Deep Understanding on Taint Analysis: a Survey on Existing Taint Analysis Techniques

Course Project Report

Aw, Snap!: Weijia Sun, Xinyu Lyu, Mengmei Ye
Rutgers University

ABSTRACT

This is a course project report for the course 16:332:579 - Malware Analysis and Reverse Engineering in Spring 2019 at Department of Electrical and Computer Engineering, Rutgers University.

Android phones have been dominating the mobile device market. However, there are many security and privacy concerns for Android users. To prevent from the secret data leakage, researchers and engineers have been adopted taint analysis to detect the illegal sensitive information flow. The techniques behind the taint analysis mainly originate from the software engineering community. In the meantime, taint analysis has been widely used in the security domain to defend against many threat models in multiple applications. Considering the background and the applications of taint analysis, it is significant to have a comprehensive study on the existing taint analysis work, so that the researchers could have views from both of the domains.

This course report presents a survey on the existing taint analysis techniques to perform a comprehensive study by explaining the software engineering techniques and security usage, as well as the design/implementation challenges in the existing work. In addition, in the experiment section, we further evaluate the existing taint analysis tools and Android-app analysis tools.

KEYWORDS

Malware Analysis, Software Engineering, Software Security, Mobile Security

ACM Reference Format:
Aw, Snap!: Weijia Sun, Xinyu Lyu, Mengmei Ye. 2019. Deep Understanding on Taint Analysis: a Survey on Existing Taint Analysis Techniques: Course Project Report. In *Proceedings of Malware Analysis and Reverse Engineering (16:332:579).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Mobile devices have been ubiquitous in the recent decade. While app developers design many mobile applications and publish them into app stores, mobile users are able to download these applications from the app stores. Although the mobile devices and applications significantly improve life convenience and work efficiency, they could compromise user security and privacy.

To defend against the security and privacy threats, taint analysis has been widely applied to enhance the application/system security. Researchers and engineers on the security community and the software engineering community have been actively working on taint analysis for many years. Considering the significance of taint analysis in both of the community, we collect the recent research work and do a survey report on the existing taint analysis techniques.

Particularly, we are going to target on the following three research questions (RQs) in this report.

- RQ1: what are the software-engineering concepts behind taint analysis?
- RQ2: what are the advantages and limitations of taint analysis?
- RQ3: how efficient are the taint analysis techniques for Android applications?

To answer the questions, we will be studying on recent research work published in the security or software-engineering conferences. Also, we will be empirically evaluating the efficiency of the open-source taint analysis tools about how they successfully analyze and detect information leakage in Android devices. The reason we target on Android security analysis is that the Android market share is more than 80% in the whole mobile market[24]. It is significantly important to address the security and privacy concerns in Android devices.

2 MOTIVATION

There has been much excellent research work published in both of the security community and the software engineering community. By reviewing the research papers published in recent years and evaluating the open-source tools for taint analysis, this comprehensive study could help researchers obtain the summary of recent work related to taint analysis as well as help us gain technical experience on the security and software engineering domains. In detail, the motivation for the project is listed in the following.

Contribution to the software-engineering domain. The taint analysis technology originates from the software engineering domain. The survey could be helpful for software-engineering researchers to have a comprehensive view regarding how the taint analysis technology was developed and improved in recent years.

Contribution to the security domain. Based on the RQs in our report, security researchers could obtain information on the advantages and disadvantages of taint analysis, which could help them further protect against the security vulnerabilities in different systems/applications.

Our personal interests and future career development. All of our group members have both backgrounds on software engineering and security. Working on the taint analysis is a great fit for us, and we have a strong passion for gaining a deep understanding of taint analysis. The survey on taint analysis would be a wonderful opportunity for us to enhance technical skills related to software engineering and security for our future career development.

16:332:579, Spring 2019, Department of ECE Rutgers University
2019. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

3 RQ1: SOFTWARE ENGINEERING TECHNIQUES BEHIND TAINT ANALYSIS

In general speaking, the workflow of taint analysis could be the following steps.

