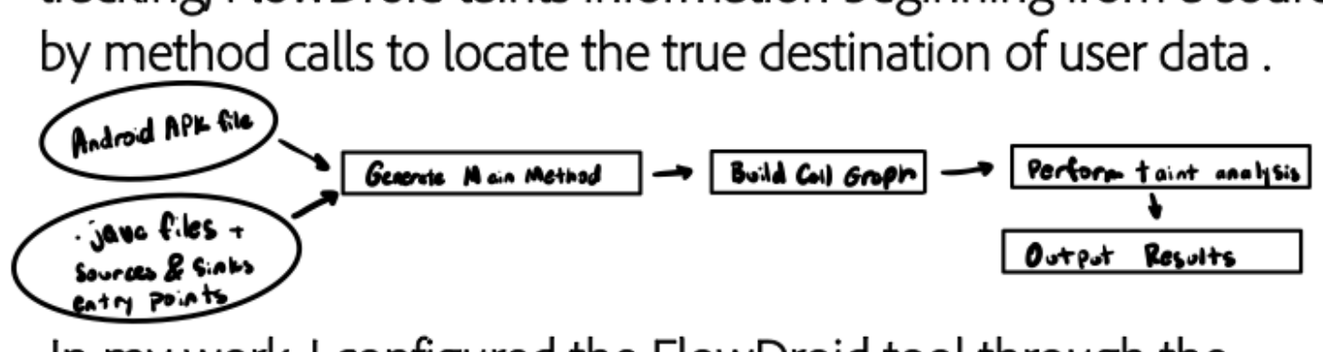
GLOBAL RESEARCH IMMERSION PROGRAM FOR YOUNG SCIENTISTS

FLOWDROID: Analyzing User Privacy Leaks in Android Apps Through Object-Sensitive and Life-cycle Aware Taint Tracking

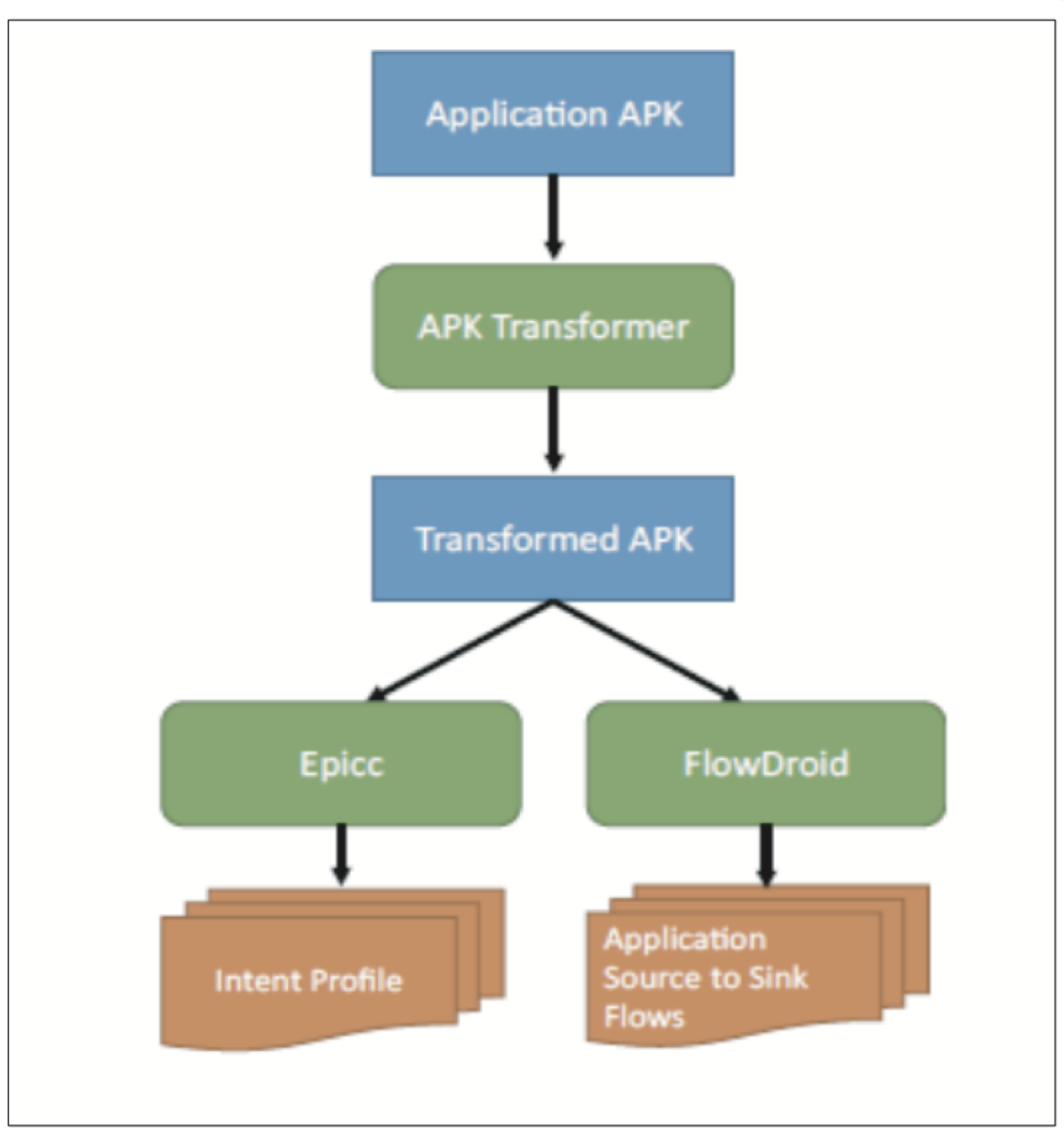
Author: Alexander Yao - University of Maryland College Park - alex.r.yao@gmail.com/ayao@umd.edu

