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.

Taint analysis has been actively discussed and studied in both the security and software engineering communities. 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 malware analysis tools, where these tools leverage taint analysis.

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, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Taint analysis has been widely applied to enhance application/system security. Researchers/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 plan to 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 course project.

- 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 malware analysis?

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 Android malware. The reason we would like to target on Android malware analysis is that Android security is significantly important and the Android market share is more than 80% in the whole mobile market[16].

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 about 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 have a collection of information on the advantages and disadvantages of taint analysis, which could help them further protect against the security vulnerabilities in the 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 great opportunity for us to enhance technical skills related to software engineering and security.

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 contains sensitive information.
(2) Analyze the program with an analysis tool, and generate the analysis log.

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) If the tainted object eventually goes to the insecure or unauthorized destination, the analysis tool identifies that the behavior of the application is abnormal.

Specifically, the analysis techniques could be 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 running the program [15]. Different from static analysis, dynamic analysis requires to execute the target program at runtime to detect the abnormal behaviors. In addition, compared to dynamic analysis that requires to provide actual inputs, symbolic execution analysis could obtain the tracing path of the tainted object with symbolic inputs [14].

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 [11][17]. 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 malware analysis, we are discussing more details on this scope in the following.

Android security has been actively discussed in the security community. Many Android malware analysis tools apply static, or dynamic analysis to detect malware. Considering that sometimes the Android APK source code is not feasible, TaintDroid[6] 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 [10].

Furthermore, due to imperfection of the basic analysis technology, researchers have been targeting on the sensitivity types in the analysis tools, such as context-based, flow-based, and type-based [3][10] 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 [14][15]. 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[18], 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 [10]. Therefore, to achieve scalability, it requires additional optimization work (e.g., using genetic algorithms [13]) or applies different analysis techniques (e.g., using data flow analysis instead of points-to analysis [10] or tracing sensitive data based on the lifespan of variables [3]).

5 CASE STUDY ON GENERAL TAINT ANALYSIS TOOLS

In order to have a better understanding before we actually work on the Android malware analysis tools, we first work on a dynamic taint analysis tool, namely Taintgrind[12], which uses the Valgrind[5] 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*.

```c
#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.

```
0x4008EF: main (sign-ds.c:15) | t24_4548 = LOAD I32
    t21_5414 | 0x3e8 | 0xffffffff | --22310-- warning:
    evaluate_Dwarf3_Expr: unhandled DW_OP_ 0xf2
t24_4548 <- a
0x4007BC: get_sign (sign-ds.c:2) | STORE t41_1824 =
    t43_1624 | 0x3e8 | 0xffffffff | --22310-- warning:
    evaluate_Dwarf3_Expr: unhandled DW_OP_ 0xf2
x <- t43_1624
0x4007C2: get_sign (sign-ds.c:4) | t12_11253 = LOAD I32
    t49_1753 | 0x3e8 | 0xffffffff | --22310-- warning:
    evaluate_Dwarf3_Expr: unhandled DW_OP_ 0xf2
t12_11253 <- x
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
0x4007DE: get_sign (sign-ds.c:6) | t34_2455 = LOAD I32
    t31_2476 | 0xbb8 | 0xffffffff | --22310-- warning:
    evaluate_Dwarf3_Expr: unhandled DW_OP_ 0xf2
t34_2455 <- s
0x400904: main (sign-ds.c:15) | STORE t8_7902 = t10_9174
    | 0xbb8 | 0xffffffff | --22310-- warning:
    evaluate_Dwarf3_Expr: unhandled DW_OP_ 0xf2
s <- t10_9174
```

6 EXPERIMENTAL SETUP

We implement the Android analysis tools on a PC with Intel Xeon CPU and execute the tools in the Ubuntu 18.04 OS. The following subsections include the tools we attempt to work with.

6.1 Infer [4]

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 malware 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 malware analysis uses.

To deploy the Infer tool in our PC, we first install it based on the instructions in [8]. Then, based on the given hello-world examples in [9], 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.

```
class Hello_Java {
  String test() {
    String s = null;
    return s.substring(3);
  }
}
```

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
Capturing in javac mode...
Found 1 source file to analyze in /home/hwsel011/
    Spring2019/infer_examples/java_app/infer-out
Starting analysis...

legend:
  "F" analyzing a file
  "." analyzing a procedure

F..
Found 1 issue

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.    }


Summary of the reports

  NULL_DEREFERENCE: 1
```

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
Capturing in javac mode...
Found 1 source file to analyze in /home/hwsel011/
    Spring2019/infer_examples/fixed_java_app/infer-out
Starting analysis...

legend:
  "F" analyzing a file
  "." analyzing a procedure
```

```

F..
  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 [7] 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
Capturing in gradle mode...
Running and capturing gradle compilation...
Found 5 source files to analyze in /home/hwsel011/
    Spring2019/infer/examples/android_hello/infer-out
Starting analysis...