(1) Taint an object (e.g., variable) as a source. Typically, the tainted object is or contains sensitive information.
(2) Analyze the program with an analysis tool, and generate the analysis log.
(3) If the tainted object eventually goes to the insecure or unauthorized destination, the analysis tool identifies that the behavior of the application is malicious.

Typically, the analysis techniques contain static analysis, dynamic analysis, and symbolic execution analysis. Static analysis mainly targets on the offline analysis of program structure and the source code without actually executing the program [23]. Different from static analysis, dynamic analysis requires to execute the target program to detect the abnormal behaviors.

To demonstrate how the static and dynamic taint analysis work, we show a sample code in the following.

```c
1  #include <stdio.h>
2
3  int main()
4  {
5      int a = 5, b = 0, flag = 1;
6      if (flag)
7          b = a - 1;
8      else
9          b = a + 1;
10     return 0;
11 }
```

Assume that the variable *a* is sensitive data. Both the static and dynamic taint analysis tools will treat the variable *b* as sensitive data as well, since the variable *b* contains the value of *a*. However, the static taint analysis tool does not execute the program and only analyzes the structure of the program. Therefore, the tool considers all the possible dataflow paths, and lines 7 and 9 are treated as sensitive computations. Different from the static taint analysis tool, the dynamic taint analysis tool executes the program. In this example, the variable $flag$ is equal to 1, so it does not trace the dataflow of the computations in the *else* statement. In other words, only line 7 is tainted.

Static and dynamic taint analysis could apply to different applications, which depends on the usage and resource/performance constraints. For example, static taint analysis is applicable for the applications that do not have a complicated program structure, so that the tool could analyze the program precisely and cover the necessary data flows. For the dynamic taint analysis, it is feasible for the applications that do not have too many complicated test suites, so that the tool will not miss the important data flows. In addition, depending on different programming languages and different applications, the taint analysis requires additional technology to precisely analyze the programs. For example, for the Android applications that are developed by the Java language, there are multiple levels of sensitivities to analyze the programs [21][3].

- context-level sensitivity, which depends on the context of function calls. In other words, by targeting one function call, the analysis tool will trace all the subfunction calls as well.
- object-level sensitivity, which depends on the object allocation. In other words, by targeting a/an variable/object, the analysis tool will trace all the computations related to the variable/object.
- field-level sensitivity, which depends on a certain field under a class. Different from the object-level sensitivity, the tool with the field-level sensitivity analyzes the program based on the related fields instead of the objects.
- flow-level sensitivity, which depends on the flow or the order of the computations. For example, when there are the computations a=1; b=2; c=a+b; a=2, the tool will treat the value of c is 3 instead of 3 and 4.
- path-level sensitivity, which depends on the feasible path that the program could execute. For example, in the *if-else* statement, based on different conditions, the feasible path could be the *if* path only or the *else* path only.

4 RQ2: ADVANTAGES AND LIMITATIONS OF TAINT ANALYSIS

4.1 Applications and Usage

Based on our study, taint analysis has been leveraged in many different applications to defend against attacks, such as mobile and web malware. The web applications are vulnerable to injection attacks or cross-site scripting attacks. There has been much work conducting taint analysis to defend against the web attacks, such as [16][25]. In mobile security, the taint analysis mainly targets on whether the sensitive information (e.g., passwords, text messages, and location data) leaks to non-trusted party.

Because this course report mainly works on the Android app analysis, we are discussing more details on this scope in the following.

Android security has been actively discussed in the security community. Many Android analysis tools apply static, or dynamic analysis to detect sensitive information leakage. Considering that sometimes the Android APK source code is not feasible, TaintDroid[10] leverages dynamic taint analysis with different levels of taint granularity by executing the apps and automatically tainting the sensitive sources.

