBench Git repository, that is full of test cases[22].

- Amandroid[40][2] is a static analysis framework for Android apps. It takes an APK archive as input, then Dex2PilarConverter uses Dex2Pilar module to decompile the `.dex` file into Pilar format, an intermediate internal format. Pilar is a highly flexible, typed, annotation based intermediate representation language. Amandroid introduces component-level models instead of FlowDroid's whole app-level model, but it is not able to analyze `ContentProvider` as well as some ICC methods such as `bindService` and `startActivityForResult`. Annotations on sources and sinks are done by an interface, and they are provided by the developer. For complicated String operations (e.g.concatenation in a while loop), if Amandroid cannot infer the exact string value, it reports it as any string (as in our analysis), ensuring the soundness of the analysis.

- TaintDroid[31] is a dynamic taint tracking system. It uses a modified Dalvik in order to track flows of tainted data. It assumes that downloads and third part applications are not trusted, and monitors in real time how those applications access and manipulate user's personal data. TaintDroid labels sensitive and private informations and applies labels during the propagation through variables, files, and interprocess messages. Start with the identification of taint sources and how those data impacts on other data. Its always performed at instruction level, and identified before they leave the system as taint sink.

Chapter 7

Conclusion

In this section we collect main considerations about the developed analysis and its limitations and future work.

- The analysis cannot yet detect dynamic `BroadcastReceiver` registration.
- Improving the string analysis. When an intent modifier is matched, the analysis inspects the input parameters of the method: in case of constant values, the match is perfect, since we know exactly the target key, if it is not the case, the approximation introduced consider all possibles reachable components in the application as possible destinations. The approximation introduced consider all the reachable classes, satisfying the manifest rules, and any reachable component. This may propagate a lot of false positive taint data. A more precise string analysis might yield better practical results.
- Field analysis. The backward and forward analyses performed by Julia, stops when a `getfield` or `putfield` bytecode instruction is reached. The last (or first) bytecode instruction reached by the analysis correspond reasonably to a `getfield` or to a `putfield`, based on which analysis between backwards and forwards is performed. If the intent is an intent from method or parameter, and it is assigned to a field in the constructor, it could be modified in the rest of the class. However it is not possible to relates the initialization with the modifier, since starting from the initialization point the modifiers cannot be reached using those two analyses (Listing 3.6). A solution could be to look for modifiers in the code and go back to the constructor (the same is done for intent extras), but produces a lot of overload, since it needs to collect all calls to intent modifiers and for each one go back until an initialization method is found.
- Sometimes both the creation and backward analyses are not precise, and the returned set of information regarding a call instruction could be huge. The precision of the analysis decreases as the analysis becomes wide. Since intents are usually created on the fly each time they are needed, filled with the desired information and sent to a component, the creation or backward analysis usually does not produce an empty result set if limited to the same method, or classes. This means that if extending the analysis to the entire

application does not produce precise results, the risk is that in order to find the starting point of an intent, could be included a lot of false positive communications. Our heuristic assumption of course is not sound, since it may happen that potential creation points are discarded, but it can be easily applied to problems where the result set is huge and reduces time and complexity.

- The previous point clearly introduces consideration about the analysed test suite. If the scope of a static analyser is to resolve and find issues of a real application distributed on the market it is interesting to test it over all available android components. Maybe it is more interesting to tests all traditional communication patterns based on intents (as in iccta[34]) for any component instead of focus on extremely rare situations for just few of them.

- In the developed analysis any intent has no relation with the component class it belongs. An intent flowing into a Service, has the same properties as an intent flowing into an Activity. This solution was taken in order to simplify the model, and there exist few cases where it is necessary to be discriminatory and those are managed separately. In fact the communication between activities through `startActivityForResult(Intent i, int n)` as depicted in result section, allows the receiver of the intent to get back to the sender a result, through an intent. The backward communication has dedicated rules, in fact the receiver does not have to set the address of sender or the action as usual, but it simply calls the `onResult(Intent i, int n)` method. This particular communication is managed as a special case. Software engineering suggests to connect components and intent, in order to manage those situations that our analysis cannot cover.