1. Introduction

- The development of mobile devices in the past half-century has revolutionized the daily lifestyles of all people. However, with such a unified reliance on technology comes expected security and privacy risks. Privacy agreements appear in almost every new download, application, or service, but are vastly overlooked for many reasons.
- Even if users read all articles stated in these lengthy agreements, <u>how can we be sure our data is actually safe</u>? To understand the hidden inconsistencies between lengthy privacy policies agreements and actual developer code, a tool has been designed to uncover potential dangers apps pose to user security: FlowDroid.
- FlowDroid is a tracking tool created to analyze an application's byte code and configuration files to locate any potential privacy leaks either created by developers' carelessness, or malicious intent. By utilizing taint tracking, FlowDroid taints information beginning from a source and ending at a sink, tracing paths generated by method calls to locate the true destination of user data .



- In my work, I configured the FlowDroid tool through the terminal's command line, as well as Flowdroid's built-in android directory, and analyzed several various Android app apk's to demonstrate Flowdroid's capabilities.

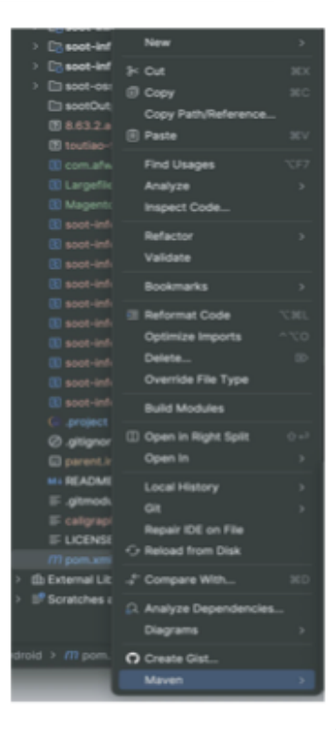


2. Soot Framework & Maven

```
<dependency>
    <groupId>org.soot-oss</groupId>
    <artifactId>soot</artifactId>
    <version>${soot.version}</version>
</dependency>
```