However, dynamic taint analysis requires additional time consumption due to the app execution. Therefore, Flowdroid [3], as a static taint analysis tool, analyzes the bytecode of the target program with the context, flow, and object sensitivity. However, Flowdroid does not perform very well in the time consumption and memory usage, Huang et al. proposed DFlow and DroidInfer to achieve low performance and high precision with scalability [15].

Furthermore, due to the imperfection of the basic analysis technology, researchers have been targeting the sensitivity types in the analysis tools, such as context-based, flow-based, and type-based [3][15] in order to strengthen the effectiveness and efficiency of the taint analysis tools.

4.2 Limitations and Challenges

Trade-off between precision and performance of analysis. A great trade-off between precision and performance of analysis could effectively avoid over-tainting or under-tainting problems [22][23]. Specifically, although static analysis could analyze the program without causing additional app execution time, it could encounter over-taint/approximation problems. For example, an inappropriate static analysis could enroll in the infeasible dataflow, which is a waste of time and memory. On the other hand, although dynamic analysis could achieve high precision with appropriate test suites, it requires much time consumption to execute the apps in order to conduct the analysis. For the trade-off between time consumption and precision of analysis, a feasible solution could be adding additional work based on granularity and sensitivity as we mentioned in the previous section.

Test case generation and code coverage. Compared to static analysis, the dynamic analysis could effectively analyze the program based on the test suites. However, how to generate a suitable test suite is a significant but difficult topic. A great test suite could cover the most important portion of the program, namely high code coverage, so that the dynamic taint analysis tool could detect bugs with high efficiency. To mitigate this problem, there are many methods to generate suitable test suites, such as random fuzzing approach and symbolic testing. Recently, Wong et al. proposed a framework, namely IntelliDroid[27], to generate appropriate test suites with high code coverage, so that dynamic analysis could run the generated test suites in low-performance overhead.

Scalability. As we discussed in the previous subsections, researchers have spent great efforts on improving the precision of taint analysis. However, while achieving the precise taint analysis, researchers could encounter another problem, namely scalability. In details, after improving the precision, the time consumption and memory space usage heavily increase at the same time. When a program has a relatively large size, the taint analysis could spend a long time to finish the entire process and require a large memory space to trace the libraries and store the logs [15]. Therefore, to achieve scalability, it requires additional optimization work (e.g., using genetic algorithms [18]) or applies different analysis techniques (e.g., using data flow analysis instead of points-to analysis [15] or tracing sensitive data based on the lifespan of variables [3]).

5 RQ3: EXPERIMENTAL SETUP AND EVALUATION FOR TAINT ANALYSIS TOOLS

In this section, we show the setup process and discuss the analysis tools that we execute.

5.1 Taintgrind [17]

In order to have a better understanding before we actually work on the Android analysis tools, we first work on a dynamic taint analysis tool, namely Taintgrind[17], which uses the Valgrind[8] dynamic programming tool. As shown in the following source code example, we execute it with Taintgrind instructions provided by the authors. In the example, we assume the variable *a* in the *main* function contains sensitive data and further taint it. After the taint analysis, the variable *s* is treated as sensitive data as well, because the *get_sign* function assigns the value of *a* to the variable *s*.

```
#include "taintgrind.h"
int get_sign(int x, int y) {
    int s;
    if (x > 0)  s = x + y;
    else   s = x + 1;
    return s;
}
void main(int argc, char **argv){
    // Turns on printing
    TNT_START_PRINT();
    int a = 1000;
    int b = 2000;
    // Defines int a as tainted
    TNT_MAKE_MEM_TAINTED_NAMED(&a,4, "myint");
    int s = get_sign(a, b);
    // Turns off printing
    TNT_STOP_PRINT();
}
```

