SecDroidBug: A Debugger for Detecting Security Violations in Android Apps

Submitted in partial fulfillment of the requirements of the degree of

Master of Technology (M.Tech)

by

Gopesh Singh Yadav

***Roll no.* 163050045**

Supervisor:

Prof. G. Sivakumar

Co-Supervisor:

Prof. R. K. Shyamasundar

Department of Computer Science & Engineering

Indian Institute of Technology Bombay

2018

Abstract

We have built SecDroidBug, an Android debugger that simplifies debugging and testing of Android Apps for security properties. Though there are several tools that analyze and debug the code of an App, there is no tool available for debugging an App for its security properties. SecDroidBug interacts with an App running on an Android device and monitors its behavior and reports when there is any sensitive data leakage by the App to outside world through network interfaces. SecDroidBug allows setting of breakpoints where it will stop the running App. It monitors running App from first breakpoint till last breakpoint. SecDroidBug can watch variables supplied by the user. On each breakpoint it will give the list of variables touched by each of these watch variables. The debugger considers user supplied watch variables and other variables that store sensitive information e.g, location, contacts, etc as sensitive variables. Between the breakpoints, if any variable touches a sensitive variable, debugger includes it in the set of sensitive variables. SecDroidBug maintains and gives a list of sensitive variables line by line between the breakpoints. If any of these sensitive variables are sent to outside world the data leakage is reported. SecDroidBug requires Java source code of the Android App. Therefore it can be used only with Apps whose source code is available. It is extremely useful for App developers to debug their Apps for security properties.

Contents

List of Figures

Chapter 1

Introduction

Smart-phones are important part of life in today's world. Due to their portability and increasing functionality they are used today for most of the works e.g., online shopping, entertainment etc. Applications giving these functionalities requires some sensitive information from user e.g., passwords etc and from Android device e.g., location, contacts etc. Because of this, all smart-phones attract people who want to steal sensitive informations from smart-phones. Android being one of the popular mobile platforms, is on the target too. While most of the Apps available on Google Play Store and some other App stores are secure and do not share sensitive information of user to unintended recipients or if they use it they inform user about their policy, but there are still many Apps that steal user's sensitive information e.g., location, contacts, images stored on phones. Although Android platform too provides security and tries to stop stealing of information as much as it can, but it can not stop all data leakages from Android App level. Whenever user gives permission to Android App to access some sensitive resource it has no view of where the App is sending that information. Android has no fine grained access control over flow of sensitive information.

In next section we explain various static and dynamic security analysis tools available for Android and their literature. Then we explain Java Platform Debug Architecture (JPDA) [9] supported by Oracle that is used to make custom debuggers for virtual machines which support Java Debug Wire Protocol (JDWP), without working from scratch. We mainly concentrate on Java Debug Interface (JDI) part of JPDA, which provides enough functionality to write debugger without caring much about other parts in JPDA.

Then we explain JavaParser library [16] which can be used to parse Java source code to get elements of interest from it e.g., method calls, variables etc.

We developed SecDroidBug, a debugger using both JDI and JavaParser library to debug security inside Android Apps. We explain it in detail in Chapter 5. This tool can be used in testing of Apps, provided their source code is available. If we want to debug any class using our debugger, the full source code of the file containing it is necessary. This tool can also be used to test open source Apps and some third party or open source modules whose source code is available.

SecDroidBug allows setting up of multiple breakpoints on the source code of an App. It allows user to give list of sensitive variables to be watched. On each breakpoint SecDroid-Bug, our debugger will print what variables these watch variables has touched. If any variable touches any sensitive source API call return data e.g., location, contacts, gallery etc, it becomes sensitive. Our debugger treats watch variables which are supplied by the user and variables storing information from sensitive sources as sensitive. Any variable that touches these sensitive variables also becomes sensitive. The debugger prints current sensitive variables starting from first breakpoint line by line. If any sensitive variable is sent outside through sensitive sink then direct data leakage is reported.

We developed SecDroidBug using two Approaches, first Approach with only JDI and current Approach with both JDI and JavaParser library. We explain these two approaches in detail in next chapters. Then we show how to use our debugger and after that we show results when we used our debugger with sample App and some Apps from Android Play Store and F-Droid Android App repository.

After trying to implement debugger with only JDI library there were some limitations in the Approach. To find variables in the arguments of methods it required `MethodEntry` events to cover each and every method that were called inside the method under debugging. This was causing performance issues. Also there is no support from JDI library to watch local variables for modifications and accesses. So we used JavaParser library for above two tasks. JavaParser is a parser to parse Java source code and identify elements of interest from it e.g., methods, variables etc.

Organization of this report is as follows. In Chapter 2 we talk about related work. In Chapter 3 we discuss Features of SecDroidBug. In Chapter 4 and 5 we discuss used

Java libraries in our work in detail and implementation of SecDroidBug respectively. In Chapter 6 we explain usage of our debugger. Chapter 7 discusses other tools used with SecDroidBug for debugging Apps. Finally we discuss experiments, conclusion and future work in subsequent chapters.

Chapter 2

Background and Related Work

Android security testing methods can be partitioned into two categories, dynamic testing and static testing. We shall discuss five different works done for testing security in Android using dynamic testing and static testing.

Flowmine

This paper talks about a tool called Flowmine [15] which uses Flowdroid [6] a static taint analysis tool to decide whether an Android App is malicious or benign based on the behavior similarity of that App with a large number of known benign and malicious Apps. The behavior similarity is examined after finding common paths from sensitive data sources to sensitive data sinks after applying taint analysis using Flowdroid. Sources are defined as calls into resource methods returning non constant data and sinks are calls into resource methods accepting non-constant data. These sources and sinks are found using a machine learning classifier that classifies all permission based Android API methods into sources, sinks, and neither. For Flowmine these were found after taking union of all sources and sinks from Android API version 7 to 22. Due to large number sources and sinks they were further categorized into 12 sources and 14 sinks. Using Flowdroid, flows between these sources and sinks were found in 2800 benign Apps and 15000 malicious Apps. On the basis of number of occurrences of source-sink pair in benign Apps and malicious Apps, benignity rank and malignity rank of that pair is decided. If that pair of source and sink comes mostly in malicious Apps, then malignity rank of that pair is higher.

If it comes mostly in benign Apps then its Benignity Rank will be higher. FlowMine finds the percentage use of each source-sink pair in both benign samples and malicious samples and then gives them weight as (% use in benign Apps - % use in malicious Apps). These weights are then stored in weight lookup table. To decide an App as malicious or benign Flowmine first finds source-sink pairs in it then it sums up weights of these source sink pairs using weight lookup table. If summed weight comes as positive Flowdrive declares it as Benign otherwise it declares it as malicious. FlowMine classifies Apps with an accuracy of 96.5%. It can correctly classify 96% of all benign Apps and 97.2% of malware samples.
Limitations: It does not check code wrApped in native code and its accuracy depends on accuracy of Susi [5]. So if Susi misses to detect some sensitive API calls as sources or sinks then Flowmine will ignore it.
Method: static analysis, source-sink weights

DroidTest

This paper [14] talks about a black-box testing method to detect leakage of data in Android Apps based on correlated test cases generated from some test case. Test case is defined as sequence of clicks on screen that generates network flow.

The property of these correlated test cases is that they will trigger the same result in the system if there is no leakage of private data. The data leakage is reported in App if these correlated test cases give different output network traffic on running. These correlated test cases includes sensitive information, e.g., current location, personal pictures. If we observe different outputs generated by a set of correlated test cases, then it is reported that it is due to the difference in the sensitive inputs. This Approach does not require App's source code. This method is Applied on DroidTest [14] a modified Android operating system. DroidTest has three components test case generator, test case executor and kernel log collector. The test case generator converts each existing test case into a pair of correlated test cases that only differ in the private data part. It changes APIs calling sensitive information e.g., location data, contact lists etc with custom source code to generate random values. So that different invocation of these methods will return different values. This setting creates correlated test cases with variations in the sensitive data. The test

case executor runs the Application manually on these two correlated test cases and reports when private information is leaked through network interfaces. The kernel log collector monitors the network interface and collects the outgoing packet data. This is done by intercepting the system calls.
Limitations: It may miss some true positives as it may not cover all cases. There may be cases where program output without changing the sensitive input.

Privacy Leak Detection via Dynamic Taint Analysis

In [11] the authors present a system which uses dynamic taint analysis to keep track on the flow of sensitive information inside an Android App. The system is PC-based Java Application to instrument Apps with taint propagation functionality which is said to be able to run as Android App too after recompilation. The system decompiles an Android App to be tested and inserts taint tracking functionality into the code, and recompiles it into a behaviorally-equivalent App. When this new instrumented App is run, it produces alerts whenever sensitive information leaves the device, e.g., over the Internet or through SMS. When the App is run on the Dalvik VM of the device taints are propagated. First, the APK file from the App to be tested is pulled. Then using APKTool [17] decode option the APK is then disassembled into its assembly Smali files. After this the PC based Java Application parses the assembly code, and inserts the Appropriate taint propagation calls into the taint propagation library corresponding to each instruction. The resulting instrumented assembly code is assembled together with the taint propagation library. Then assembly files are build to make new APK and finally this APK file is signed and then run on the Android device. The taint propagation library defines the sensitive fields or taint sources and keeps track of taint information for live objects in the program and alerts when tainted objects go out of the device.

DroidMate

This paper talks about DroidMate [8] an automated GUI execution generator for Android Apps. Dynamic analysis of Android Apps requires executions that should cover program behavior thoroughly in all possible combinations. Doing this in manual way is very time consuming and requires human interaction. DroidMate as an automated tool avoids this problem and repeatedly reads at runtime the device GUI and monitored calls to Android APIs methods and makes a decision what next GUI action to should be executed, based on that data and provided exploration strategy until some termination criterion is met. DroidMate is fully automatic after it has been set up. The setup requires exploration strategy, termination criterion and a set of monitored methods. It was tested on 126 Apps being in the top 5 in all Google Play categories and it ran successfully on 123 of them. It runs without root or OS modifications. It can be modified as its source code is publicly available in case of newer Android versions in future. DroidMate takes as input directory containing a set of .apk files to be tested. Each of these files then undergoes a slight dalvik bytecode modification to enable monitoring of Android SDK APIs methods. Then DroidMate installs each App on an Android device and then launches main activity. DroidMate then operates in a loop, based on termination condition.

Secflowdroid

This work talks about an integrated tool called SecFlowDroid [4] which uses both static and dynamic analysis to check data leakage in Android Apps. It uses modified Soot [7] and modified APIMonitor [18] inside it. It generates dynamic traces using APIMonitor . Then using those traces it converts App's method code into Jimple code which is 3-address code, using a tool called Soot. It then Applies Reader-Writers Flow Model which covers direct and indirect flows on it and analyses the flow of sensitive data between calls. APIMonitor dynamically analyses the App and outputs the sensitive APIs that are invoked during execution. APIMonitor has configuration file which contain the API calls that have to logged, based on that it inserts monitor code for only those APIs. It extracts the APK file into Smali format and insert the monitor code for API calls and signs the APK file. This repackaged file when run on an emulator and interact with the user gives the desired logs. Logs contains name of class and method in App where sensitive APIs are called, and the name of those sensitive APIs and methods.This log contains order of execution of protected APIs. Now to apply RWFM model APK is converted into Jimple form. Trace the data flow of each statement of the converted program using RWFM model and report misuse if any hAppen.

Chapter 3

Features in SecDroidBug for Debugging Android Apps

In this chapter, we discuss debugging features for debugging security violations for Android Apps incorporated in SecDroidBug. They are as follows:-

Line by line keep track of sensitive variables

In SecDroidBug sensitive variables are defined as variables that are either provided by the user for debugging (also called as watch variables) or variables that store sensitive information e.g, location, contacts etc (also called as sensitive sources) during the debugging process. If any variables touches these sensitive variables, it also becomes sensitive. Based on the data stored by the variables in the App code, line by line number of current sensitive variables keeps changing. SecDroidBug keeps track of current sensitive variables and prints them line by line while monitoring the running Android App.

Watch user provided variables

These variables are provided by the user. They are considered as sensitive. They are monitored throughout the debugging process. For each of these watch variables, SecDroidBug keeps track of variables accessed by them from starting of the debugging process. On each breakpoint SecDroidBug prints this information for all watch variables. If A is a watch variable and it accesses variable B which had accessed variable C before, then both B and C are included in the list of variables accessed by A.

Detect data leakage

SecDroidBug keeps track of API calls which send data to outside world, also called as sensitive sinks. If it finds that a sensitive variable is being sent to outside world it reports data leakage. These sensitive variables may be watch variables or variables storing information from sensitive sources or from watch variables.

The working of SecDroidBug is shown in the Figure 3.1. It takes list of watch variables and breakpoints from the user and monitors running Android App. It runs App step by step and prints list of current sensitive variables. On breakpoint variables accessed by user provide breakpoints are printed. During the debugging process if any sensitive variable is sent to outside world the data leakage is reported. In the next section we discuss list of debugging command supported by SecDroidBug to debug security in Android App.