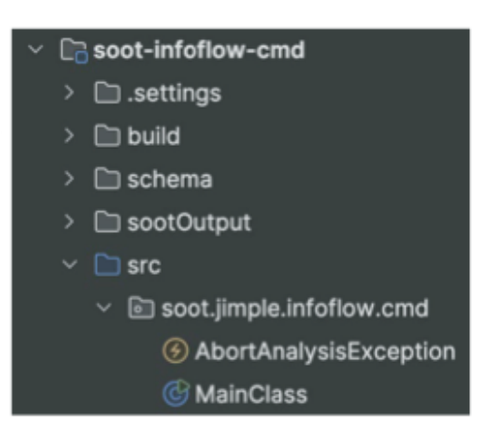
- Soot is a framework utilized in the FlowDroid GitHub repository and is essential for FlowDroid's precise static taint analysis.
- Soot provides four intermediate representations to analyze & transform Java bytecode Baf, Jimple, Shrimple, and Grimp, however, I focused on understanding and utilizing Jimple in my work with FlowDroid. Jimple is essentially a type of middle code to which the Java code is converted to that is easier to analyze than pure bytecode.
- Soot can be added to any Java project through any Java build system, such as Maven and Gradle, as a dependency.
- Within my research, I used Maven to build and configure FlowDroid through IntelliJ IDEA.
- The analysis was run on a remote server and I used apk's provided by my professor and others I found online.

```
8.63.2.apk.1
toutiao-9-1-0.apk.1
com.afwsamples.testdpc_9.0.9-9009_minAPI21(nodpi)_apkmirror.com.apk
LargefilesFinder-Free_1.40S_APKPure.apk
MagentoMobileSDKDemo_1.1.1_APKPure.apk
```



3. Method #1

- Within FlowDroid, there are two methods to perform taint analysis. The first of which is the provided main class that exists in the cmd directory.
- This default class is a standalone tool for users that strictly want to run FlowDroid without writing any code. It can be configured using specific linux arguments, like -a, -p, -s, and other identifiers to specify the inputs.




```
alex.r.yao@Alexs-MacBook-Pro-2 soot-infoflow-cmd % java -jar soot-infoflow-cmd-2.13.0-jar-with-dependencies.jar \
  -a 8.63.2.apk.1 \
  -p android-35/android.jar \
  -s SourcesAndSinks.txt
```

- Although the ease of use for the given main class is beneficial for starters, it's less flexible than performing taint analysis through the android directory.

- The second method to execute taint analysis is through the soot-inflow-android directory. Unlike the cmd directory, android does not come with an executable main class - this leaves it up to the programmer to utilize the resources provided in the FlowDroid's GitHub package to configure and customize what is analyzed throughout the analysis.
- I programmed a second main class to mimic the default main class, as shown with the accepted arguments. ere are samples results of the outputs from the main classes when executed on different apk's:

APK #1: toutiao-9-1-0.apk.1

Leaks Identified: 1

APK #2: com.afwsamples.testdpc_9.0.9-9009_minAPI21(nodpi)_apkmirror.com.apk

Leaks Identified: 2

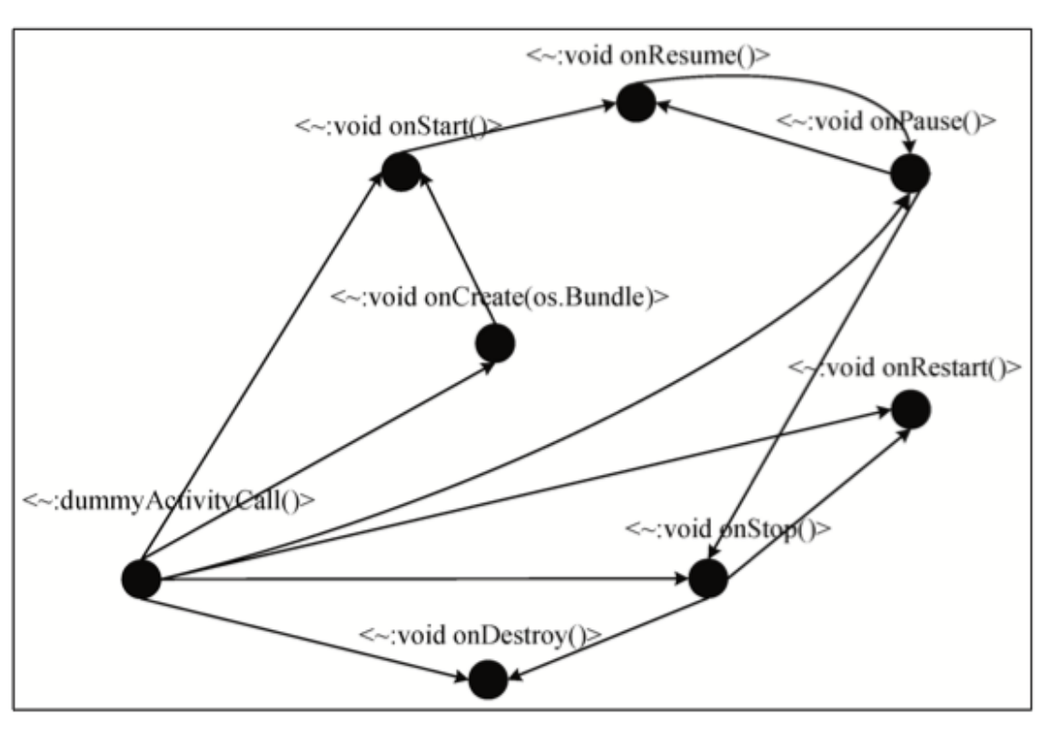
4. Method #2

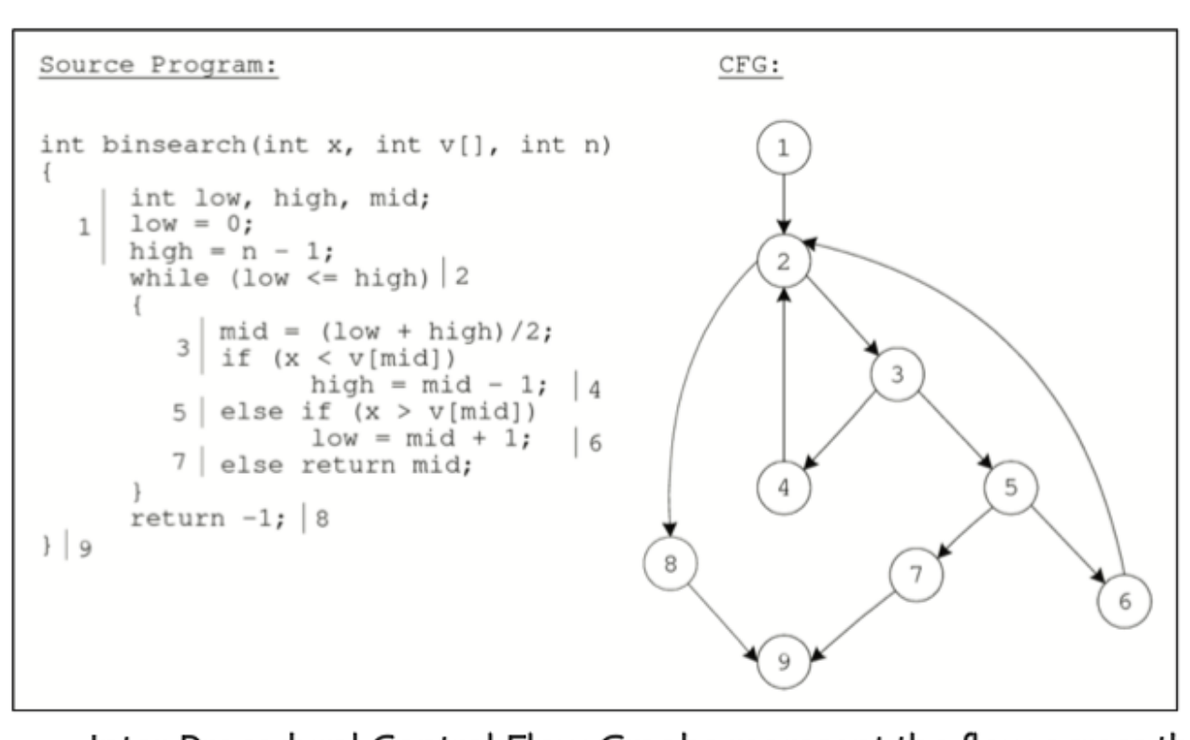
- Import Statements necessary for Flowdroid's execution
- The arguments accepted when running the main class are assigned to the respective variables - this is the primary area of versatility within this directory and the benefit, since you can configure your main class to analyze based on certain parameters.
- Soot options are set to the inputted parameters from above, configuring Flowdroid before it runs.
- SummaryTaintWrapper is a class used to provide summaries of library methods, helping reduce the analysis time by summarizing commonly used methods.
- Taint wrapper helps FlowDroid understand the behavior of library methods without analyzing their internals, using the provided summaries.
- The try statement first runs the data flow analysis and handles any exceptions that might occur
- runInfoflow starts the analysis
- If results is not null, it is printed, which outputs the details of the detected taint flows.

```
package soot.jimple.infoflow.android;

import soot.jimple.infoflow.InfoflowConfiguration;
import soot.jimple.infoflow.methodSummary.data.provider.LazySummaryProvider;
import soot.jimple.infoflow.results.InfoflowResults;
import soot.jimple.infoflow.methodSummary.taintWrappers.SummaryTaintWrapper;
import soot.options.Options;

import java.util.Collections;

public class MainClass2 {
    public static void main(String[] args) throws Exception {
        if (args.length < 4) { // Ensure that there are at least 3 arguments
            System.out.println("Usage: java MainClass2 <apk-file> <android-platforms> <sources-sinks-file>");
            System.exit(1);
        }

        String apkPath = args[0];
        String androidPlatformDir = args[1];
        String sourceSinkFile = args[2];
        String callback = args[3];

        // Set up the application with the provided paths
        SetupApplication app = new SetupApplication(androidPlatformDir, apkPath);
        // Configure Soot options and Infoflow configuration
        app.setSootConfig((Options options, InfoflowConfiguration infoflow) -> {
            options.set_src_prec(Options.src_prec_apk);
            options.set_output_format(Options.output_format_none);