Furthermore, we extract the sensitive dataflow of this example from the log that is generated by Taintgrind, which is shown in the following. Based on the dataflow, it clearly presents the location of the sensitive computation and the flow direction of the tainted data.

```
0x4007C8: get_sign (sign-ds.c:4) | t15_8648 = LOAD I32
     t12_11254 | 0x3e8 | 0xffffffff | --22310-- warning:
     evaluate_Dwarf3_Expr: unhandled DW_OP_ 0xf2
t15_8648 <- x
0x4007D0: get_sign (sign-ds.c:4) | STORE t27_2679 =
     t29_2209 | 0xbb8 | 0xffffffff | --22310-- warning:
     evaluate_Dwarf3_Expr: unhandled DW_OP_ 0xf2
s <- t29_2209
```

5.2 Infer [7]

The Infer tool is developed by Facebook, and it supports to analyze programs with multiple languages including Android, Java, C, C++, and iOS/Objective-C. Although it is a general static analysis tool without the taint techniques, it is still worth to evaluate due to the following reasons.

- This is a reliable and prevalent Android analysis tool from an industry.
- The taint analysis techniques are under developed, and the taint-embedded Infer will be released soon.
- Even if our report primarily targets on taint analysis technology, the static analysis that the Infer tool applies could be the base of the taint analysis. Working on this tool is helpful for us to illustrate the techniques that the analysis tool uses.

To deploy the Infer tool in our PC, we first install it based on the instructions in [12]. Then, based on the given hello-world examples in [13], we execute two types of applications, namely a Java program and an Android program. The reason we execute the Java program is that the Android program is written in Java.

The following source code shows the Java program.

```
1 class Hello_Java {
2   String test() {
3     String s = null;
4     return s.substring(3);
5   }
6 }
```

In the Java program, we notice that the return instruction attempts to output a substring of *s*. However, in line 3, the string *s* is equal to null. The Infer tool is able to detect this bug, which is reported in the following log file. In the log file, the first line indicates the command that we input, and the content in the follows indicates the log from the Infer tool. In the log, it clearly shows that there is a null-dereference issue detected.

```
$ infer run -- javac Hello.java

Hello.java:4: error: NULL_DEREFERENCE
  object s last assigned on line 3 could be null and is
      dereferenced at line 4.
  2.      String test() {
  3.        String s = null;
  4. >      return s.substring(3);
  5.      }
  6.    }
```

In order to fix this problem, we can provide an input value to the string *s*, as shown in the following source code. In the source code, we set the string *s* is equal to "Hello World!".

```
class Hello_Java {
  String test() {
    String s = "Hello World!";
    return s.substring(3);
  }
}
```

By executing the source code by the Infer tool, we notice that there is no issue in the current source code, as shown in the following log file.

```
$ infer run -- javac fixed_Hello.java

  No issues found
```

Furthermore, we intentionally change line 4 in the source code to be “return s.substring(20);". It is supposed to throw an exception, namely *java.lang.StringIndexOutOfBoundsException*, since the substring is out of the range. However, the Infer tool still detects that there is no issue. We guess it is because the Infer tool uses static analysis without executing the code, and it is not able to detect bugs caused at runtime.

Furthermore, we execute an Android app in the Infer tool, and the source code of this app is in [11] provided by Facebook. The log generated by Infer for this example is shown in the following. The log indicates that there are 3 issues detected, namely 2 null-dereference issues and 1 resource-leakage issue. Specifically, the resource-leakage issue is generated because the program conducts a file access but it does not release the output stream after the file access.

```
$ infer run -- ./gradlew build

app/src/main/java/infer/inferandroidexample/MainActivity.
    java:48: error: RESOURCE_LEAK
  resource of type java.io.FileOutputStream acquired to
      fis by call to FileOutputStream(...) at line 45
      is not released after line 48.
**Note**: potential exception at line 46
  46.           fis.write(arr);
  47.           fis.close();
  48. >       } catch (IOException e) {
  49.           // Deal with exception
  50.         }
```

In summary about the Infer tool, it works as expected based on the information listed in [7]. However, we argue that there are the following drawbacks when it analyzes the Android apps.