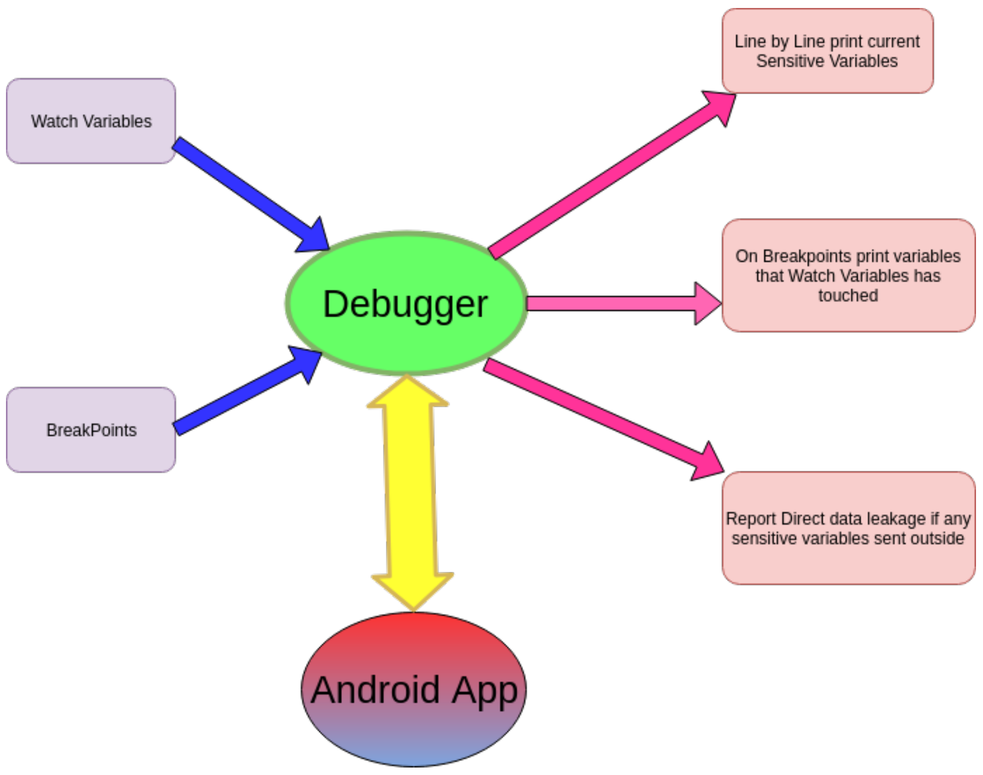

Watch Variables
BreakPoints
Debugger
Android App
Line by Line print current Sensitive Variables
On Breakpoints print variables that Watch Variables has touched
Report Direct data leakage if any sensitive variables sent outside

Figure 3.1: Working of SecDroidBug

3.1 Debugging Commands supported by SecDroid-Bug

Setting up of breakpoints

User can select any number of breakpoints. They should be inside single method. It is also supported to set breakpoint and debug inside anonymous, inner classes. To set a breakpoint user need to click on line number. On clicking the line goes red. If again clicked on same line then breakpoint is unset. Breakpoints are not allowed on comments, blank lines and on method and class declaration first lines.

Giving list of variables to be watched

There is one textbox to get name of variables to be monitored. Their names should be space separated. The variable names should be fully correct.

Start Debugging

There is red button with text ‘Start Debugging’ which when clicked starts the debugger. Before clicking on it the breakpoints must be set otherwise the debugger may fail. After the breakpoint is hit the App is stopped until user resumes it.

Resume debugging after hitting breakpoint

Except last and first breakpoints whenever debugger hits any breakpoint it pauses the App. On such breakpoint hits there is resume button which goes green. On clicking it App resumes and starts executing from the line of breakpoint.

In the next chapter we discuss two Java libraries used in developing SecDroidBug, in detail. In chapter 5 we describe implementation of SecDroidBug.

Chapter 4

Design Choices for Building SecDroidBug

We have used Java Platform Debugger Architecture (JPDA) [9] and JavaParser Library [16] in our work. While using JPDA, we have ability to dynamically monitor any App, JavaParser allows us to understand elements from the source code. JavaParser alone could not be used in dynamic analysis as it is static code analysis tool, and JPDA alone could not be used in our work because it does not have functionality to track flow of data between variables. So we needed features that could be supported only if we used combination of both approaches. We now discuss JPDA and JavaParser in detail next.

4.1 Java Platform Debugger Architecture

JPDA [9] stands for Java Platform Debugger Architecture. It's a multi-layered debugging architecture which enables developers to write custom debuggers for debugging programs. JPDA [9] is a collection of APIs to help writing custom debuggers. It has three layers JVM TI [9], JDWP [9] and JDI [9] which are discussed further in detail. JDI is highest of all layers and easiest to write custom debugger. Oracle has also provided a reference implementation of all three layers. So just by modifying JDI layer it is possible to write most of the debuggers. These debuggers will also work with other virtual machines that implement JDWP for e.g., Android Dalvik virtual machine. Many debuggers e.g., Netbeans, Eclipse, InteliJ etc are implemented using JDI. The overall architecture of JPDA is shown in Figure 3.1. The three layers discussed are:-

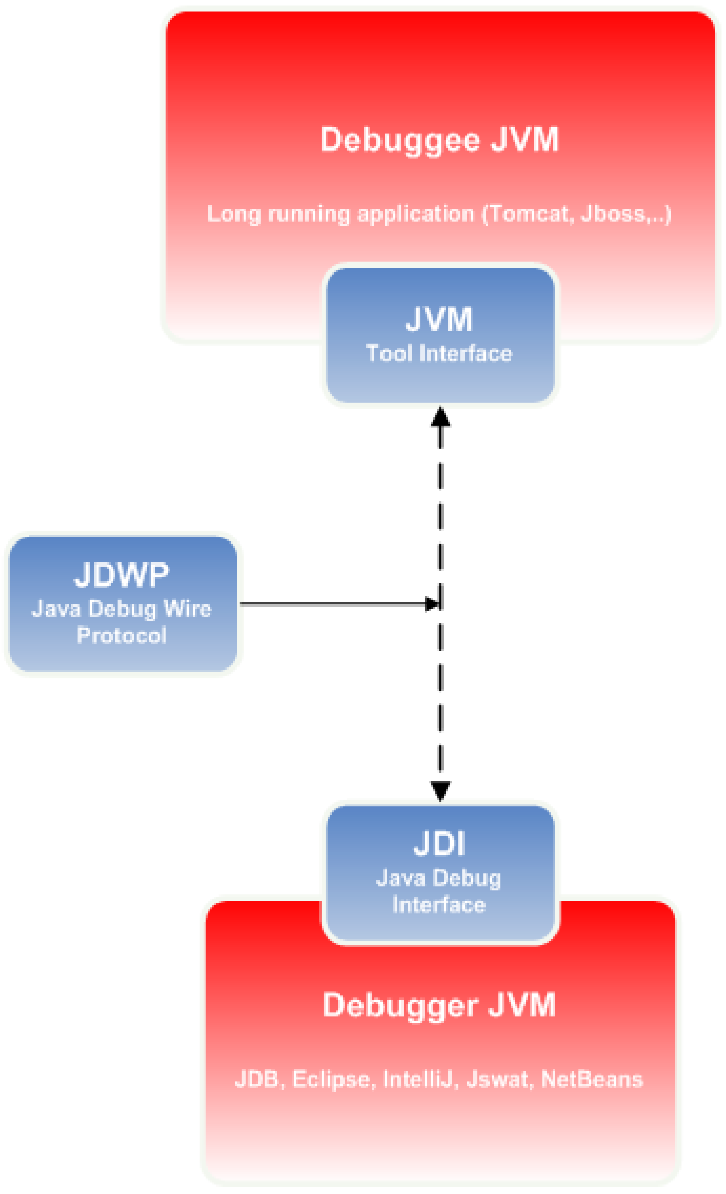


Figure 4.1: JPDA architecture [3]

Java Virtual Machine Tools Interface (JVM TI)

JVMTI [9] is a native programming interface implemented by VM. It defines the services a VM must provide for debugging, for example current stack frame, and actions to make breakpoint, step points and generate events according to them.

Java Debug Wire Protocol (JDWP)

JDWP [9] Defines the format of information and requests transferred between the debuggee process and the debugger front-end. The specification of the protocol allows the debuggee and debugger front-end to run under separate VM implementations and/or on separate platforms. It also allows the front-end to be written in a language other than Java, or the debuggee to be non-native (e.g., Java).

Information and requests are roughly at the level of the Java Virtual Machine Debug Interface (JVM TI), but will include additional information and requests necessitated by bandwidth issues, examples include information filtering and batching.

Java Debug Interface (JDI)

JDI [9] is a high-level Java interface implemented as front end. Oracle recommends this to use on JDI layer for all debugger development without the need to modify at JVM TI or JDWP layers. It uses JDWP for communication with the debuggee JVM. In next section we describe it in detail.

4.1.1 Java Debug Interface

The Java Debug Interface (JDI) [9] is a high level Java API providing the debuggers ability to debug running process on some other virtual machine. The JDI provides control over a virtual machine's execution. JDI implements JDWP protocol to connect with remote VM which is implementing JVM TI. It provides ability to suspend and resume threads, and to set breakpoints, watchpoints, notification of exceptions, class loading, thread creation and the ability to inspect a suspended thread's state, local variables and stack-traces. It provides these functionalities with following requests to event request manager of virtual machine manager which represents target virtual machine:-

Supported event requests under JDI

BreakPoint Requests This request enables debugger to make running program stop on specific line in the code. This may be used to inspect the state of the program at that exact line. This request usually follows `ClassPrepared` Request which gives all possible line locations where breakpoints can be set. When ever line corresponding to breakpoint requests are to be executed the break point events are generated.

The events generated for this request gives information about event location e.g., line number, method and class which contains that location, thread reference, stack-frames, local variables from stack-frame, getting and setting their values

Field Modification requests on class variables This request is one of most useful requests which is used to pause or alert whenever a variable is modified in execution. This events allows us to get information of that variable. This event can be applied only on class/instance variables. At the time of creating this request we have to specify variables/fields to be monitored. This request usually follow `ClassPrepared` request. The events generated for this request gives information about event location, thread reference, Object reference, Field modified: current values and value to be.

Field access requests on class variables This request is generated whenever a specified variable is read, it includes whenever this variable's value is copied into some other variable or it is passed as an argument to some function. This event also can be applied only on class/instance variables and this request also follow `ClassPrepared` request. Variables to be monitored are specified at the time of request creation. The events generated for this request gives information about event location, thread reference, Object reference, stack-frame, Field accessed: current values

Step requests This request is used to run program step by step. It takes argument as number of steps to be jumped to stop on next step. The program stops on line which is specified number of steps away from the line where this request was created. This request runs in three modes:-

1. STEP_OVER: Go to next line without going inside method which is called on next line.
2. STEP_OUT: Step out and stop on line which follows line where the method was called.

3. STEP_IN: Go inside a method if it is called on next line

The events generated for this request gives information of event location, thread reference, stack-frames, getting local variables from stack-frame, getting and setting their values.

MethodEntry and MethodExit requests This request is used to monitor the calls to functions of specified class. Whenever control enter in any method, `MethodEntry` event is generated and when control comes out, `MethodExit` event is generated. The events generated for this request gives information of methods that are called and their classes, location of those methods, local variables, class, arguments, method type information

Requests for informing thread creation and deletion These requests are used to monitor thread creations and thread deletions. These gives information about thread which caused this to happen.

ClassPrepare request This request is used to capture when is class loaded into the virtual machine. This is generated for each class whenever they are loaded into virtual machine. The events generated for this request gives information of thread reference, class reference type, class type information, all fields, methods of class and all executable line locations in the class.

The above all request allows them to be restricted to only some classes, class patterns or threads. This all information can be supplied at the time of creation of these requests.

Supported connectors between debuggee VM and debugger

In JDI three different type of connectors are used to make connection between debuggee process and debugger. There types are:-

1. Listening connector: Used to listen for a debuggee process to connect to it to start debugging

2. Launching connector: This launches a process to be debugged and then connects debugger to it.

3. Attaching connector: This connects debugger to already running process

4.2 JavaParser Library

JavaParser [16] is an open-source Java parser library for parsing Java source codes files to get its subelements. It can be used to analyze, transform and generate Java code. It is based on JavaCC, which is parser generator and generates a parser from formal grammar. JavaParser can also be used with JavaSymbolSolver [16] sublibrary to resolve type of variables and method calls. JavaParser converts java source files into abstract syntax trees. Root will represent whole source code file. The children of root will be all top elements in the file, for example class/interface declarations, import statements, package definition etc. Each of these child nodes will be further divided into nodes representing their sub elements, for example a node representing a class will be divided into nodes representing fields, method declarations etc. These nodes will be further divided until the node becomes atomic, and can not be divided further. These will be called leaf nodes in the AST.

By using JavaParser it is possible to extract elements of interest from a Java source code file, for example it is possible to find all classes, method calls, method definitions, if-else statements, for loops etc by parsing source code files. Figure 4.2 shows one example after parsing source code. With JavaParser it is also possible to get more information about nodes of parsed ASTs. For e.g., from a node representing a class we can find its beginning and ending location, fields, methods etc. We can search for sub-elements for a specific node.

JavaParser library also allows to modify the structure of the source code. It is possible to move nodes around the parsed tree. The resultant tree can be used to generate different source code file and thus allowing code generation.

We have now discussed two Java libraries that we will be using to write SecDroidBug. We have used two approaches for implementation. We discuss them in next chapter.

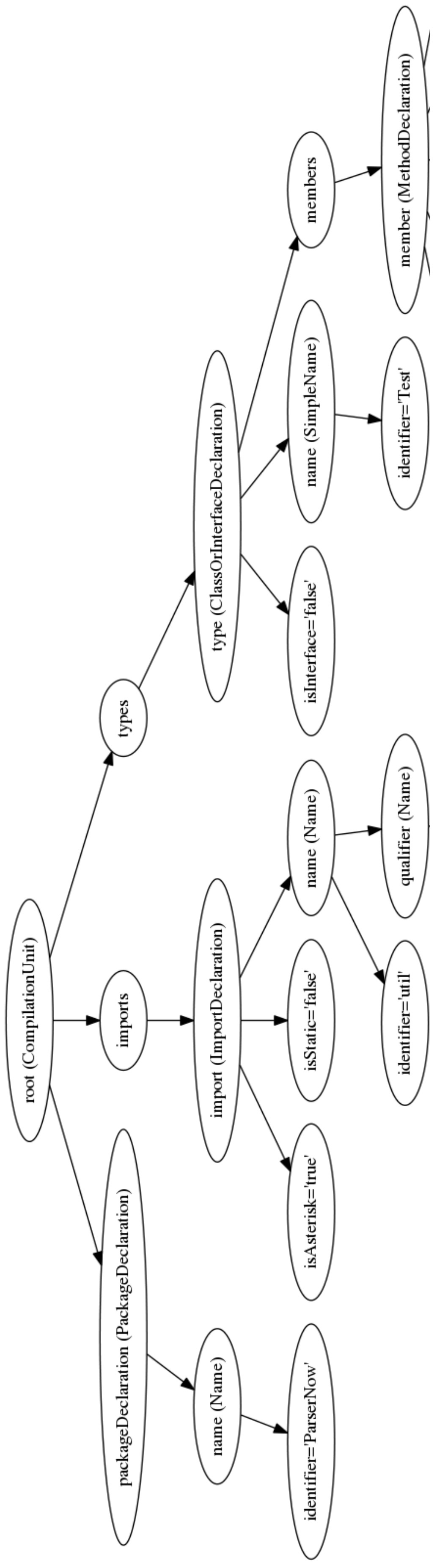


Figure 4.2: Example of parsed Java source code using JavaParser

Chapter 5

Implementation of SecDroidBug

We have developed SecDroidBug with two approaches: i) with only JDI [9] library, and ii) with both JDI and JavaParser [16] library. Our first approach did not perform better in terms of speed, so we chose to develop the second approach, which is our current approach. In both the approaches we used sensitive source and sink API list from APIMonitor [18]. We also modified it to add/delete some API calls.