- The final goal of the project is to extend the analysis to inter-application communication, since the goal is to detects if communication between applications could results in tainted flow.

Acknowledgments

First and foremost, I would like to thank my thesis rapporteur Prof. Agostino Cortesi who proposed to me the internship in Julia S.R.L and supported me in the draft of this thesis.
I would like to thank all the members of Julia S.R.L, especially Pietro Ferrara and Fausto Spoto that anytime and for any request were available to help and support me from the first day I started the internship and let me discover what a scientist is. A special thanks to my family, to my girlfriend and to all of my friends for supporting me during the development of this work.

Bibliography

[1] A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. https://www.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_EC-Spride/Talks/SuSi_Talk_NDSS_2014.pdf.

[2] Amandroid Web Site. http://amandroid.sireum.org/docs/.

[3] Android ContentProvider. https://developer.android.com/guide/topics/providers/content-provider-basics.html.

[4] Android Services. https://developer.android.com/guide/components/services.html.

[5] AndroidStudio. https://developer.android.com/studio/index.html.

[6] APKTool. https://ibotpeaches.github.io/Apktool/.

[7] class Activity. https://developer.android.com/reference/android/app/Activity.html.

[8] class Activity method getIntent(). https://developer.android.com/reference/android/app/Activity.html#getIntent().

[9] class BroadcastReceiver. https://developer.android.com/reference/android/content/BroadcastReceiver.html.

[10] class SmsManager. https://developer.android.com/reference/android/telephony/SmsManager.html.

[11] class SmsManager method sendTextMessage. https://developer.android.com/reference/android/telephony/SmsManager.html#sendTextMessage(java.lang.String,java.lang.String,java.lang.String,android.app.PendingIntent,android.app.PendingIntent).

[12] Device IMEI code. https://developer.android.com/reference/android/telephony/TelephonyManager.html#getDeviceId().

[13] Dex2Jar. https://sourceforge.net/projects/dex2jar/.

[14] DroidBench Git Repository. https://github.com/secure-software-engineering/DroidBench.

[15] DroidBench Website. https://blogs.uni-paderborn.de/sse/tools/droidbench/.

[16] FlowDroid. http://www.abartel.net/static/p/pldi2014-FlowDroid.pdf.

[17] FlowDroid Web Site. https://blogs.uni-paderborn.de/sse/tools/flowdroid/.

[18] GitTelegram. https://github.com/DrKLO/Telegram.

[19] Gmail on Google Play. https://play.google.com/store/apps/details?id=com.google.android.gm&hl=it.

[20] Gradle. https://developer.android.com/studio/index.html.

[21] Graphviz Dot Representation. http://www.graphviz.org/.

[22] Iccta DroidBench Branch. https://github.com/secure-software-engineering/DroidBench/tree/iccta.

[23] Iccta Git Repository. https://github.com/lilicoding/soot-infoflow-android-iccta.

[24] Julia S.R.L. http://www.juliasoft.com/ita.

[25] JuliaDexConverter. https://bitbucket.org/tregua87/juliadexconverter.

[26] Soot Git Repository. https://sable.github.io/soot/.

[27] Telegram. https://telegram.org/faq.

[28] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, SOAP 2012, Beijing, China, June 14, 2012*, 2012.

[29] Agostino Cortesi. Partial orders, lattices, continuous functions, fixpoints, 2015.

[30] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[31] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones. *Commun. ACM*, 2014.

[32] Michael D. Ernst, Alberto Lovato, Damiano Macedonio, Ciprian Spiridon, and Fausto Spoto. Boolean formulas for the static identification of injection attacks in java. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, volume 9450, pages 130–145. Springer, 2015.

[33] Samir Genaim and Fausto Spoto. Information Flow Analysis for Java Bytecode. In *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*, volume 3385, pages 346–362. Springer, 2005.

[34] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, 2015.

[35] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[36] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[37] Damien Octeau, Somesh Jha, and Patrick McDaniel. Retargeting android applications to java bytecode. In *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. ACM, 2012.

[38] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. USENIX, 2013.

[39] Fausto Spoto. The Julia Static Analyzer for Java. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, pages 39–57, 2016.

[40] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, 2014.