(1) Analyzing an Android app takes a long time. In the example [11], where the program size is small, the Infer tool takes 16.557s to complete the analysis. If the program increases to a large size (especially for a complicated Android app), the time consumption would be unacceptable.
(2) The Infer tool specifically targets on the design analysis for the software developers. It is able to check if there are any bugs existed. However, it is not able to check if there are illegal data flow in the app. In other words, it is a great tool to check the quality of the app, but it is not suitable to verify the security.

5.3 Flowdroid [3]

Furthermore, we implement Flowdroid, which is an advanced static taint analysis tool. In addition, to analyze the sensitive information leakage by static taint techniques, Flowdroid is able to achieve superior recall and precision by being aware of context, flow, and lifecycle in the app. In the experiments, we apply an example from DroidBench that is an open-source benchmark to evaluate the Android-target taint analysis tools. The example is called SimpleAliasing1 [1], which shuffles the sensitive data by different alias in the program. The Flowdroid tool could further detect if there are sensitive data leakage issues based on the pre-defined sources and sinks, namely [2] in our experiments.

By executing the example in Flowdroid, we notice that the program detects 1 leakage issue, as shown in the following simplified log generated by Flowdroid. In addition, we measure the time consumption by executing the example. Flowdroid consumes 9.91s to complete the analysis.

```
time java -jar soot-infoflow-cmd-jar-with-dependencies.
     jar -a FlowDroid/Aliasing/SimpleAliasing1.apk -p
     Android/sdk/platforms/android-28/android.jar -s
     SourcesAndSinks.txt

[main] INFO soot.jimple.infoflow.android.
     SetupApplication$InPlaceInfoflow - Data flow solver
     took 0 seconds. Maximum memory consumption: 42 MB
[main] INFO soot.jimple.infoflow.android.SetupApplication
      - Found 1 leaks
```

5.4 Amandroid [26]

Furthermore, we execute Amandroid, which is a static analysis tool with the flow and context sensitivities. The advantages of Amandroid contain:

- Amandroid is able to handle the sensitive dataflow across the different applications in the Android device. In other words, Amandroid treats the inter-component control and data flows as regular method calls.
- Amandroid could evaluate the real Android applications in addition to the micro-testbenches.

To make sure a fair comparison between Flowdroid and Amandroid, we also conduct the analysis experiment for the SimpleAliasing1 testbench. The following information is the partial log generated from Amandroid. We notice that the Amandroid tool also reports one information leakage as the Flowdroid. However, it takes much longer time to complete the analysis. The total analysis time is 28.08s.

```
Application Name: SimpleAliasing1.apk
    TaintPath:
```

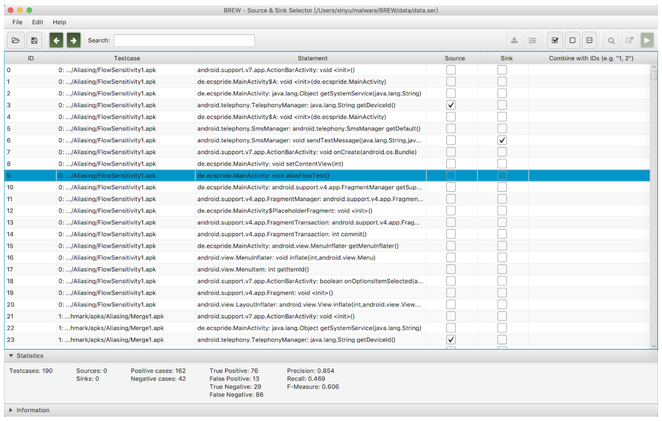
```
 Source: <Descriptors: api_source: #L0975d6.  call
temp:=  getDeviceId (v13) @signature  Landroid/
telephony/TelephonyManager;.getDeviceId:()Ljava/lang
/String;  @kind virtual;>
 Sink: <Descriptors: api_sink: #L097618.  call
sendTextMessage (v0, v1, v2, v3, v4, v5) @signature
 Landroid/telephony/SmsManager;.sendTextMessage:(
Ljava/lang/String;Ljava/lang/String;Ljava/lang/
String;Landroid/app/PendingIntent;Landroid/app/
PendingIntent;)V  @kind virtual;>
 Types:
 The path consists of the following edges ("->").
The nodes have the context information (p1 to pn
means which parameter). The source is at the top :
   List(#L0975d6.  call temp:=  getDeviceId (v13)
@signature  Landroid/telephony/TelephonyManager;.
getDeviceId:()Ljava/lang/String;  @kind virtual;, #
L097618.  call  sendTextMessage (v0, v1, v2, v3, v4,
 v5) @signature  Landroid/telephony/SmsManager;.
sendTextMessage:(Ljava/lang/String;Ljava/lang/String
;Ljava/lang/String;Landroid/app/PendingIntent;
Landroid/app/PendingIntent;)V  @kind virtual;)
```