legend:
  "F" analyzing a file
  "." analyzing a procedure

FFFFF.........................................
Found 3 issues

app/src/main/java/infer/other/MainActivity.java:22: error
    : NULL_DEREFERENCE
  object returned by  source()  could be null and is
    dereferenced at line 22.
  20.     @Override
  21.     protected void onCreate(Bundle
    savedInstanceState) {
  22. >     source().toString();
  23.     }
  24.

app/src/main/java/infer/inferandroidexample/MainActivity.
    java:25: error: NULL_DEREFERENCE
  object s  last assigned on line 24 could be null and
    is dereferenced at line 25.
  23.       setContentView(R.layout.activity_main);
  24.       String s = getDay();
  25. >     int length = s.length();
  26.       writeToFile();
  27.     }

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.       }


Summary of the reports

  NULL_DEREFERENCE: 2
     RESOURCE_LEAK: 1
```

In summary about the Infer tool, it works as expected based on the information listed in [4]. 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 [7], 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.

6.2 FlowDroid [3]

Furthermore, we implement FlowDroid in our PC. FlowDroid is an advanced static taint analysis tool. In addition, to analyze the malware 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 aliasing [1], which shuffles the sensitive data 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 only consumes 2.478s to complete the analysis, which is much faster compared to Facebook Infer.

```
$ java -jar soot-infoflow-cmd/target/soot-infoflow-cmd-
    jar-with-dependencies.jar -a ../DroidBench/apk/
    Aliasing/Merge1.apk -p ~/Android/Sdk/platforms/
    android-22/android.jar -s soot-infoflow-android/
    SourcesAndSinks.txt
[main] INFO soot.jimple.infoflow.android.
    SetupApplication$InPlaceInfoflow - The sink
    virtualinvoke $r8.<android.telephony.SmsManager:
    void sendTextMessage(java.lang.String,java.lang.
    String,java.lang.String,android.app.PendingIntent,
    android.app.PendingIntent)>("+49 1234", null, $r7,
    null, null) in method <de.ecspride.MainActivity:
    void aliasFlowTest()> was called with values from
    the following sources:
[main] INFO soot.jimple.infoflow.android.
    SetupApplication$InPlaceInfoflow - - $r7 =
    virtualinvoke $r6.<android.telephony.
    TelephonyManager: java.lang.String getDeviceId()>()
    in method <de.ecspride.MainActivity: void
    aliasFlowTest()>
[main] INFO soot.jimple.infoflow.android.
    SetupApplication$InPlaceInfoflow - Data flow solver
    took 0 seconds. Maximum memory consumption: 36 MB
```

```
[main] INFO soot.jimple.infoflow.android.SetupApplication
        - Found 1 leaks
```

6.3 TaintDroid[6]

Different from the previous tools that support offline checking only, TaintDroid is able to monitor the security of Android apps at runtime, which is similar to an antivirus app in Android phones.

Unfortunately, we fail to implement TaintDroid[6] in our PC because this tool is strict with many dependent environment configurations. We encounter many challenges while setting the configurations and have to give up on the deployment for this tool considering the time constraints. In addition, the recommended platform to evaluate the tool is an Android phone. Considering that we do not have such a device, we leave this tool as our future work.

7 ADDITIONAL NOTES

In this submission, we mainly worked on Section 6, which includes our current experimental setup and preliminary results, such as time consumption of the tool execution and analysis of the tools. In the next phase, we are going to finish the experimental part by implementing more Android-target taint analysis tools, and further answer RQ3.

REFERENCES

[1] Steven Arzt. [n. d.]. Merge1.apk. https://github.com/secure-software-engineering/DroidBench/blob/master/apk/Aliasing/Merge1.apk
[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] 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.
[5] The Valgrind Developers. [n. d.]. Valgrind. http://valgrind.org/
[6] 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.
[7] Facebook. [n. d.]. android_hello. https://github.com/facebook/infer/tree/master/examples/android_hello/
[8] Facebook. [n. d.]. Getting started with Infer. https://fbinfer.com/docs/getting-started.html
[9] Facebook. [n. d.]. Hello, World! https://fbinfer.com/docs/hello-world.html
[10] 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.
[11] 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.
[12] Wei Ming Khoo. [n. d.]. A taint-tracking plugin for the Valgrind memory checking tool. https://github.com/wmkhoo/taintgrind
[13] John R. Koza. 1992. Genetic Programming: On the Programming of Computers by Means of Natural Selection. In *MIT Press*.
[14] 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.
[15] 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.
[16] StatCounter. 2019. Mobile Operating System Market Share Worldwide. http://gs.statcounter.com/os-market-share/mobile/worldwide
[17] 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.
[18] 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.