In both the approaches in JDI part we reuse the basic example provided by Oracle [13] to get started with JPDA [9], to some extent in our code. We used Attaching connector (see section 4.1.1) in JDI Parts to connect our debugger with running Android App. Also in both approaches watch variables supplied by user as input are treated as sensitive and added to sensitive variables list by default. Sensitive variable list contains watch variables and variables touching sensitive sources e.g., location, contacts etc. In both approaches debugger GUI is made using Java Swing framework, which shows source code of the App to user. GUI allows user to select breakpoints and contains buttons to start and resume debugging.

5.1 Using JDI Library

To implement our debugger it is important to understand the flow of data from the source code. So from each Java expression we need to find which variable is getting modified and which variable is getting accessed. To define these we shall discuss our approach next. After discussing our approach we shall see in section 5.1.2 how we can use found modified and accessed variables to track sensitive variables and to check data leakage.

5.1.1 Find which variable is accessing which variables on current line

Since JDI provides different support for class and local variables we shall see how we can find modified and accessed class and local variables, separately using JDI .

Find modified variables

- **Class Variables**

  Using `ModificationWatchPoint` Events in JDI [9] it is possible to find which class variable was accessed. For all class variables debugger registers request to App VM to be notified on each such events.

- **Local Variables**

  Debugger maintains a table which contains values of local variables. After each line execution this table is updated as per their current values. To find which variable's value got modified we compare its current value with the table entry value, which contains its value on last line, so if they do not match then we can say that that local variable was modified on current line.

Find accessed variables

- **Class Variables**

  Using `AccessWatchPoint` Events in debugger, which are generated on class variable accesses, it was possible to find all class variable accessed on current line.

- **Local Variables**

  Accessed local variable on current line was found by comparing value of modified variable with value of all other variables inside table which contain values of all variables till last line.

This was very basic approach which does not cover many cases, so it required more work to do to find all accessed local variables.

5.1.2 Implementation

For watching modification and access of local variable one table was maintained to keep track of local variable values after execution of line.

Line by line keep track of sensitive variables

`MethodEntry` Events were used to record sensitive source API calls. For each sensitive source call, the variable that is storing its return value was found and was added in to list containing sensitive variables.

If a variable touches any sensitive variables it was added into sensitive variables list.

For each variable keep record of all variables it has touched

One table was maintained to keep track of variables that a variable has touched beginning with first breakpoint. If any variable is getting modified then add variables that it is accessing and also those accessed variables' table entries to table entry corresponding to this modified variable.

Detect Direct data leakage

Using `MethodEntry` Events sensitive sink API calls were monitored. And it was checked whether values passed to them contain any value from sensitive variables. If yes, then direct data leakage was reported.

5.1.3 Limitations

- For expressions like

  ```
  variableD=variableZ+variableA.funcionCall(variableB,variableC.....)
  ```

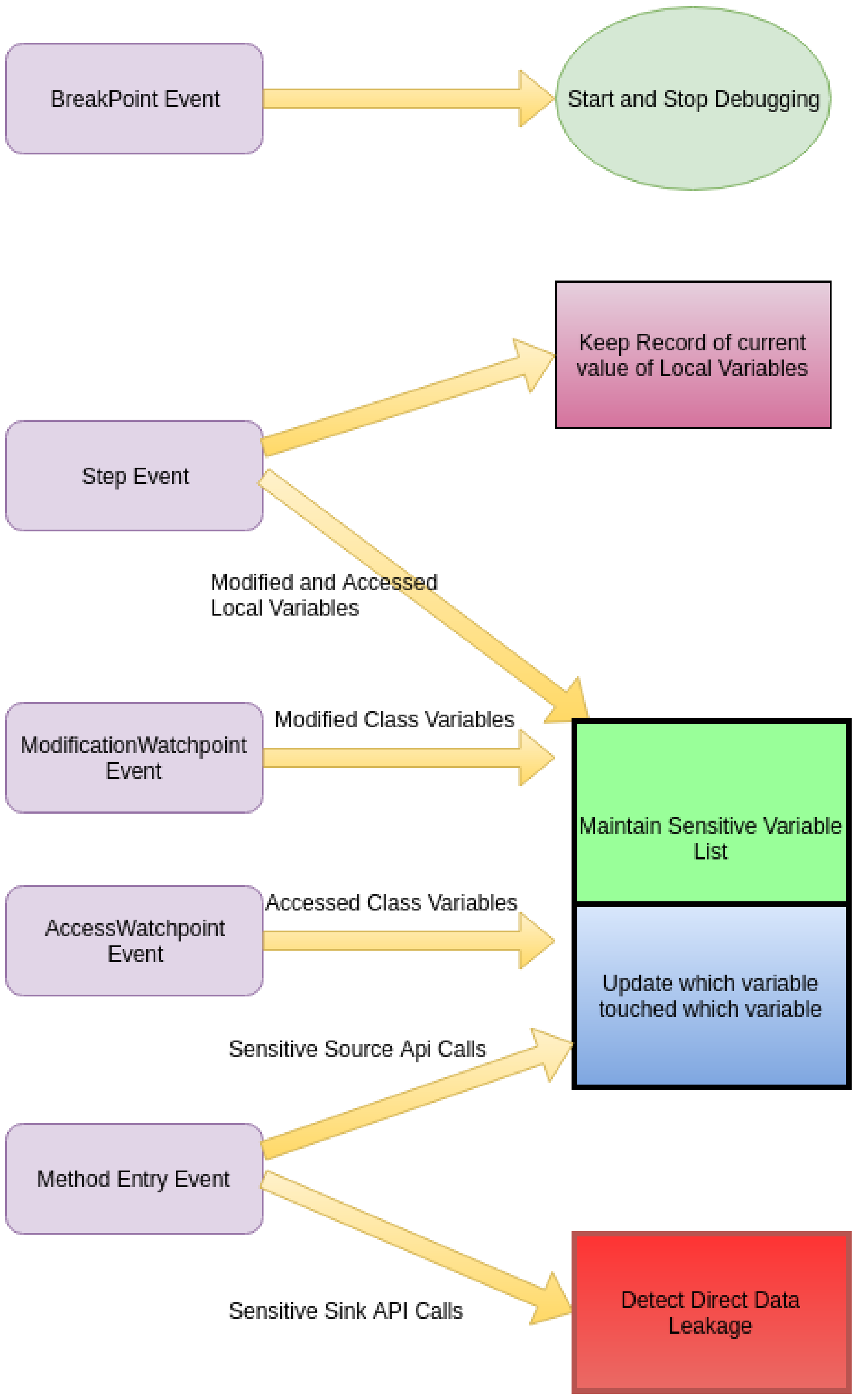

BreakPoint Event
Start and Stop Debugging
Keep Record of current value of Local Variables
Step Event
Modified and Accessed Local Variables
ModificationWatchpoint Event
Modified Class Variables
Maintain Sensitive Variable List
AccessWatchpoint Event
Accessed Class Variables
Update which variable touched which variable
Sensitive Source Api Calls
Method Entry Event
Sensitive Sink API Calls
Detect Direct Data Leakage

Figure 5.1: Flow diagram of debugger built using JDI library

to find variable accessed inside method, `MethodEntry` events were required. This required to catch each and every `MethodEntry` event in the program. Since extracting out variables inside methods required suspending threads and analyzing stack-frames this caused the debugger to run very slow.

- Also in this approach for monitoring local variables thread-frames were examined. The values of variables were matched to find which variable is flowing into which variable. This approach was also slowing debugger since threads are suspended in this approach. Also this approach caused to miss many cases.

So because of above problems it was not possible to cover all cases due to performance constraints and limitations in covering all cases.

5.2 Using JDI and JavaParser Libraries

In section 5.1.3 we reached to the conclusion that we can not use JDI library alone. We needed a faster approach to understand the flow of information in the source code more accurately. So we used JavaParser library in our work. Next we describe how we can use JavaParser library to find modified and accessed variables from each Java expressions to know flow of information. After that we discuss implementation part of our current approach which uses both JDI and JavaParser libraries.

5.2.1 Finding variables flowing into a variable

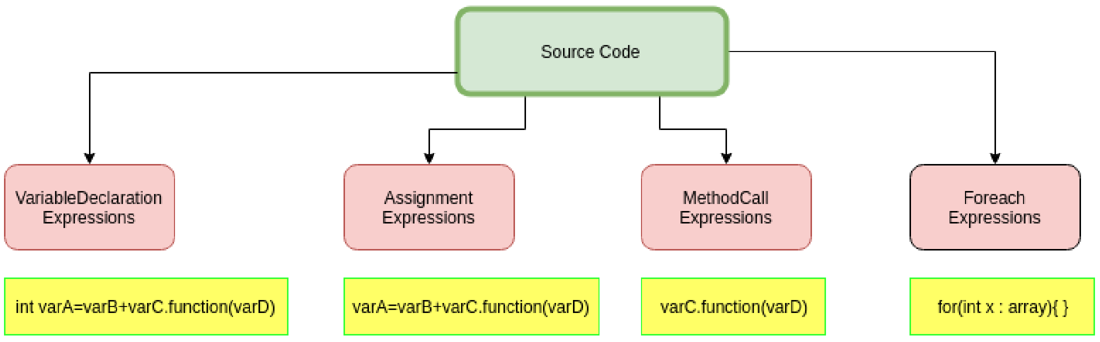


Figure 5.2: Java expression partition

We divided data transfer expressions in Java into four categories. We discuss each of them in next section. For each category we define which variable is getting its value modified and what variables it is touching. In implementation part this basic idea is used to track information flow. For all of the four Expression types we used `AssignExpr` and `FieldAccessExpr` node types in JavaParser library [16] to identify variables used just by their names e.g., 'varA' and variables whose name contains a 'dot' e.g., 'objectA.lenght' respectively.

Next we are going to discuss all four expressions:

1. **MethodCall Expressions**

   These expressions look like

   ```
   variableA.funcionCall(variableB,variableC.....)
   ```

   In all expressions of above type it is declared that values of function arguments variableB, variableC etc are flowing into variableA.

2. **VariableDeclaration Expressions**

   These expressions look like

   ```
   int variableX=variableZ+variableA.funcionCall(variableB,variableC.....)
   ```

   In all expressions of above type it is declared that values of variables appearing on right side of '=' are flowing into variable appearing on left side of '='.

3. **Assignment Expressions**

   These expressions look like

   ```
   variableD=variableZ+variableA.funcionCall(variableB,variableC.....)
   ```

   In all expressions of above type it is declared that values of variables appearing on right side of '=' are flowing into variable appearing on left side of '='.

4. **Foreach Expressions**

   These expressions look like

   ```
   for(int x:array){ }
   ```

   In all expressions of above type it is declared that values of variables appearing on right side of ‘:’ are flowing into variable appearing on left side of ‘:’.

5.2.2 Implementation

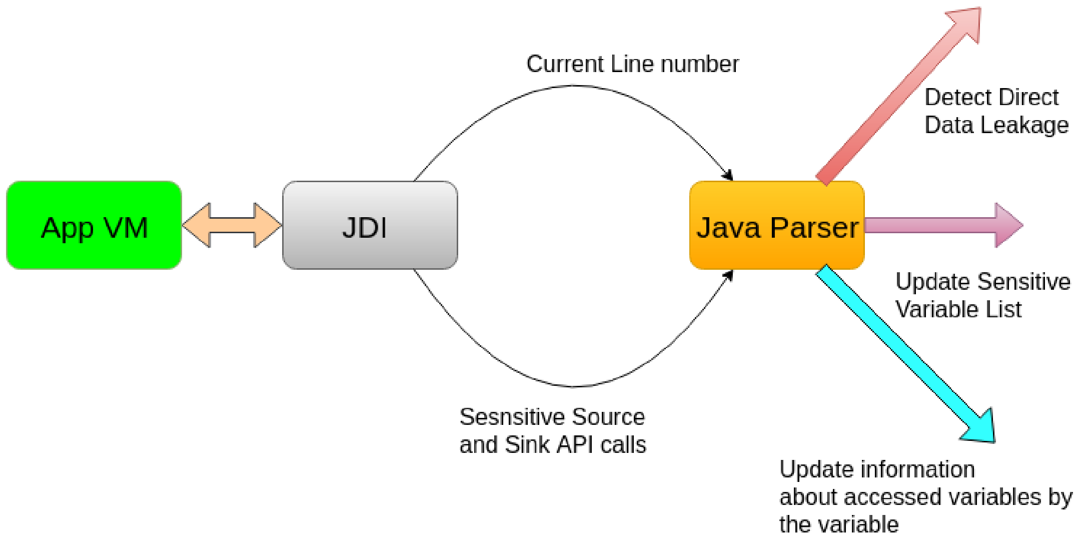

Current Line number
Detect Direct Data Leakage
App VM
JDI
Java Parser
Update Sensitive Variable List
Sesnsitive Source and Sink API calls
Update information about accessed variables by the variable

Figure 5.3: Flow diagram of debugger built using both JDI and JavaParser libraries

The full debugger involves two parts the first one written using JDI library [9] and the other one written using JavaParser library [16].