5.5 ReproDroid [20]

In addition, we find that there is a research work, namely ReproDroid [20], that reproduces and evaluates the existing static analysis tools. They develop an evaluation tool, namely BREW, to analyze the tool. The workflow of BREW is shown in Figure 2. First, after we download the Brew tool and prepare for the configuration files, we load the benchmark (e.g., Droidbench [1]) into BREW. Then, we elect the analysis tool and further configure the XML file, so that BREW starts to analyze the tool. The tools that we execute in BREW includes Flowdroid [3], Amandroid [26], Droidsafe[14], IccTa[19], Dialdroid[6], and Didfail [9]. Figure 1 shows the interface of BREW after analyzing the Amandroid tool. Based on the results of Amandroid in the figure, we notice that there are 76 true positive cases and 13 false positive cases, where the Amandroid tool successfully detects the information flow leakage for 89 out of 204 testbenches. However, there are 29 true negative cases and 86 false negative cases, where Amandroid detects 115 testbenches wrong. The precision of Amandroid is 85.4%. For our total results for each analysis tool, please check BREW/BREW results.docx in [5].

The primary contribution of the ReproDroid work is that ReproDroid creates a standardized and machine-readable format, namely AQL-Answer, to display the data flows. By observing the dataflow results generated by Flowdroid and Amandroid in Section 5.3 and 5.4, the formats are completely different. By applying AQL-Answer, BREW is able to analyze different Android tools and compare the results automatically.

Although the BREW tool is an excellent tool, ReproDroid work does not elaborate too much about how they generate the ground truth for the fair comparison. In other words, they do not discuss why the proposed method is able to achieve the "unbias" comparison for every tool with different source and destination definitions. Also, they do not elaborate on how BREW identifies the critical information leakage automatically. It seems that the tool is not fully automatic and still requires some manual work.


BREW - Source & Sink Selector (/Users/xinyu/malware/BREW/data/data.ser)
File
Edit
Help
Search:

ID	Testcase	Statement	Source	Sink	Combine with IDs (e.g. "1, 2")
0	0: .../Aliasing/FlowSensitivity1.apk	android.support.v7.app.ActionBarActivity: void <init>()	☐	☐	
1	0: .../Aliasing/FlowSensitivity1.apk	de.ecspride.MainActivity$A: void <init>(de.ecspride.MainActivity)	☐	☐	
2	0: .../Aliasing/FlowSensitivity1.apk	de.ecspride.MainActivity: java.lang.Object getSystemService(java.lang.String)	☐	☐	
3	0: .../Aliasing/FlowSensitivity1.apk	android.telephony.TelephonyManager: java.lang.String getDeviceId()	✓	☐	
4	0: .../Aliasing/FlowSensitivity1.apk	de.ecspride.MainActivity$A: void <init>(de.ecspride.MainActivity)	☐	☐	
5	0: .../Aliasing/FlowSensitivity1.apk	android.telephony.SmsManager: android.telephony.SmsManager getDefault()	☐	☐	
6	0: .../Aliasing/FlowSensitivity1.apk	android.telephony.SmsManager: void sendTextMessage(java.lang.String,jav...	☐	✓	
7	0: .../Aliasing/FlowSensitivity1.apk	android.support.v7.app.ActionBarActivity: void onCreate(android.os.Bundle)	☐	☐	
8	0: .../Aliasing/FlowSensitivity1.apk	de.ecspride.MainActivity: void setContentView(int)	☐	☐	
9	0: .../Aliasing/FlowSensitivity1.apk	de.ecspride.MainActivity: void aliasFlowTest()	☐	☐	
10	0: .../Aliasing/FlowSensitivity1.apk	de.ecspride.MainActivity: android.support.v4.app.FragmentManager getSup...	☐	☐	
11	0: .../Aliasing/FlowSensitivity1.apk	android.support.v4.app.FragmentManager: android.support.v4.app.Fragmen...	☐	☐	
12	0: .../Aliasing/FlowSensitivity1.apk	de.ecspride.MainActivity$PlaceholderFragment: void <init>()	☐	☐	
13	0: .../Aliasing/FlowSensitivity1.apk	android.support.v4.app.FragmentTransaction: android.support.v4.app.Frag...	☐	☐	
14	0: .../Aliasing/FlowSensitivity1.apk	android.support.v4.app.FragmentTransaction: int commit()	☐	☐	
15	0: .../Aliasing/FlowSensitivity1.apk	de.ecspride.MainActivity: android.view.MenuInflater getMenuInflater()	☐	☐	
16	0: .../Aliasing/FlowSensitivity1.apk	android.view.MenuInflater: void inflate(int,android.view.Menu)	☐	☐	
17	0: .../Aliasing/FlowSensitivity1.apk	android.view.MenuItem: int getItemId()	☐	☐	
18	0: .../Aliasing/FlowSensitivity1.apk	android.support.v7.app.ActionBarActivity: boolean onOptionsItemSelected(a...	☐	☐	
19	0: .../Aliasing/FlowSensitivity1.apk	android.support.v4.app.Fragment: void <init>()	☐	☐	
20	0: .../Aliasing/FlowSensitivity1.apk	android.view.LayoutInflater: android.view.View inflate(int,android.view.View...	☐	☐	
21	1: ...hmark/apks/Aliasing/Merge1.apk	android.support.v7.app.ActionBarActivity: void <init>()	☐	☐	
22	1: ...hmark/apks/Aliasing/Merge1.apk	de.ecspride.MainActivity: java.lang.Object getSystemService(java.lang.String)	☐	☐	
23	1: ...hmark/apks/Aliasing/Merge1.apk	android.telephony.TelephonyManager: java.lang.String getDeviceId()	✓	☐	

Statistics

```
Testcases: 190    Sources: 0    Positive cases: 162    True Positive: 76     Precision: 0.854
                  Sinks: 0      Negative cases: 42     False Positive: 13    Recall: 0.469
                                                       True Negative: 29     F-Measure: 0.606
                                                       False Negative: 86
```

Information

Figure 1: Analysis Results of Amandroid

6 CONCLUSION

In this course report, we worked on three RQs for the taint analysis techniques. First, we elaborated the software-engineering concepts behind taint analysis. Second, we discussed the advantages and limitations of taint analysis. Third, we evaluated multiple taint analysis tools for Android applications.

For more information about our experimental setup and evaluation results, please check the following links, where we store the source code, test programs, and experimental results, as well as recorded demos in the Google Drive. Please use your Rutgers account to have the permission to access.

For the source code, test programs, and experimental results for each analysis tool, please check [5].

For the recorded demos for the analysis tools, please check [4].