            options.set_android_jars(androidPlatformDir);
            options.set_process_dir(Collections.singletonList(apkPath));
            options.set_force_android_jar(androidPlatformDir);
        });

        // Set the source and sink file for the data flow analysis
        app.setCallbackFile(callback);

        // Verify that the source/sink file is correctly set
        if (app.getConfig().getAnalysisFileConfig().getSourceSinkFile() == null) {
            System.err.println("Error: Source/sink file is not set.");
            System.exit(1);
        }

        // Optionally, set a taint wrapper if you have one
        SummaryTaintWrapper taintWrapper = new SummaryTaintWrapper(new LazySummaryProvider("summariesManual"));
        app.setTaintWrapper(taintWrapper);

        try {
            // Run the data flow analysis
            InfoflowResults results = app.runInfoflow(sourceSinkFile);
            System.out.println("Analysis Results:");
            if (results == null) {
                System.out.println("No results found.");
            } else {
                System.out.println(results);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

5. Callgraphs: CFG's vs ICFG's

- Finally, FlowDroid, with soot, has the ability to create graphs: namely call graphs (CGs) and inter-procedural control flow graphs (ICFGs).
- The larger the apk, the more intricate and complex the graph will be, since there are significantly more edges and nodes to be included within the graphs. To try to create and display an ICFG, I configured a class within the soot-infoflow-android directory called ICFGConstructor, as shown on the left.
- Below are the example diagrams of CG's and ICFGs, as well as primary differences and use cases for both graphs and why they are important and relevant to FlowDroid



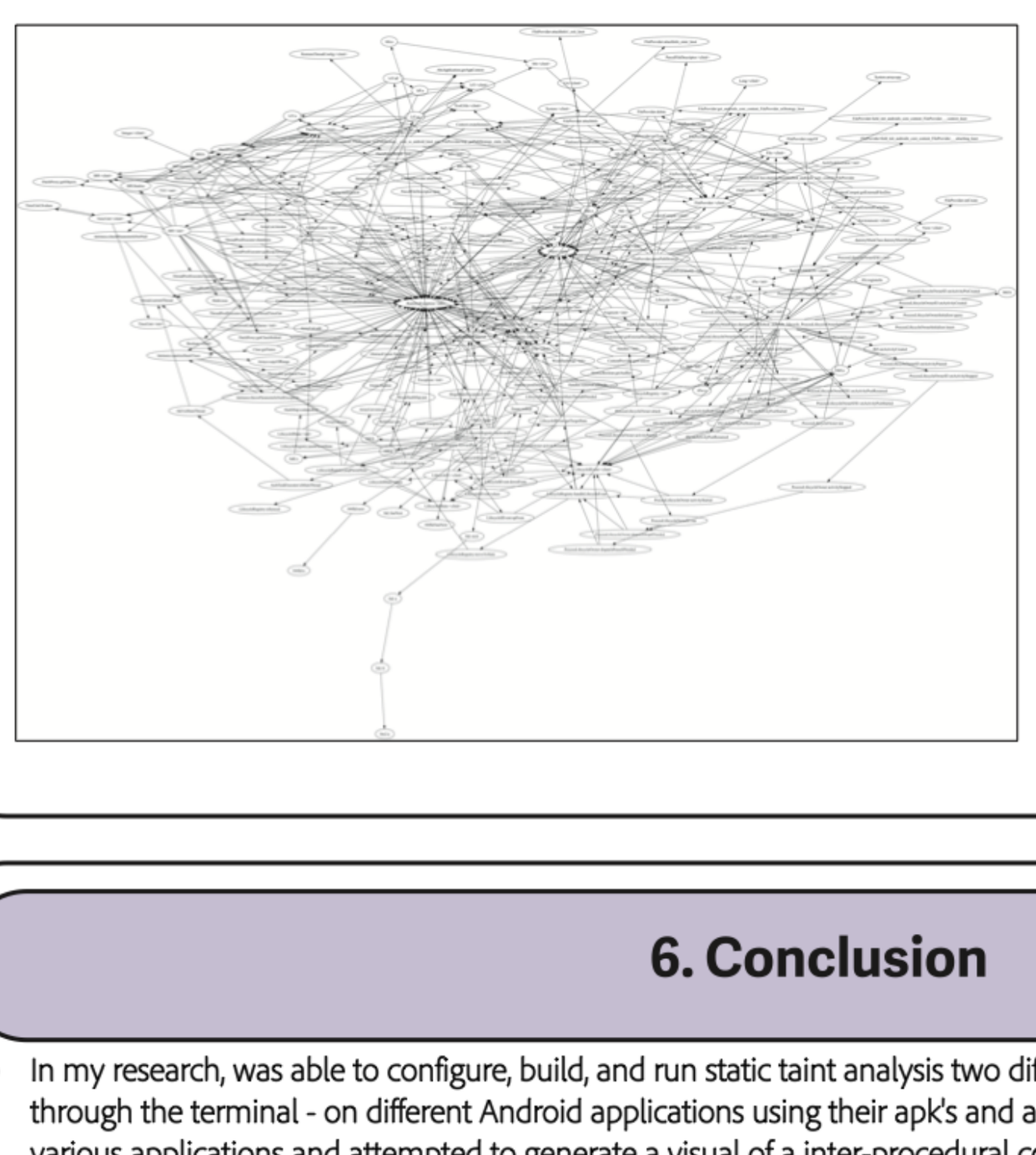


- <u>Call Graphs</u> represent the calling relationships between program methods or functions.
- Each node in the graph represents an individual method or function, and each edge is a call from one method to another - If method A calls method , a directed edge from A to exists.
- <u>Usage & Relevance:</u> Gs focus on the higher level structure of a program, such as which methods connect, and are often used in static analysis and understanding program dependencies. FlowDroid constructs call graphs by default since they help to identify all methods that are called inside an application. y understanding which methods are invoked, FlowDroid can identify potential leaks between sources and sinks.