JDI Part

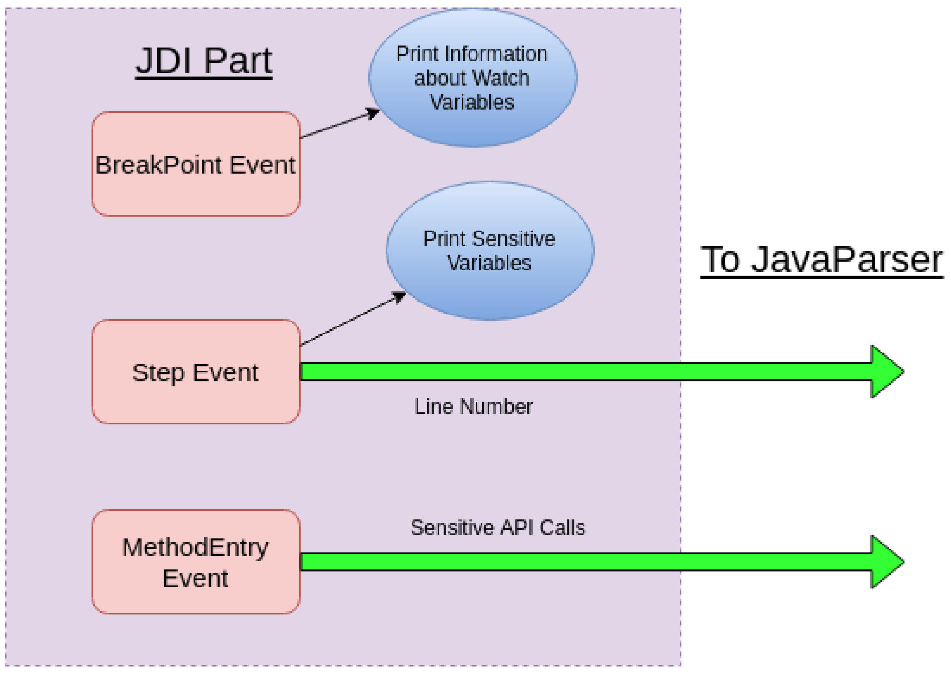


Figure 5.4: JDI Part flow diagram

JDI Part is mainly responsible for interacting with running app and sending useful information to JavaParser Part as shown in Figure 5.3. It uses `BreakPoint` Events to know when user selected breakpoints are getting hit, `Step` Events to be notified after one line execution and `MethodEntry` Events to know if sensitive source or sinks API was called on current line. Its internal working is shown in Figure 5.4. JDI Part does following tasks:-

- Interact with App's virtual machine
- Send line number of current line under execution to JavaParser Part
- Find sensitive API calls on current line and send them to JavaParser Part
- Run Program step by step
- On breakpoints, for each watch variable print what variables they have touched beginning with first breakpoint
- Print current sensitive variables line by line

JavaParser Library [16] Part

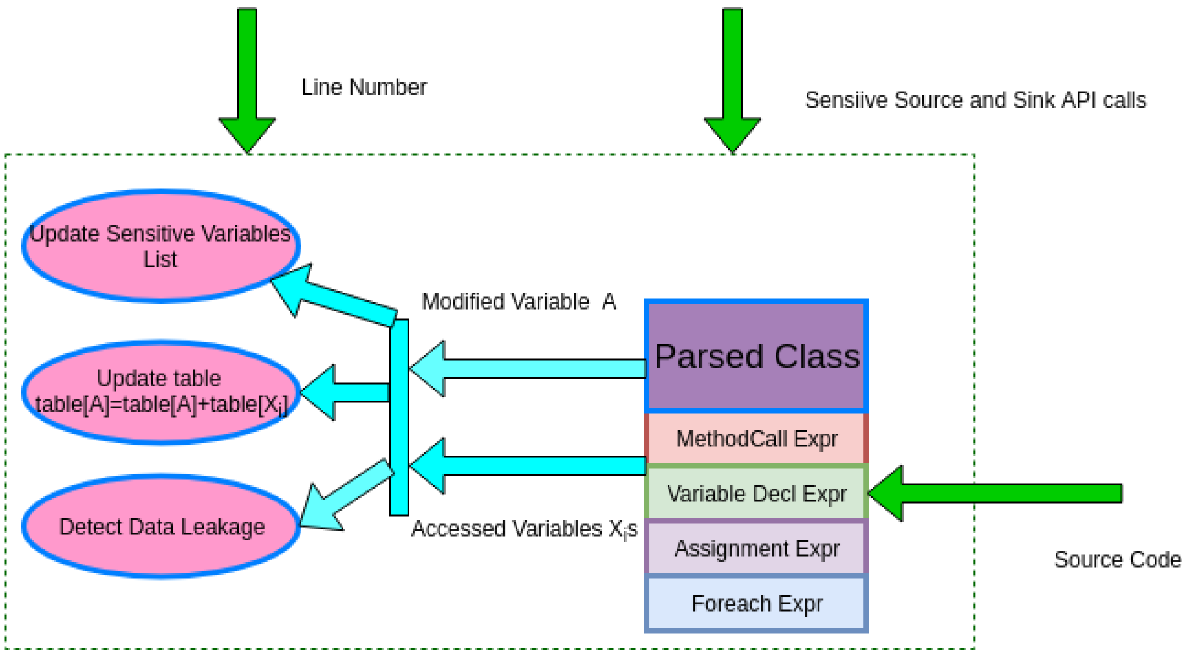


Figure 5.5: JavaParser Part flow diagram

JavaParser library [16] was used to detect out-going data and track the information flow in the Java code. Using JDI part, which provides line number of current line under execution, JavaParser part analyses current line under execution from parsed current line from source code. Before connecting debugger, source file is parsed to get all **MethodCall, VariableDeclaration, Foreach** and **Assignment** expressions line by line and are stored in a table. Full working of JavaParser Part is explained in Figure 5.5. JavaParser part does following tasks while interacting with JDI part.

Line by line keep track of sensitive variables

Java Parser part gets current line under execution and sensitive API call from JDI part. It keeps a list of sensitive variables. It does following things to maintain this list :-

- First it finds variable A in current line which is getting its value changed
- If there is any sensitive API call then add A to sensitive variables
- If A touches any variable B which was sensitive add it to sensitive variables list
- If before execution of current line A was in sensitive variables list, and A's value is overwritten by any constant or nonsensitive variables on current line then A is removed from sensitive variables list.

For each variable keep record of all variables it has touched

Using **MethodCall, VariableDeclaration** and **Assignment** expressions data can go from one variable to another. JavaParser maintains a table to keep track of all variables that a specific variable has touched. It does following things to fill that table:-

- Get variable A which is getting its value modified on current line
- Find all other variables X_is used on this line
- Append X_is and their table entries to table entry of A

Detect Direct data leakage

If any sensitive variables is being sent outside through some network interface then it can be possible data leakage, so it is reported. This task involves following steps:-

- Get sensitive sink method calls for current line from JDI Part
- Using JavaParser part find MethodCall expressions for those API calls and check their arguments contain sensitive variables, if yes then report data leakage

Chapter 6

Using SecDroidBug

6.1 Usage

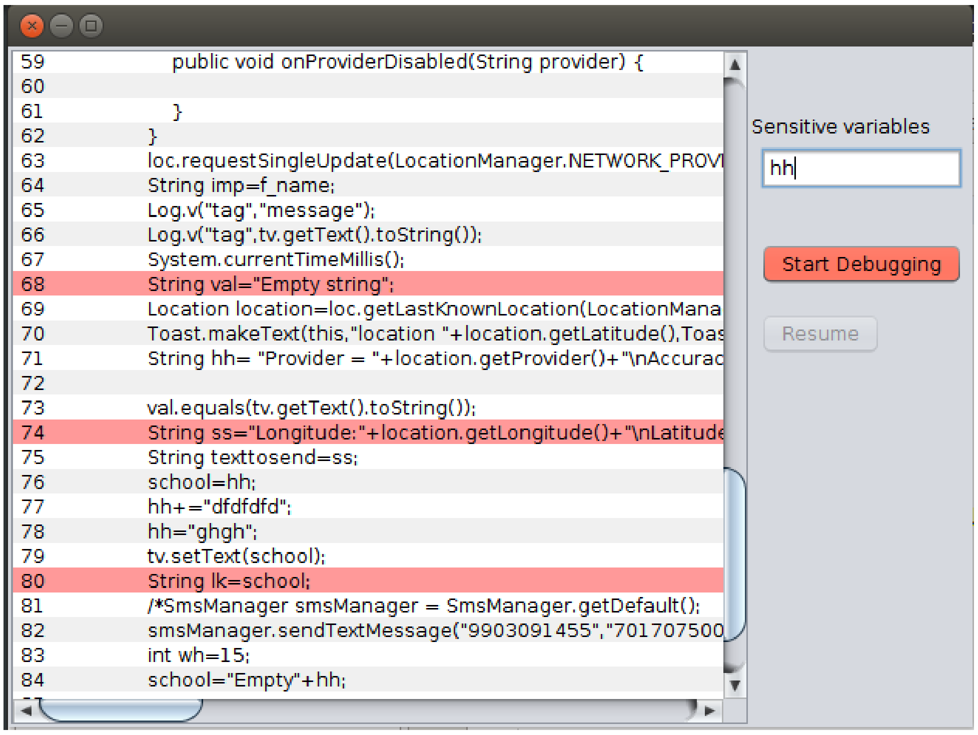
When user runs debugger he is shown file choose dialog, after selecting source file to be debugged, he is shown window showing him full source code of the file. All GUI interface in SecDroidBug is developed using Java Swing framework. By clicking on the line number breakpoints can be set and unset. A click on start debugging button starts debugging. When App hits any breakpoint except last and first it is suspended until user clicks on resume button. All sensitive variable information line by line shown on console and on each breakpoint the variable which user requested to be monitored are printed with information of variables that they have touched.

The debugger front end is shown in Figure 6.1. The high level working of debugger can explained by the Figure 3.1 where debugger takes Watch variables and breakpoints from user and output sensitive variables line by line, reports direct data leakage and variables touched by watch variables on breakpoints.

6.2 Running the debugger

To run Debugger use following steps

- Make sure your App is set as debuggable
- Run in terminal following command to install and to run APK in debug mode and



```
            public void onProviderDisabled(String provider) {

            }
        }
        loc.requestSingleUpdate(LocationManager.NETWORK_PROVI
        String imp=f_name;
        Log.v("tag","message");
        Log.v("tag",tv.getText().toString());
        System.currentTimeMillis();
        String val="Empty string";
        Location location=loc.getLastKnownLocation(LocationMana
        Toast.makeText(this,"location "+location.getLatitude(),Toas
        String hh= "Provider = "+location.getProvider()+"\nAccurac

        val.equals(tv.getText().toString());
        String ss="Longitude:"+location.getLongitude()+"\nLatitude
        String texttosend=ss;
        school=hh;
        hh+="dfdfdfd";
        hh="ghgh";
        tv.setText(school);
        String lk=school;
        /*SmsManager smsManager = SmsManager.getDefault();
        smsManager.sendTextMessage("9903091455","701707500
        int wh=15;
        school="Empty"+hh;
```

Figure 6.1: Debugger GUI

to make the App wait for Debugger to connect

```
./run.sh <apk_file_name>
```

- Put ‘sinks_a’ and ‘sources_a’ and SecDroidBug.jar in same folder
- Run following command to launch Debugger

```
java -jar SecDroidBug.jar
```

- Select Java source file to be debugged using shown file choose dialog box
- Mark breakpoints and fill watch variable names in the textbox
- Click on ‘Start Debugging’ button and click on ‘Resume’ button to resume the suspended App when ‘Resume’ button goes green

Using all debugging commands supported by SecDroidBug some methods in different Apps were debugged to show working of our debugger. They are discussed in Chapter 8. In next chapter we will discuss other used with our debugger to do testing on Android Apps.

Chapter 7

Tools used for Logging and Automating the Debug Process

In this chapter we will discuss some tools used by us during testing of Android Apps using SecDroidBug. For testing any method with SecDroidBug, it is necessary to hit marked breakpoints. Because its not easy to find GUI interaction to be performed with the App to reach upto that line, it becomes hard to do this task with any random App. For this task we tried to use following four tools. Out of these four, DroidMate could not be used with our Debugger because it applies App instrumentation, so marked breakpoints correspond to different lines on instrumented App. DroidBot performed best for exploring GUI execution path to reach upto breakpoints. AndroidViewClient was helpful to generate GUI execution tests, which were helpful for debugging methods coming under execution under those tests. From following all tools, tool described in Section 7.1 was used most because it was hard to explore App with other tools to hit one breakpoint as it was too much time consuming.

7.1 ApiCallsWithDeb to get App method calls

We developed a small debugger ApiCallsWithDeb [2] using JDI [9]. This was used to find what App methods are getting called on performing interaction with Android device. These methods were logged and were debugged with full debugger after performing same UI interaction with the App. This tool was used the most in testing.

7.2 Android Monkey

The Monkey [1] is a command-line tool to generate pseudo-random user events on the Android device. usually it is used to stress test any Android App. Because it generates random events its code coverage is not very good. But it can perform very fast and can explore some paths which were hard to discover using any model based tool.

Its parameters can be tuned to change speed of events, type of events, number of events and restricting the events to certain part only ex:-packages, Apps etc.

7.3 DroidBot

DroidBot [10] is automatic GUI executor based on GUI model. Its coding coverage was best among all four tools. It has following features:-

- Good code coverage quickly
- Does not fill text fields, but it supports external scripts to fill texts
- No App instrumentation
- Works with latest API versions and devices
- Keeps control within App / Returns control back to App in case of irrelevant exploration
- produce UI structures and method traces for analysis
- Runs App in continuous mode

7.4 DroidMate

DroidMate [8] repeatedly reads device GUI at runtime and based on monitored calls to Android APIs methods it decides next GUI action to taken. The parameters required for running DroidMate are exploration strategy, termination criterion and a set of monitored methods. It runs without root or OS modifications. But since it performs instrumentation of Android App and it does not support recent Android API versions this App was not much tested with our Debugger.

7.5 AndroidViewClient

AndroidViewClient [12] is a tool that automates and simplifies test script generation. These scripts are python scripts which when run, will perform same UI interaction as done at the time of creation of those test script, with the device. On running this tool it will show some windows where Android devices' live display is shown. User can click, scroll and do other GUI interactions with the window which will also be relayed to Android device, these all interactions will be saved as a python script, which will perform same interaction with device on running it.

AndroidViewClient also has a mode called Culebra concertina mode. In this mode AndroidViewClient works as automatic GUI executor. It analyses the contents/views of the current screen and randomly selects a suitable event or action for randomly chosen view. In experiments we found that this tool gets stuck on some Apps.