REFERENCES

[1] Steven Arzt. [n. d.]. Droidbench Source. https://uni-paderborn.sciebo.de/s/ZmlRvtzI6pVYHVP/download?path=%2Fbenchmarks&files=DroidBench30.zip
[2] Steven Arzt. [n. d.]. SourcesAndSinks.txt. https://github.com/secure-software-engineering/FlowDroid/blob/master/soot-infoflow-android/SourcesAndSinks.txt
[3] 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. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, 259–269.
[4] Snap! Aw. [n. d.]. Demonstrations. https://drive.google.com/open?id=1bneHe-Y_RKEoq2suKWeLc4387HHOrpyd
[5] Snap! Aw. [n. d.]. Experiments. https://drive.google.com/open?id=14fU7fg-1E3qGScsAOBRUE0ivvUXtKfAj
[6] Amiangshu Bosu, Fang Liu, Danfeng Daphne Yao, and Gang Wang. [n. d.]. Android Collusive Data Leaks with Flow-sensitive DIALDroid Dataset. ([n. d.]).
[7] Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods*, Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer Berlin Heidelberg, 459–465.
[8] The Valgrind Developers. [n. d.]. Valgrind. http://valgrind.org/
[9] K. Dwivedi, H. Yin, P. Bagree, X. Tang, L. Flynn, W. Klieber, and W. Snavely. [n. d.]. DidFail. https://drive.google.com/open?id=1bneHe-Y_RKEoq2suKWeLc4387HHOrpyd
[10] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, 393–407.
[11] Facebook. [n. d.]. android_hello. https://github.com/facebook/infer/tree/master/examples/android_hello/

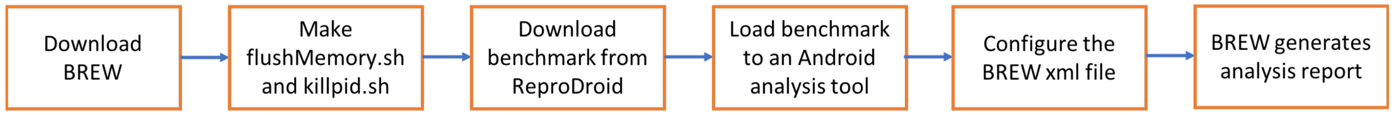

Download BREW
Make flushMemory.sh and killpid.sh
Download benchmark from ReproDroid
Load benchmark to an Android analysis tool
Configure the BREW xml file
BREW generates analysis report

Figure 2: Workflow of the BREW Tool

[12] Facebook. [n. d.]. Getting started with Infer. https://fbinfer.com/docs/getting-started.html

[13] Facebook. [n. d.]. Hello, World! https://fbinfer.com/docs/hello-world.html

[14] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. 2015. Information flow analysis of android applications in droidsafe.. In *NDSS*, Vol. 15. 110.

[15] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. 2015. Scalable and precise taint analysis for android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 106–117.

[16] N. Jovanovic, C. Kruegel, and E. Kirda. 2006. Pixy: a static analysis tool for detecting Web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*. 6 pp.–263.

[17] Wei Ming Khoo. [n. d.]. A taint-tracking plugin for the Valgrind memory checking tool. https://github.com/wmkhoo/taintgrind

[18] John R. Koza. 1992. Genetic Programming: On the Programming of Computers by Means of Natural Selection. In *MIT Press*.

[19] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. 2015. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 280–291. https://doi.org/10.1109/ICSE.2015.48

[20] F. Pauck, E. Bodden, and H. Wehrheim. 2018. Do Android Taint Analysis Tools Keep Their Promises?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. 331–341.

[21] L. Qiu, Y. Wang, and J. Rubin. 2018. Analyzing the Analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. 176–186. https://doi.org/10.1145/3213846.3213873

[22] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE Symposium on Security and Privacy*. IEEE, 317–331.

[23] Elena Sherman and Matthew B. Dwyer. 2018. Structurally Defined Conditional Data-Flow Static Analysis. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, 249–265.

[24] StatCounter. 2019. Mobile Operating System Market Share Worldwide. http://gs.statcounter.com/os-market-share/mobile/worldwide

[25] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher KrÃŒgel, and Giovanni Vigna. 2007. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis.

[26] F. Wei, S. Roy, X. Ou, and Robby. 2014. 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 (CCS '14)*. 1329–1341.

[27] Michelle Y Wong and David Lie. 2016. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware.. In *NDSS*, Vol. 16. 21–24.