<u>Key Differences & Similarities:</u>

- Call Graphs are more general, created to understand the high-level layout of an Android application and its method calls.
- Inter-procedural Control Flow Graphs track data flow between and within method calls, allowing FlowDroid to more precisely understand leaks between sources and sinks.
- Both graphs are vital for FlowDroid's capabilities since they allow the tool to understand the Android application as a whole, and the relationships of its method calls.

- <u>Inter-Procedural Control Flow Graphs</u> represent the flow across the entire program, including between and within the program's methods themselves.
- Each node represents any statement or block that could be inside any function or method within the program
- Two types of edges exist in the graph:
  - <u>Intra-procedural edges</u> - the control flow inside a method, which could be connecting statements or other blocks depending on the program's control structure loops, conditionals, etc.
  - <u>Inter-procedural edges</u> - method & returns. Bridges the calling statement in one method to the entry point of another method, then back to the return point.
- <u>Usage & Relevance:</u> ICFGs display a detailed analysis throughout the program's execution, tracing data flow, taint analysis, and optimization that helps with understanding the program's control flow between and within method calls.
- ICFGs enable FlowDroid to track the data flow between methods and components of the application. This precise analysis of the application allows taint analysis to accurately trace data flow on a more intricate and context-sensitive level.



- This graph that I was able to generate was different than the goal output - it included both calls from inside the ICFG constructor, as well as the apk. The ICFG should look similar to the example shown above, without redundant edges and more concise information.

6. Conclusion

- In my research, was able to configure, build, and run static taint analysis two different ways - through a directory and through the terminal - on different Android applications using their apk's and a Android jar file. I uncovered data leaks in various applications and attempted to generate a visual of a inter-procedural control flow graph created by FlowDroid. Despite inconsistencies in my control flow graph, I was able to generate a working graph, and begin configuring the output data in the graph.