7.6 Comparison of the tools used

Table 7.1: Tools Comparison

Tool	**Fills form**	**App Instru- -mentation**	**Strategy**	**Supported Android versions**	**Code Coverage**
Android- -ViewClient	Yes	No	Random View & Random Event	All versions	Ok
DroidBot	No	No	Model	All versions	Good
DroidMate	No	Yes	Model	API 19 & 23	Good
Monkey	No	No	Random	All versions	Less
ApiCallsWithDeb	Manual	No	Manual	All versions	Manual

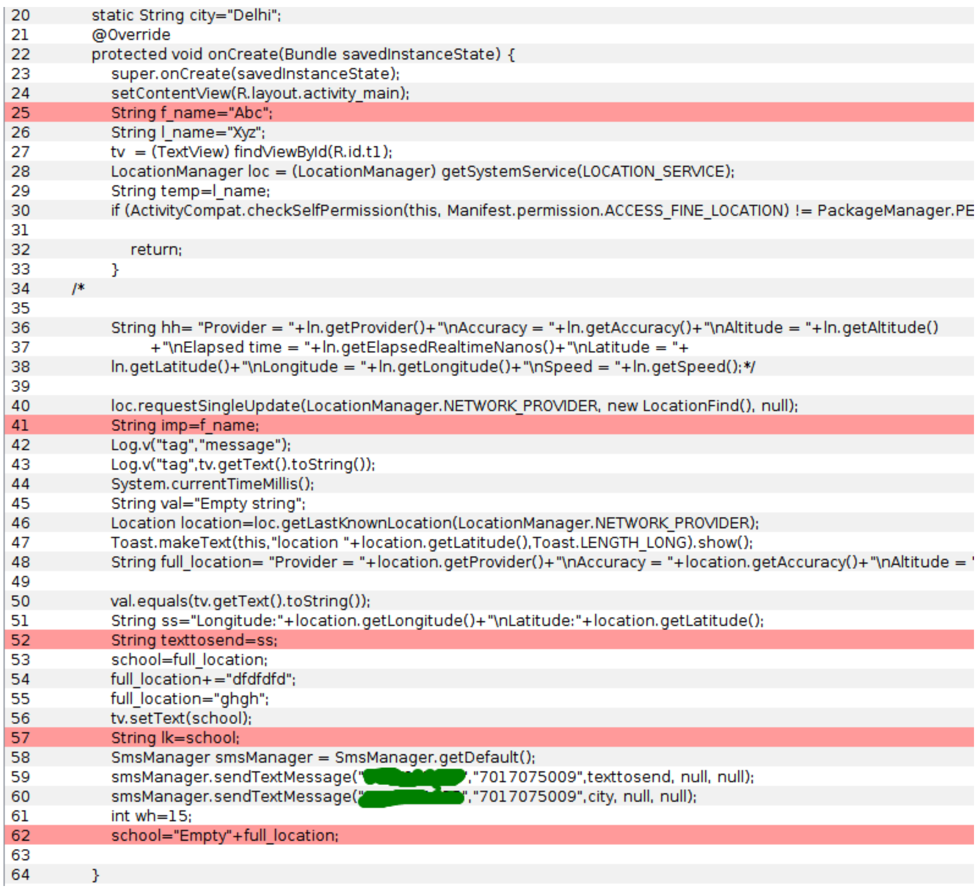
Chapter 8

Experimental Analysis and Results

In this chapter we will explain results of debugging performed on different Apps. We have showed debugger output on testing some of methods of these Apps. We will explain tests on sample App in detail since it contains different cases to show our debugger working clearly. We then show our debugging results on all other Apps.

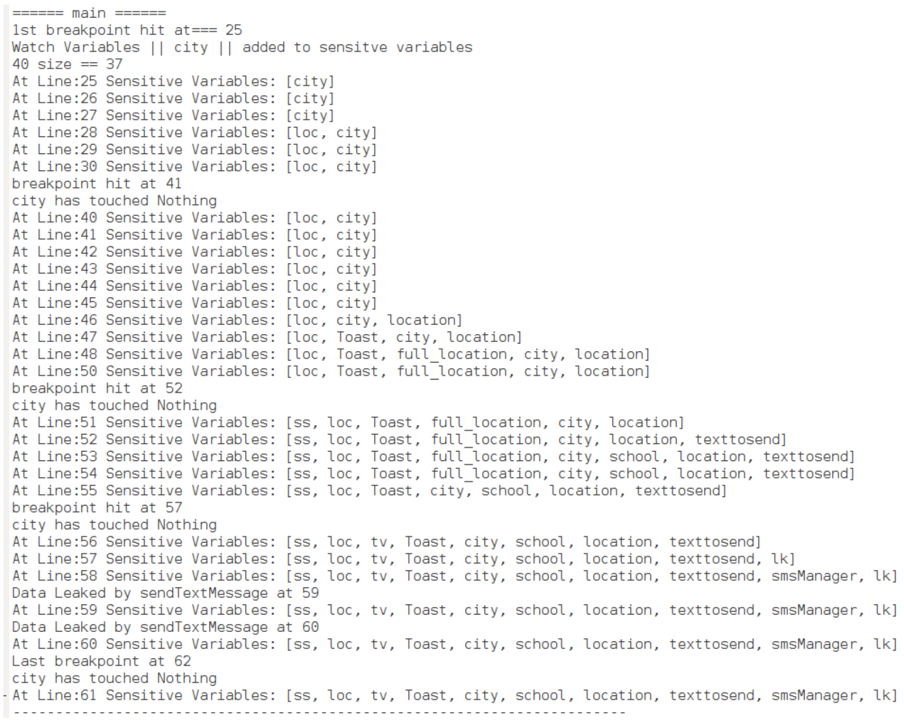
Experiments on sample App

This sample App reads users location and sends it through sms to outside (see Figure 8.1). This could be possible data leakage. Also we marked '`city`' variable as Watch Variable, so it also becomes sensitive. We can see in debugger output that '`city`' variable is sent through SMS so direct data leakage is reported. Also we can see '`texttosend`' variable also becomes sensitive variable after touching variable '`ss`' on line 52 (see Figure 8.2). When it is sent outside on line 59, it is also reported. There are three intermediate breakpoints. On each breakpoint information about what variables '`city`' variable has touched is printed. On each line all sensitive variables are printed. We can see if any variable touches any sensitive variable it also becomes sensitive. For example '`loc`' variable on line 28 becomes sensitive for storing result from sensitive source. Variable 'location' touches '`loc`' on line 46 and becomes sensitive. Similarly this information flows to other variables and they too becomes sensitive. So now we can see that the debugger can do all three task printing sensitive variables line by line, printing variables touched by watch variables line by line and detecting direct data leakage.



```
20        static String city="Delhi";
21        @Override
22        protected void onCreate(Bundle savedInstanceState) {
23            super.onCreate(savedInstanceState);
24            setContentView(R.layout.activity_main);
25            String f_name="Abc";
26            String l_name="Xyz";
27            tv = (TextView) findViewById(R.id.t1);
28            LocationManager loc = (LocationManager) getSystemService(LOCATION_SERVICE);
29            String temp=l_name;
30            if (ActivityCompat.checkSelfPermission(this, Manifest.permission.ACCESS_FINE_LOCATION) != PackageManager.PE
31
32                return;
33            }
34    /*
35
36            String hh= "Provider = "+ln.getProvider()+"\nAccuracy = "+ln.getAccuracy()+"\nAltitude = "+ln.getAltitude()
37                    +"\nElapsed time = "+ln.getElapsedRealtimeNanos()+"\nLatitude = "+
38            ln.getLatitude()+"\nLongitude = "+ln.getLongitude()+"\nSpeed = "+ln.getSpeed();*/
39
40            loc.requestSingleUpdate(LocationManager.NETWORK_PROVIDER, new LocationFind(), null);
41            String imp=f_name;
42            Log.v("tag","message");
43            Log.v("tag",tv.getText().toString());
44            System.currentTimeMillis();
45            String val="Empty string";
46            Location location=loc.getLastKnownLocation(LocationManager.NETWORK_PROVIDER);
47            Toast.makeText(this,"location "+location.getLatitude(),Toast.LENGTH_LONG).show();
48            String full_location= "Provider = "+location.getProvider()+"\nAccuracy = "+location.getAccuracy()+"\nAltitude = '
49
50            val.equals(tv.getText().toString());
51            String ss="Longitude:"+location.getLongitude()+"\nLatitude:"+location.getLatitude();
52            String texttosend=ss;
53            school=full_location;
54            full_location+="dfdfdfd";
55            full_location="ghgh";
56            tv.setText(school);
57            String lk=school;
58            SmsManager smsManager = SmsManager.getDefault();
59            smsManager.sendTextMessage("[redacted]","7017075009",texttosend, null, null);
60            smsManager.sendTextMessage("[redacted]","7017075009",city, null, null);
61            int wh=15;
62            school="Empty"+full_location;
63
64        }
```

Figure 8.1: Experiment on sample App: source code



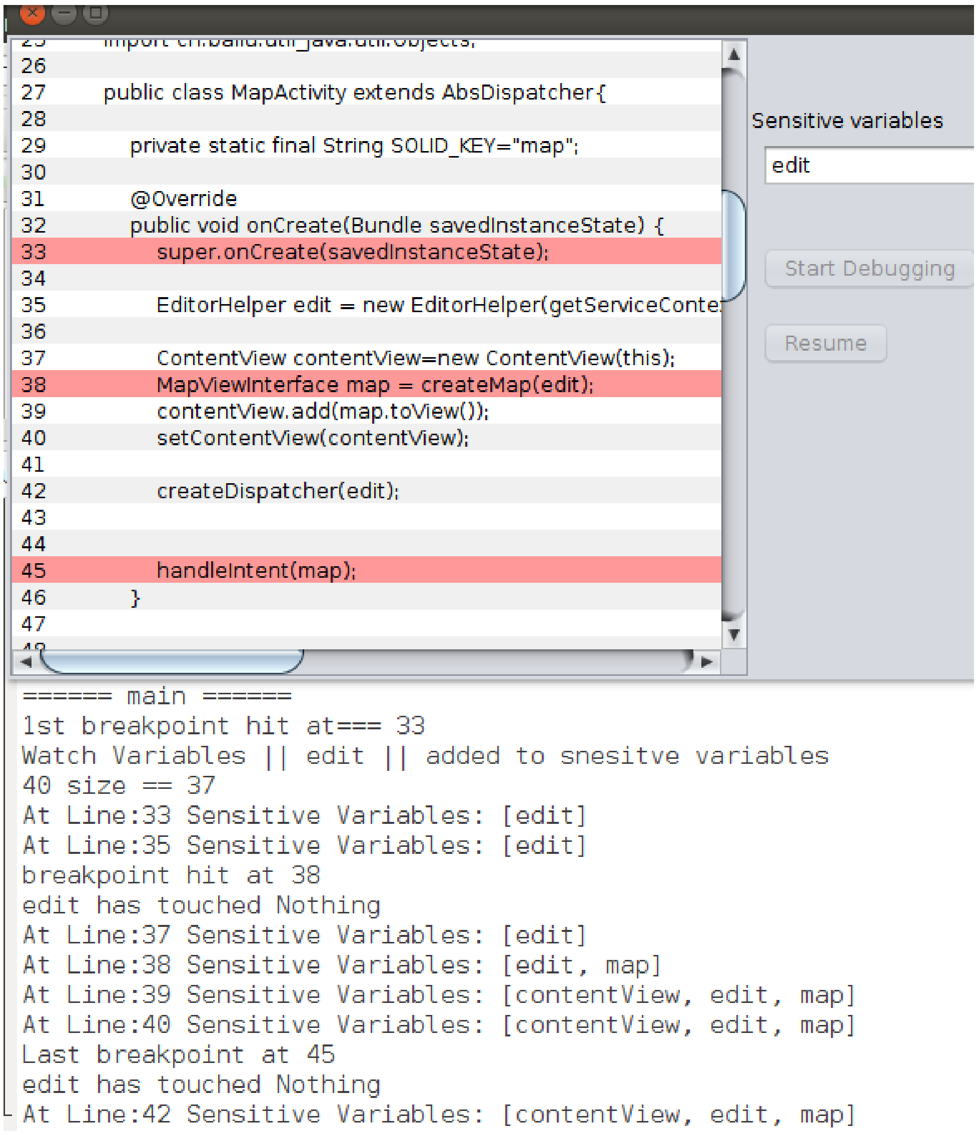
```
====== main ======
1st breakpoint hit at=== 25
Watch Variables || city || added to sensitve variables
40 size == 37
At Line:25 Sensitive Variables: [city]
At Line:26 Sensitive Variables: [city]
At Line:27 Sensitive Variables: [city]
At Line:28 Sensitive Variables: [loc, city]
At Line:29 Sensitive Variables: [loc, city]
At Line:30 Sensitive Variables: [loc, city]
breakpoint hit at 41
city has touched Nothing
At Line:40 Sensitive Variables: [loc, city]
At Line:41 Sensitive Variables: [loc, city]
At Line:42 Sensitive Variables: [loc, city]
At Line:43 Sensitive Variables: [loc, city]
At Line:44 Sensitive Variables: [loc, city]
At Line:45 Sensitive Variables: [loc, city]
At Line:46 Sensitive Variables: [loc, city, location]
At Line:47 Sensitive Variables: [loc, Toast, city, location]
At Line:48 Sensitive Variables: [loc, Toast, full_location, city, location]
At Line:50 Sensitive Variables: [loc, Toast, full_location, city, location]
breakpoint hit at 52
city has touched Nothing
At Line:51 Sensitive Variables: [ss, loc, Toast, full_location, city, location]
At Line:52 Sensitive Variables: [ss, loc, Toast, full_location, city, location, texttosend]
At Line:53 Sensitive Variables: [ss, loc, Toast, full_location, city, school, location, texttosend]
At Line:54 Sensitive Variables: [ss, loc, Toast, full_location, city, school, location, texttosend]
At Line:55 Sensitive Variables: [ss, loc, Toast, city, school, location, texttosend]
breakpoint hit at 57
city has touched Nothing
At Line:56 Sensitive Variables: [ss, loc, tv, Toast, city, school, location, texttosend]
At Line:57 Sensitive Variables: [ss, loc, tv, Toast, city, school, location, texttosend, lk]
At Line:58 Sensitive Variables: [ss, loc, tv, Toast, city, school, location, texttosend, smsManager, lk]
Data Leaked by sendTextMessage at 59
At Line:59 Sensitive Variables: [ss, loc, tv, Toast, city, school, location, texttosend, smsManager, lk]
Data Leaked by sendTextMessage at 60
At Line:60 Sensitive Variables: [ss, loc, tv, Toast, city, school, location, texttosend, smsManager, lk]
Last breakpoint at 62
city has touched Nothing
At Line:61 Sensitive Variables: [ss, loc, tv, Toast, city, school, location, texttosend, smsManager, lk]
-------------------------------------------------------------------------
```

Figure 8.2: Experiment on sample App: debugger Output

Experiments on Apps from Google Play Store and F-Droid Android App repository

We tried testing with our debugger on some Apps from Google Play Store and F-Droid. We were successfully able to debug all of them. The reason our tool worked with all of them was that we are using debugger API which is used in many IDEs and can be used to debug any debuggable App. Each experiment shows source code of App with breakpoints set and generated output by debugger after debugging. Watch Variables can be seen from debugger output.

AAT - Another Activity Tracker



Sensitive variables

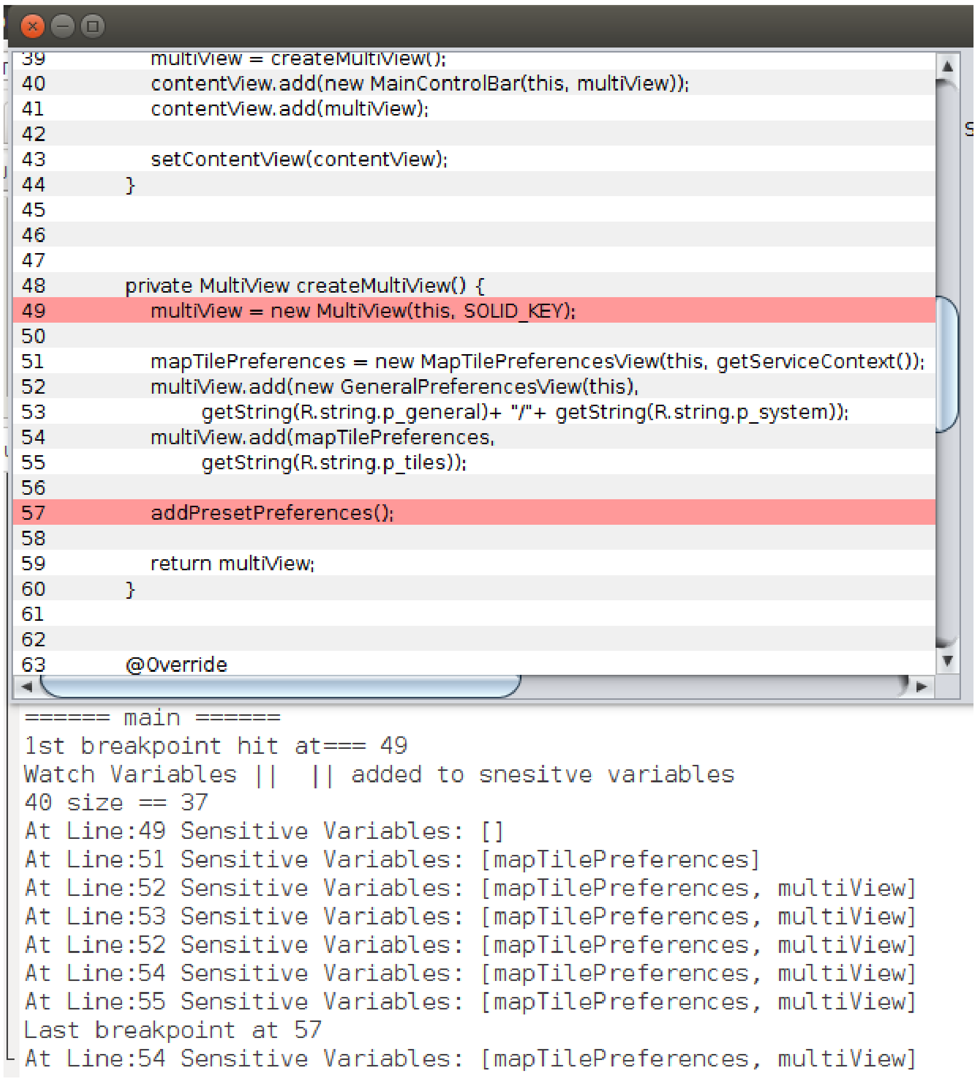
edit

```
26
27      public class MapActivity extends AbsDispatcher{
28
29          private static final String SOLID_KEY="map";
30
31          @Override
32          public void onCreate(Bundle savedInstanceState) {
33              super.onCreate(savedInstanceState);
34
35              EditorHelper edit = new EditorHelper(getServiceConte
36
37              ContentView contentView=new ContentView(this);
38              MapViewInterface map = createMap(edit);
39              contentView.add(map.toView());
40              setContentView(contentView);
41
42              createDispatcher(edit);
43
44
45              handleIntent(map);
46          }
47
```

```
====== main ======
1st breakpoint hit at=== 33
Watch Variables || edit || added to snesitve variables
40 size == 37
At Line:33 Sensitive Variables: [edit]
At Line:35 Sensitive Variables: [edit]
breakpoint hit at 38
edit has touched Nothing
At Line:37 Sensitive Variables: [edit]
At Line:38 Sensitive Variables: [edit, map]
At Line:39 Sensitive Variables: [contentView, edit, map]
At Line:40 Sensitive Variables: [contentView, edit, map]
Last breakpoint at 45
edit has touched Nothing
At Line:42 Sensitive Variables: [contentView, edit, map]
```

Figure 8.3: Experiment-1 on ‘AAT - Another Activity Tracker’ App

In experiment-1 on ‘Another Activity Tracker’ App we had provided ‘`edit`’ as watch variable. It is treated as sensitive throughout the debugging process. On line-38 ‘`map`’variable touches ‘`edit`’and becomes sensitive, similarly on line-39 ‘`contentView`’ becomes sensitive on touching ‘`map`’variable. At the end for watch variable ‘`edit`’it is printed that it has not touched any variable.



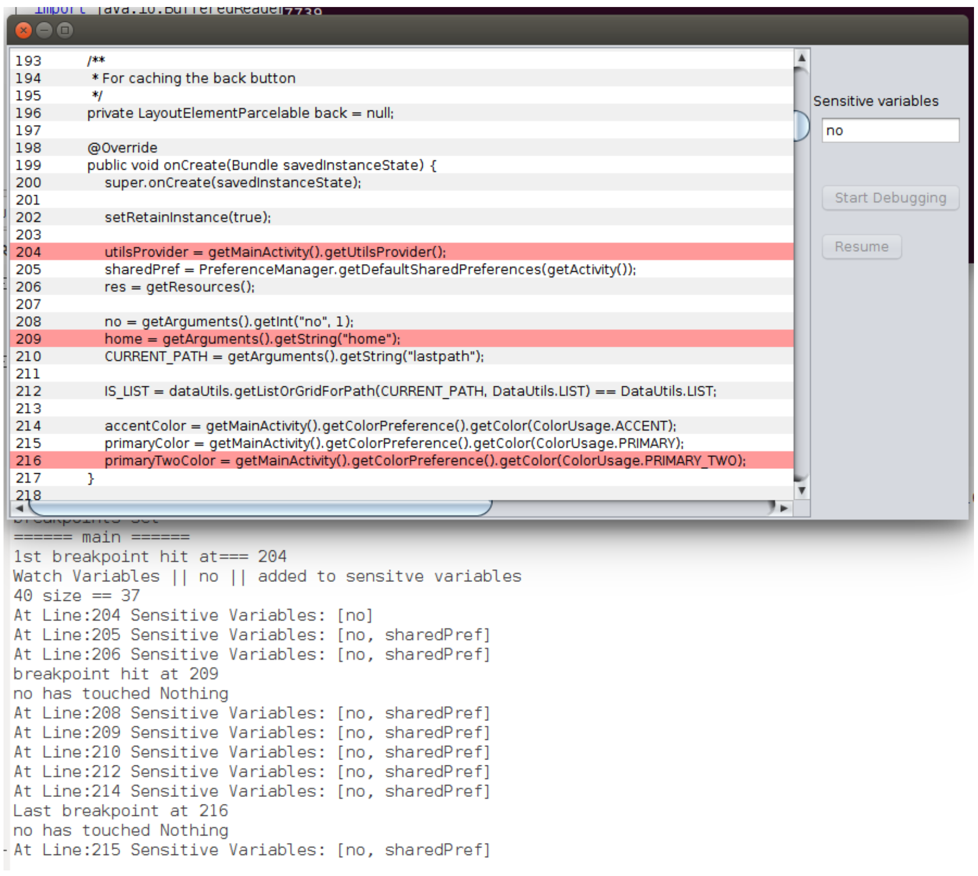
```
39          multiView = createMultiView();
40          contentView.add(new MainControlBar(this, multiView));
41          contentView.add(multiView);
42
43          setContentView(contentView);
44      }
45
46
47
48      private MultiView createMultiView() {
49          multiView = new MultiView(this, SOLID_KEY);
50
51          mapTilePreferences = new MapTilePreferencesView(this, getServiceContext());
52          multiView.add(new GeneralPreferencesView(this),
53                  getString(R.string.p_general)+ "/"+ getString(R.string.p_system));
54          multiView.add(mapTilePreferences,
55                  getString(R.string.p_tiles));
56
57          addPresetPreferences();
58
59          return multiView;
60      }
61
62
63      @Override
```

```
====== main ======
1st breakpoint hit at=== 49
Watch Variables ||  || added to snesitve variables
40 size == 37
At Line:49 Sensitive Variables: []
At Line:51 Sensitive Variables: [mapTilePreferences]
At Line:52 Sensitive Variables: [mapTilePreferences, multiView]
At Line:53 Sensitive Variables: [mapTilePreferences, multiView]
At Line:52 Sensitive Variables: [mapTilePreferences, multiView]
At Line:54 Sensitive Variables: [mapTilePreferences, multiView]
At Line:55 Sensitive Variables: [mapTilePreferences, multiView]
Last breakpoint at 57
At Line:54 Sensitive Variables: [mapTilePreferences, multiView]
```

Figure 8.4: Experiment-2 on ‘AAT - Another Activity Tracker’ App

In experiment-2 there was no watch variable provided by the user. So at the start of debugging process there is no sensitive variable. Variables ‘`mapTilePreferences`’ and ‘`multiView`’becomes sensitive on storing result of sensitive source API calls.

Amaze File Manager



```
193     /**
194      * For caching the back button
195      */
196     private LayoutElementParcelable back = null;
197
198     @Override
199     public void onCreate(Bundle savedInstanceState) {
200         super.onCreate(savedInstanceState);
201
202         setRetainInstance(true);
203
204         utilsProvider = getMainActivity().getUtilsProvider();
205         sharedPref = PreferenceManager.getDefaultSharedPreferences(getActivity());
206         res = getResources();
207
208         no = getArguments().getInt("no", 1);
209         home = getArguments().getString("home");
210         CURRENT_PATH = getArguments().getString("lastpath");
211
212         IS_LIST = dataUtils.getListOrGridForPath(CURRENT_PATH, DataUtils.LIST) == DataUtils.LIST;
213
214         accentColor = getMainActivity().getColorPreference().getColor(ColorUsage.ACCENT);
215         primaryColor = getMainActivity().getColorPreference().getColor(ColorUsage.PRIMARY);
216         primaryTwoColor = getMainActivity().getColorPreference().getColor(ColorUsage.PRIMARY_TWO);
217     }
218
```

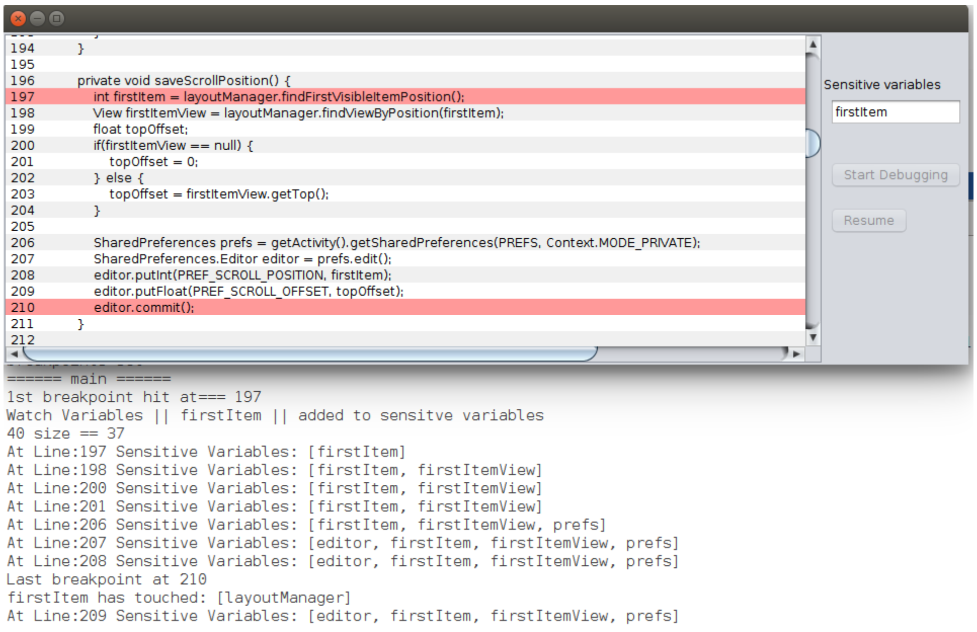
Sensitive variables: no

```
====== main ======
1st breakpoint hit at=== 204
Watch Variables || no || added to sensitve variables
40 size == 37
At Line:204 Sensitive Variables: [no]
At Line:205 Sensitive Variables: [no, sharedPref]
At Line:206 Sensitive Variables: [no, sharedPref]
breakpoint hit at 209
no has touched Nothing
At Line:208 Sensitive Variables: [no, sharedPref]
At Line:209 Sensitive Variables: [no, sharedPref]
At Line:210 Sensitive Variables: [no, sharedPref]
At Line:212 Sensitive Variables: [no, sharedPref]
At Line:214 Sensitive Variables: [no, sharedPref]
Last breakpoint at 216
no has touched Nothing
At Line:215 Sensitive Variables: [no, sharedPref]
```

Figure 8.5: Experiment on 'Amaze File Manager' App

In the experiment on 'Amaze File Manager' App there were three breakpoints and variable '`no`'was used as watch variable and thus maintained in sensitive variable list throughout debugging process. Result says there were only two sensitive variables as '`sharedPref`' becomes sensitive on line-205.