- This study highlights many of FlowDroid's capabilities, including the built-in features through soot and other imported dependencies. Oftentimes, after blindly agreeing to lengthy privacy statements, app users lack transparency in companies' data usage. When analyzed with FlowDroid, some Android applications were found to have data leaks, displaying its usefulness and ability to uncover critical data leaks to keep user data secure.

7. References & Acknowledgements

[1] Arzt, Steven, Siegfried Rasthofer, and Eric Bodden. 2014. "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps." https://www.bodden.de/pubs/far+14flowdroid.pdf.
[2] Kang, Hongzhaoning, Gang Liu, Zhengping Wu, Yumin Tian, and Lizhi Zhang. 2021. "A Modified FlowDroid Based on Chi-Square Test of Permissions" Entropy 23, no. 2: 174. https://doi.org/10.3390/e23020174.
[3] Kawaguchi, Shinji, Michihiro Horie, Kazuhiko Yamamoto, and Shin-ichi Minato. 2003. "Source Code Modularization Using Lattice of Concept Slices." 2003 IEEE International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings. https://www.researchgate.net/publication/4065402_Source_code_modularization_using_lattice_of_concept_slices?_tp-eyJjb25OZXh0Ijp7ImZpcnNOUGFnZSI6Ii19kaXJlY3QiLCJwYWdlIjoiX2RpcmVjdCJ9fQ.
[4] Secure Software Engineering Group. 2024. "FlowDroid." GitHub. https://github.com/secure-software-engineering/FlowDroid.

Although I didn't reach the next step of what I wanted to accomplish, learning about how customizable and versatile of a tool FlowDroid can be was very interesting. This has definitely been the highest-level tool that I've used as someone who's pursuing computer science, and, though the research period has concluded, I do aspire to continue using the information I gained to explore FlowDroid's capabilities in the near future. The next few steps of this research after creating a proper graph would have been configuring unique source & sink files and obtaining sample apk's using a Google Pixel through it's Google Play store. To end, I would like to express my utmost gratitude towards Professor Luo and Teacher Zhuo for providing me with the resources and topic ideas to learn about FlowDroid and offering assistance when needed, especially with such a complex tool to work with.