AntennaPod

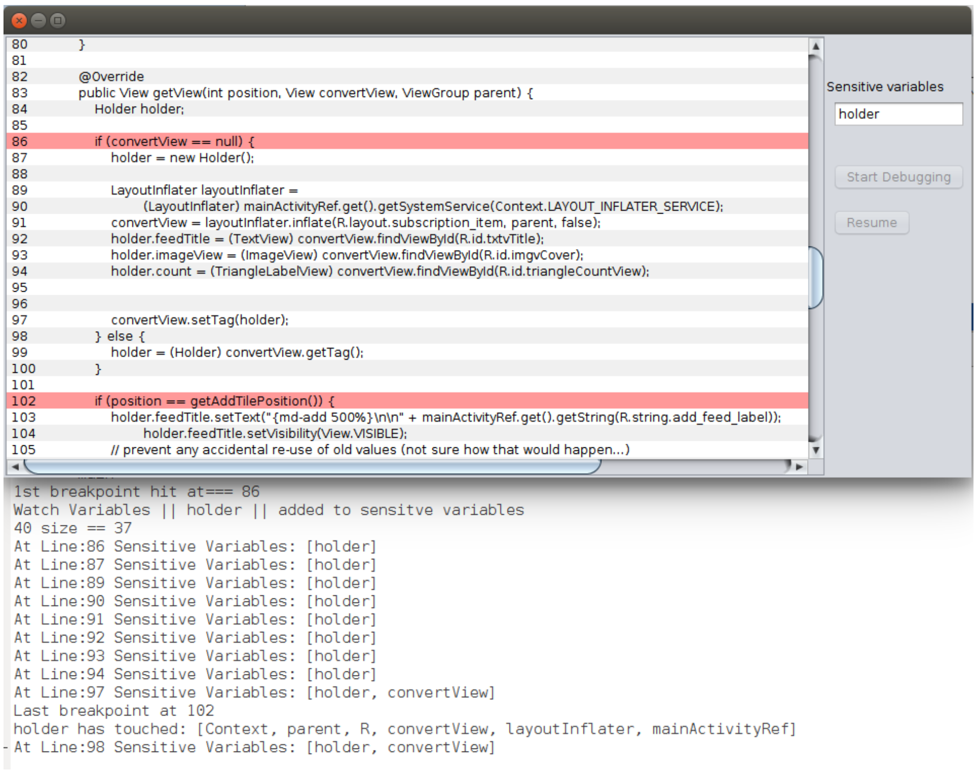


```
194       }
195
196       private void saveScrollPosition() {
197           int firstItem = layoutManager.findFirstVisibleItemPosition();
198           View firstItemView = layoutManager.findViewByPosition(firstItem);
199           float topOffset;
200           if(firstItemView == null) {
201               topOffset = 0;
202           } else {
203               topOffset = firstItemView.getTop();
204           }
205
206           SharedPreferences prefs = getActivity().getSharedPreferences(PREFS, Context.MODE_PRIVATE);
207           SharedPreferences.Editor editor = prefs.edit();
208           editor.putInt(PREF_SCROLL_POSITION, firstItem);
209           editor.putFloat(PREF_SCROLL_OFFSET, topOffset);
210           editor.commit();
211       }
212
```

```
====== main ======
1st breakpoint hit at=== 197
Watch Variables || firstItem || added to sensitve variables
40 size == 37
At Line:197 Sensitive Variables: [firstItem]
At Line:198 Sensitive Variables: [firstItem, firstItemView]
At Line:200 Sensitive Variables: [firstItem, firstItemView]
At Line:201 Sensitive Variables: [firstItem, firstItemView]
At Line:206 Sensitive Variables: [firstItem, firstItemView, prefs]
At Line:207 Sensitive Variables: [editor, firstItem, firstItemView, prefs]
At Line:208 Sensitive Variables: [editor, firstItem, firstItemView, prefs]
Last breakpoint at 210
firstItem has touched: [layoutManager]
At Line:209 Sensitive Variables: [editor, firstItem, firstItemView, prefs]
```

Figure 8.6: Experiment-1 on 'AntennaPod' App

In experiment-1 on 'AntennaPod' App, we provided '`firstItem`' variable as watch variable. '`firstItemView`' becomes sensitive on touching watch variable '`firstitem`' on line-198. '`prefs`' variable becomes sensitive on touching sensitive data on line-206. It is accessed by '`editor`' variable which also becomes sensitive on line-207. Finally it is printed on last breakpoint that '`firstitem`' accessed '`layoutManager`' variable.



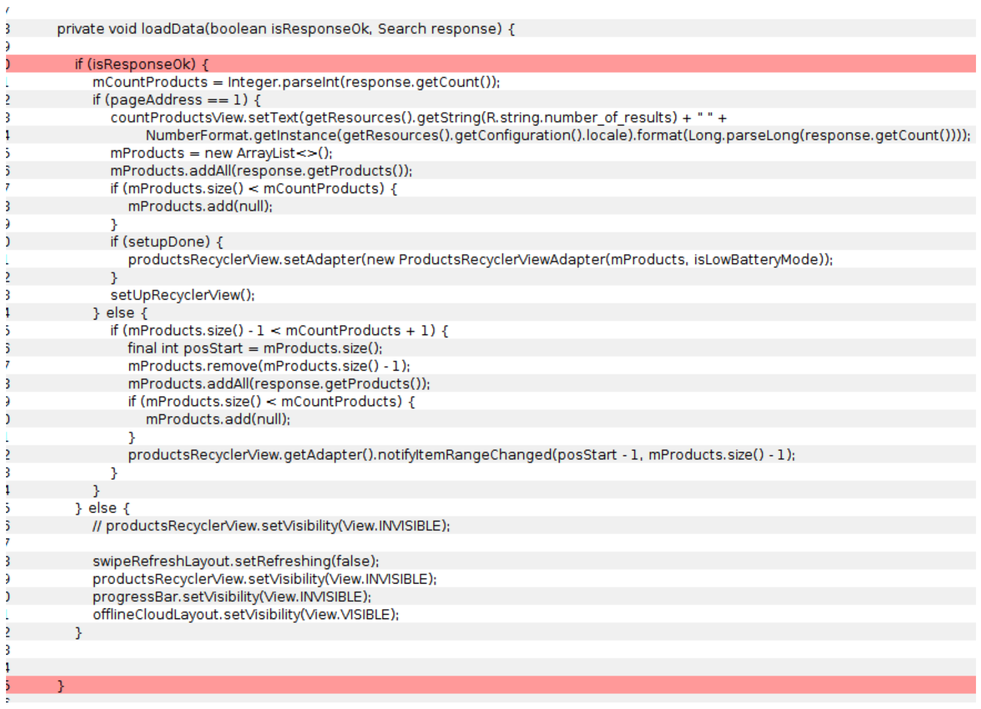
```
80        }
81
82        @Override
83        public View getView(int position, View convertView, ViewGroup parent) {
84            Holder holder;
85
86            if (convertView == null) {
87                holder = new Holder();
88
89                LayoutInflater layoutInflater =
90                        (LayoutInflater) mainActivityRef.get().getSystemService(Context.LAYOUT_INFLATER_SERVICE);
91                convertView = layoutInflater.inflate(R.layout.subscription_item, parent, false);
92                holder.feedTitle = (TextView) convertView.findViewById(R.id.txtvTitle);
93                holder.imageView = (ImageView) convertView.findViewById(R.id.imgvCover);
94                holder.count = (TriangleLabelView) convertView.findViewById(R.id.triangleCountView);
95
96
97                convertView.setTag(holder);
98            } else {
99                holder = (Holder) convertView.getTag();
100           }
101
102           if (position == getAddTilePosition()) {
103               holder.feedTitle.setText("{md-add 500%}\n\n" + mainActivityRef.get().getString(R.string.add_feed_label));
104                       holder.feedTitle.setVisibility(View.VISIBLE);
105               // prevent any accidental re-use of old values (not sure how that would happen...)
```

```
1st breakpoint hit at=== 86
Watch Variables || holder || added to sensitve variables
40 size == 37
At Line:86 Sensitive Variables: [holder]
At Line:87 Sensitive Variables: [holder]
At Line:89 Sensitive Variables: [holder]
At Line:90 Sensitive Variables: [holder]
At Line:91 Sensitive Variables: [holder]
At Line:92 Sensitive Variables: [holder]
At Line:93 Sensitive Variables: [holder]
At Line:94 Sensitive Variables: [holder]
At Line:97 Sensitive Variables: [holder, convertView]
Last breakpoint at 102
holder has touched: [Context, parent, R, convertView, layoutInflater, mainActivityRef]
At Line:98 Sensitive Variables: [holder, convertView]
```

Figure 8.7: Experiment-2 on 'AntennaPod' App

In experiment-2, '`holder`' variable is watch variable it touches many variables in the debugging process. They all are added the containing variables accessed by '`holder`'. '`convertView`' variable becomes sensitive on touching variable '`holder`' on line-97. Finally all variable touched by '`holder`' are printed in the end.

Open Food Facts



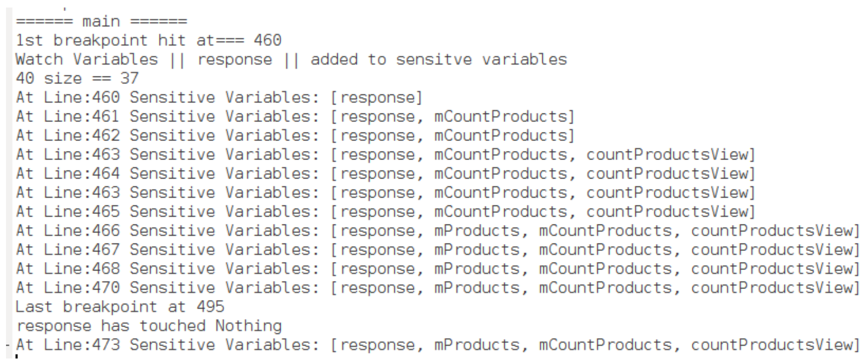
```
    private void loadData(boolean isResponseOk, Search response) {

        if (isResponseOk) {
            mCountProducts = Integer.parseInt(response.getCount());
            if (pageAddress == 1) {
                countProductsView.setText(getResources().getString(R.string.number_of_results) + " " +
                        NumberFormat.getInstance(getResources().getConfiguration().locale).format(Long.parseLong(response.getCount())));
                mProducts = new ArrayList<>();
                mProducts.addAll(response.getProducts());
                if (mProducts.size() < mCountProducts) {
                    mProducts.add(null);
                }
                if (setupDone) {
                    productsRecyclerView.setAdapter(new ProductsRecyclerViewAdapter(mProducts, isLowBatteryMode));
                }
                setUpRecyclerView();
            } else {
                if (mProducts.size() - 1 < mCountProducts + 1) {
                    final int posStart = mProducts.size();
                    mProducts.remove(mProducts.size() - 1);
                    mProducts.addAll(response.getProducts());
                    if (mProducts.size() < mCountProducts) {
                        mProducts.add(null);
                    }
                    productsRecyclerView.getAdapter().notifyItemRangeChanged(posStart - 1, mProducts.size() - 1);
                }
            }
        } else {
            // productsRecyclerView.setVisibility(View.INVISIBLE);

            swipeRefreshLayout.setRefreshing(false);
            productsRecyclerView.setVisibility(View.INVISIBLE);
            progressBar.setVisibility(View.INVISIBLE);
            offlineCloudLayout.setVisibility(View.VISIBLE);
        }


    }
```

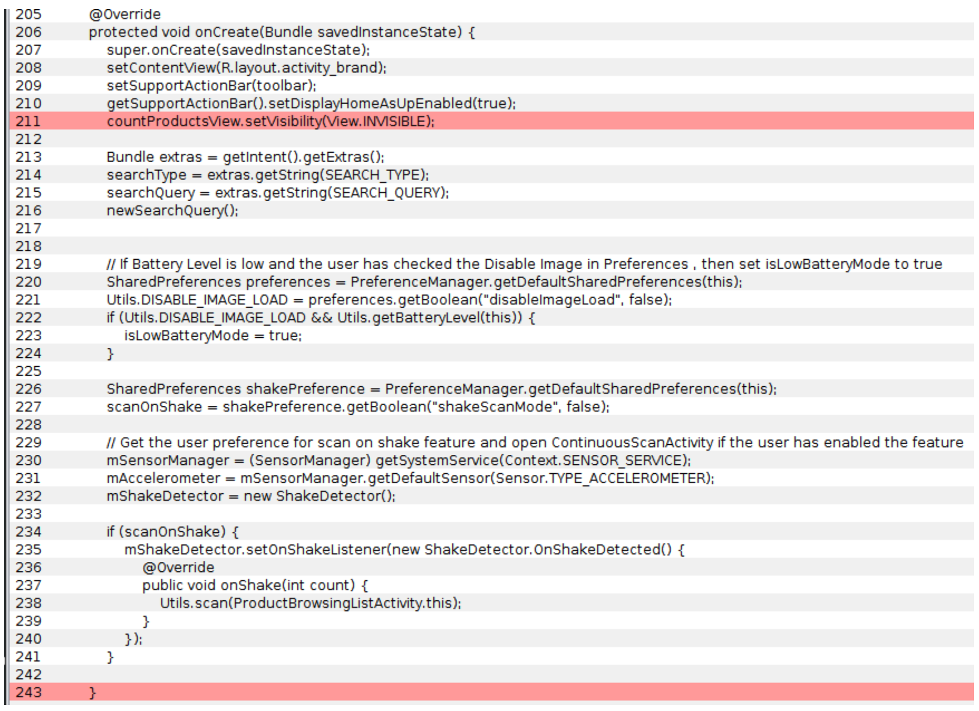
Figure 8.8: Experiment-1 on ‘Open Food Facts’ App : Source Code



```
====== main ======
1st breakpoint hit at=== 460
Watch Variables || response || added to sensitve variables
40 size == 37
At Line:460 Sensitive Variables: [response]
At Line:461 Sensitive Variables: [response, mCountProducts]
At Line:462 Sensitive Variables: [response, mCountProducts]
At Line:463 Sensitive Variables: [response, mCountProducts, countProductsView]
At Line:464 Sensitive Variables: [response, mCountProducts, countProductsView]
At Line:463 Sensitive Variables: [response, mCountProducts, countProductsView]
At Line:465 Sensitive Variables: [response, mCountProducts, countProductsView]
At Line:466 Sensitive Variables: [response, mProducts, mCountProducts, countProductsView]
At Line:467 Sensitive Variables: [response, mProducts, mCountProducts, countProductsView]
At Line:468 Sensitive Variables: [response, mProducts, mCountProducts, countProductsView]
At Line:470 Sensitive Variables: [response, mProducts, mCountProducts, countProductsView]
Last breakpoint at 495
response has touched Nothing
At Line:473 Sensitive Variables: [response, mProducts, mCountProducts, countProductsView]
```

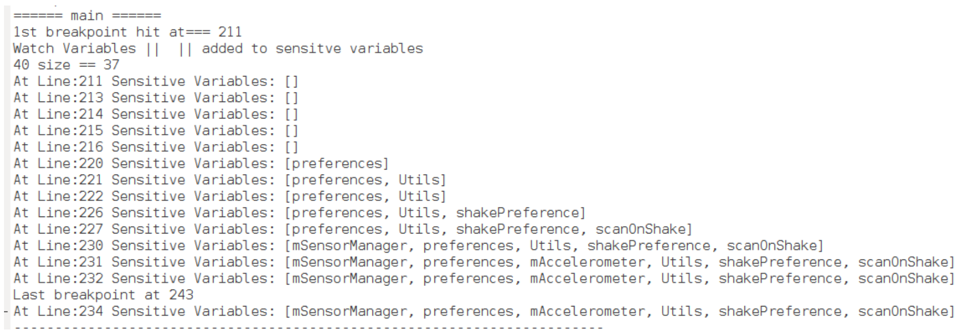
Figure 8.9: Experiment-1 on ‘Open Food Facts’ App : Debugger Output

In experiment-1, we see path followed by the running App and sensitive variables line by line. On line-461 variable ‘`mCountProducts`’ becomes sensitive on touching watch variable ‘`response`’. Similarly all other variables touching variable ‘`response`’ becomes sensitive.



```
205     @Override
206     protected void onCreate(Bundle savedInstanceState) {
207         super.onCreate(savedInstanceState);
208         setContentView(R.layout.activity_brand);
209         setSupportActionBar(toolbar);
210         getSupportActionBar().setDisplayHomeAsUpEnabled(true);
211         countProductsView.setVisibility(View.INVISIBLE);
212
213         Bundle extras = getIntent().getExtras();
214         searchType = extras.getString(SEARCH_TYPE);
215         searchQuery = extras.getString(SEARCH_QUERY);
216         newSearchQuery();
217
218
219         // If Battery Level is low and the user has checked the Disable Image in Preferences , then set isLowBatteryMode to true
220         SharedPreferences preferences = PreferenceManager.getDefaultSharedPreferences(this);
221         Utils.DISABLE_IMAGE_LOAD = preferences.getBoolean("disableImageLoad", false);
222         if (Utils.DISABLE_IMAGE_LOAD && Utils.getBatteryLevel(this)) {
223             isLowBatteryMode = true;
224         }
225
226         SharedPreferences shakePreference = PreferenceManager.getDefaultSharedPreferences(this);
227         scanOnShake = shakePreference.getBoolean("shakeScanMode", false);
228
229         // Get the user preference for scan on shake feature and open ContinuousScanActivity if the user has enabled the feature
230         mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
231         mAccelerometer = mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
232         mShakeDetector = new ShakeDetector();
233
234         if (scanOnShake) {
235             mShakeDetector.setOnShakeListener(new ShakeDetector.OnShakeDetected() {
236                 @Override
237                 public void onShake(int count) {
238                     Utils.scan(ProductBrowsingListActivity.this);
239                 }
240             });
241         }
242
243     }
```

Figure 8.10: Experiment-2 on 'Open Food Facts' App : Source Code

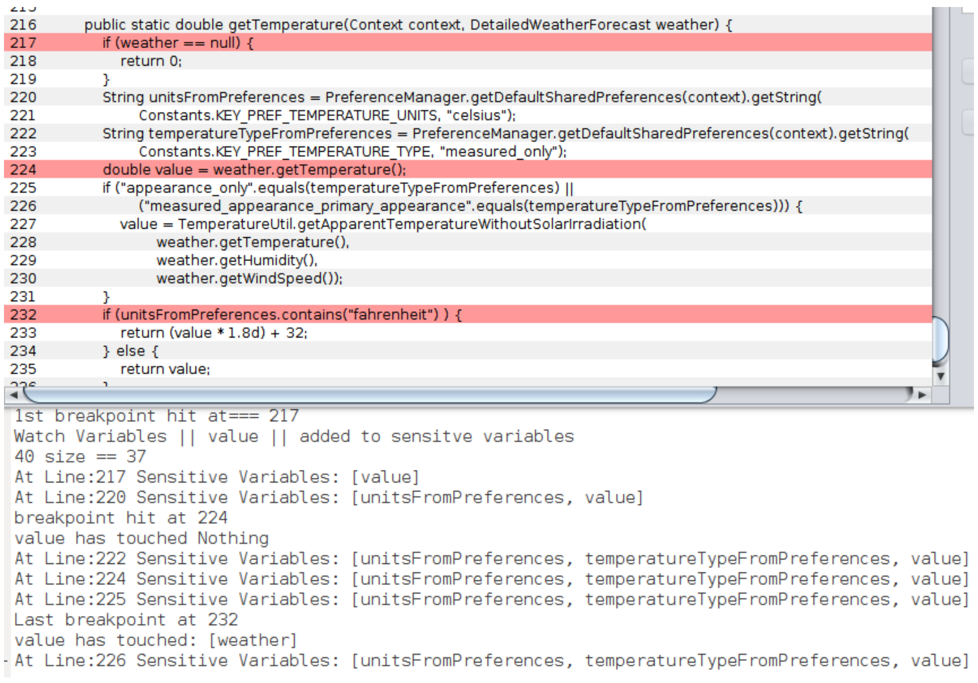


```
====== main ======
1st breakpoint hit at=== 211
Watch Variables ||  || added to sensitve variables
40 size == 37
At Line:211 Sensitive Variables: []
At Line:213 Sensitive Variables: []
At Line:214 Sensitive Variables: []
At Line:215 Sensitive Variables: []
At Line:216 Sensitive Variables: []
At Line:220 Sensitive Variables: [preferences]
At Line:221 Sensitive Variables: [preferences, Utils]
At Line:222 Sensitive Variables: [preferences, Utils]
At Line:226 Sensitive Variables: [preferences, Utils, shakePreference]
At Line:227 Sensitive Variables: [preferences, Utils, shakePreference, scanOnShake]
At Line:230 Sensitive Variables: [mSensorManager, preferences, Utils, shakePreference, scanOnShake]
At Line:231 Sensitive Variables: [mSensorManager, preferences, mAccelerometer, Utils, shakePreference, scanOnShake]
At Line:232 Sensitive Variables: [mSensorManager, preferences, mAccelerometer, Utils, shakePreference, scanOnShake]
Last breakpoint at 243
At Line:234 Sensitive Variables: [mSensorManager, preferences, mAccelerometer, Utils, shakePreference, scanOnShake]
------------------------------------------------------------------------
```

Figure 8.11: Experiment-2 on 'Open Food Facts' App : Debugger Output

In experiment-2, there is no watch variable provided. '`preferences`'variable becomes sensitive on line-220 and it causes '`Utils`' to becomes sensitive on line number-221. At the end some variables touches '`SensorManager`' service and becomes sensitive.

Your local weather



```
216     public static double getTemperature(Context context, DetailedWeatherForecast weather) {
217         if (weather == null) {
218             return 0;
219         }
220         String unitsFromPreferences = PreferenceManager.getDefaultSharedPreferences(context).getString(
221                 Constants.KEY_PREF_TEMPERATURE_UNITS, "celsius");
222         String temperatureTypeFromPreferences = PreferenceManager.getDefaultSharedPreferences(context).getString(
223                 Constants.KEY_PREF_TEMPERATURE_TYPE, "measured_only");
224         double value = weather.getTemperature();
225         if ("appearance_only".equals(temperatureTypeFromPreferences) ||
226                 ("measured_appearance_primary_appearance".equals(temperatureTypeFromPreferences))) {
227             value = TemperatureUtil.getApparentTemperatureWithoutSolarIrradiation(
228                     weather.getTemperature(),
229                     weather.getHumidity(),
230                     weather.getWindSpeed());
231         }
232         if (unitsFromPreferences.contains("fahrenheit") ) {
233             return (value * 1.8d) + 32;
234         } else {
235             return value;
```

```
1st breakpoint hit at=== 217
Watch Variables || value || added to sensitve variables
40 size == 37
At Line:217 Sensitive Variables: [value]
At Line:220 Sensitive Variables: [unitsFromPreferences, value]
breakpoint hit at 224
value has touched Nothing
At Line:222 Sensitive Variables: [unitsFromPreferences, temperatureTypeFromPreferences, value]
At Line:224 Sensitive Variables: [unitsFromPreferences, temperatureTypeFromPreferences, value]
At Line:225 Sensitive Variables: [unitsFromPreferences, temperatureTypeFromPreferences, value]
Last breakpoint at 232
value has touched: [weather]
At Line:226 Sensitive Variables: [unitsFromPreferences, temperatureTypeFromPreferences, value]
```

Figure 8.12: Experiment on ‘Your local weather’ App

In experiment on ‘Your local weather’ App we had set three breakpoints and provided ‘`value`’ as watch variable. Both variables ‘`unitsFromPreferences`’ and ‘`temperatureTypeFromPref` become sensitive on storing sensitive informations. Watch variable ‘`value`’ touches variable ‘`weather`’ after second breakpoint. At the end of debugging it is printed that ‘`value`’ had touched ‘`weather`’ variable.

Chapter 9

Conclusion and Future Work

Conclusion

Java Debug Interface (JDI) [9] is an API useful for programmers writing their custom Java debuggers. JDI has been used today in most of the debuggers in numerous IDEs. JDI allowed us to control and monitor a running debugee process. By using JDI's ability to monitor a running program we inspect its security at runtime. We observed that JDI alone can not be used for our task because it can not track local variable modification and access in the program and it becomes very time consuming to find variables passed to method calls, but JDI could be helpful with JavaParser library [16]. In our current approach to develop SecDroidBug, JDI part provides dynamic monitoring while JavaParser part helps to understand flow of data line by line. Although there are multiple dynamic analysis tools available for analysing security inside Android app, but SecDroidBug can give developers and testers an ability to inspect code between two lines, in source code, for data leakage. So our tool can work inline in the development phase like traditional program debuggers but with functionality of testing the security. Our debugger can check security inside methods and can report potential data leakage to the developers.

Future Work

SecDroidBug can be used inside a method for security testing, in some cases there may be a case that some methods called in the scope of testing may call some other methods which, in turn, may call sensitive source or sink calls. Also, some methods can pass some variables with pass-by-reference where those variable may get labeled as sensitive inside the called method. To cover these cases we needed to get inside those methods and apply debugging inside them. It required `MethodEntry` events to cover more type of method calls, as we have currently used them to catch sensitive method calls, and increasing their scope to cover more types of method calls may decrease the performance of debugger. We found that JavaParser library [16] could be used to get sensitive method calls.

In app's source code if there are more than one variables with same names it requires to resolve their types. We used the same JavaParser library to differentiate between such variables. By using JavaSymbolResolver [16] sub-library in JavaParser library it is possible to resolve method calls and variables. When we used this approach although it was useful but it failed to resolve some variables and methods. We made efforts to find alternate ways to solve these problems and allow others to continue the efforts, but our solutions did not work.

Because of these two problems we could not add functionality to our debugger to debug inside called methods, as not resolving variables with same name can cause abnormal results and not resolving method calls may decrease performance of debugger if we allow our debugger to debug called methods. We hope that by using different SymbolResolver to get type of variables and methods, or using different parser for parsing Java source code altogether may solve those two problems and we can see what is going inside called methods.

Bibliography

[1] 2018. Android Monkey Tool. `https://developer.android.com/studio/test/monkey`.

[2] 2018. ApiCallsWithDeb. `https://github.com/gsy18/ApiCallsWithDeb`.

[3] 2018. Java Platform Debugger Architecture Diagram. `https://premaseem.files.wordpress.com/2012/12/jpda.png`.

[4] Asif Azam Ali and R. K. Shyamasundar. 2017. *SecFlowDroid: A Tool for Privacy Analysis of Android Apps using RWFM Model*. Master's thesis, IIT Bombay.

[5] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2013. Susi: A tool for the fully automated classification and categorization of android sources and sinks. *University of Darmstadt, Tech. Rep. TUDCS-2013-0114* .

[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49(6):259–269.

[7] Eric Bodden et al. 2018. Soot - A framework for analyzing and transforming Java and Android applications. `https://sable.github.io/soot/`.

[8] Konrad Jamrozik and Andreas Zeller. 2016. DroidMate: A robust and extensible test generator for android. In *Mobile Software Engineering and Systems (MOBILESoft), 2016 IEEE/ACM International Conference on*. IEEE, pages 293–294.

[9] TM Java. 1999. Java Platform Debugger Architecture. `https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/architecture.html`.

[10] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: a lightweight UI-Guided test input generator for android. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*. IEEE, pages 23–26.

[11] Jedidiah McClurg, Jonathan Friedman, and William Ng. 2013. Android privacy leak detection via dynamic taint analysis. *EECS* 450:2013.

[12] Diego Torres Milano. 2018. AndroidViewClient. `https://github.com/dtmilano/AndroidViewClient`.

[13] Oracle. 2018. Java Platform Debugger Architecture (JPDA) example. `https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/trace.html`.

[14] Sarker T Ahmed Rumee and Donggang Liu. 2015. DroidTest: Testing Android applications for leakage of private information. In *Information Security*, Springer, pages 341–353.

[15] Lovely Sinha, Shweta Bhandari, Parvez Faruki, Manoj Singh Gaur, Vijay Laxmi, and Mauro Conti. 2016. Flowmine: Android app analysis via data flow. In *Consumer Communications & Networking Conference (CCNC), 2016 13th IEEE Annual*. IEEE, pages 435–441.

[16] Danny van Bruggen, Federico Tomassetti, and et al. 2018. JavaParser Library. `http://javaparser.org/` and `https://github.com/javaparser/javaparser`.

[17] R Winsniewski. 2012. Android–apktool: A tool for reverse engineering android APK files. `https://ibotpeaches.github.io/Apktool/`.

[18] Kun Yang. 2018. APIMonitor. `https://code.google.com/archive/p/droidbox/wikis/APIMonitor.wiki`.