

Vetting Network Calls in Android APKs

Chenyang Zhu

VitoZCY@outlook.com

2023 April

1 Paper Reading and Backgrounds

1.1 The Betrayal At Cloud City: An Empirical Analysis Of Cloud-Based Mobile Backends

1.1.1 Introduction

The Cloud backends of mobile applications often contain many bugs and vulnerabilities in the software layer, but developers often lack awareness of the backend they are using.

Mobile app backends can be vulnerable for a number of reasons:

- The server software stack is complex and relatively uncontrollable.
- Third-party libraries used by the app may be using the backend that beyond the developer's knowledge.

The advantages of this work over the previous ones are:

- The focus is not limited to the service layer software, and the software of all layers in the system is analyzed.
- Provides guidelines for developers on fixes.
- Opensource tools SkyWalker to make processes reproducible and available to developers themselves.

Vulnerabilities analyzed by SkyWalker can be divided into 0-day and N-day. N-day is mainly obtained by collecting software version information and inquiring known vulnerabilities, while 0-day mainly includes SQLi, XSS and XXE.

1.1.2 Motivation Example

crime City Real Police Driver is a mobile app that uses several third-party SDKs, li-brary, These include Amazon In-App Billing, SupersonicAds, Google AdMob, Unity3D, Nuance voice recognition suite, and Xamarin Mono.

The software stacks of each server can be obtained by fingerprint, and CVEs of multiple PHP versions can be obtained by comparing NVD results. In addition, Debian version running in the background are no longer supported and do not receive any updates from vendors.

For cloud servers, Google will automatically patch some software, but the premise is that the version used is within the supported range, so developers need to upgrade some software to the specified version. For third-party server problems, developers can consider reporting or participating in the bounty program. However, doing this kind of analysis manually is not practical for the average app developer, so we try to develop a atuomatic tool to achieve this.

1.1.3 Background

Backend Definition

Five layers of mobile backend stack:

- **Hardware** (HW) refers to the physical or virtual hard-ware that hosts the backend.
- **Operating System** (OS) refers to the OS running on the hardware, i.e., Linux or Windows.
- **Software Services** (SS) refers to software services running in the OS, i.e., database service, web service, etc.
- **Application Software** (AS) refers to the custom application interface used by mobile apps to interact with the running services.
- **Communication Services** (CS) refers to the communication channel supported between the mobile app and the mobile backend.

Four labels for the mobile backends with respect to the app developer:

- **First-Party** (B1st) refers to backends that are fully managed by the mobile app developers (i.e., full control over the backend).
- **Third-Party** (B3rd) refers to backends that are fully managed by third-parties (i.e., no control over the backend).
- **Hybrid** (Bhyb) refers to backends that are co-managed by third-parties and developers such as cloud infrastructure (i.e., some control over the mobile backend).
- **Unknown** (Bukn) refers to backends that ownership could not be established with high confidence.

Five mitigation strategies for developers:

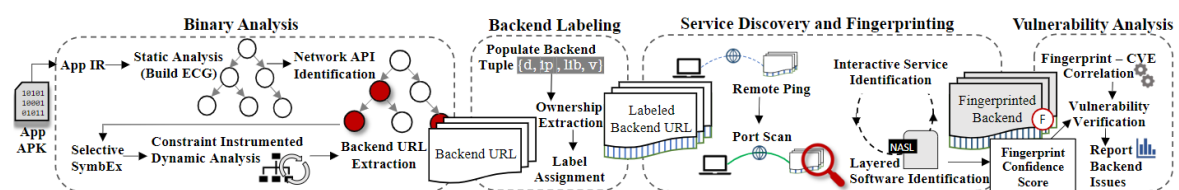
- **Upgrade** (u) the software to vendor supported versions.
- **Patch** (p) vulnerable software with a vendor patch.
- **Block** (b) incoming internet traffic to exposed services.
- **Report** (r) the vulnerability to the responsible party.
- **Migrate** (m) the backend to secure infrastructure.

Counting Vulnerabilities

N-day: collects statistics by class and instance based on CVE. If a software instance has more than one CVE, it counts as only one instance because it is considered that the repair operation only needs to be performed once.

0-day: this work looks at three types of 0-day vulnerabilities, SQLi (SQL injection), XSS (attackers inserting malicious script code and XXE into web pages), and counts every instance of every API endpoint on the mobile back end.

1.1.4 Method Implementation



Binary Analysis

SkyWalker take advantage of "Smartgen: Exposing server urls of mo-bile apps with selective symbolic execution" to perform binary analysis and extract query messages from the APK binary. SkyWalker dynamically executes code paths for network functions and extracts native usage of back-end apis. Native usage of apis includes URI paths and their parameter types/values.

Backend Labels

Backend labeling assigns one of the four labels defined in our model. The labels are used to map the responsible parties and the mitigation strategies needed (excluding unknown). Moreover, the labels are used to identify where the most common issues are found.

The author uses the IP address list (ipcat data set) of the backend based on the collected cloud service provider (CP), Colo, and third-party library (top 5k app extracts it using libscout). If Backend extracts it from a third-party library, it is directly marked 3rd. Otherwise, label it hybrid or 1st depending on whether it is in the CP or Colo list.

Service Discovery and Fingerprinting

The author focuses on reducing the impact of scanning on the target server. The network segment of the server is divided and one network segment is randomly selected each time. Check that the server is alive and send TCP ping (SYN-RST) to common ports. Scan all ports with SYN; Finally, the TCP handshake connection is performed on the surviving port. The resulting information can be used to identify OS, SS, and CS. For the OS, the author uses NASL, a script that analyzes a variety of characteristics to determine how well a server matches a certain operating system's fingerprint.

Vulnerability Analysis

For N-days, SkyWalker correlated the CVE entries with the results of the fingerprint to identify possible problems. All 983 results were manually verified, all of which were true positive.

For 0-days, taking SQLi as an example, the author prepared some messages with delay and normal messages, and measured the time at different times in a week. If the time difference was the same as the specified delay, there would be defects. XSS uses reflection patterns to insert elements into the returned content; XXE initiates a Web request on the server. 655 results were collected, with no false positives.

On the <https://mobilebackend.vet/> website, the author summarizes the results of analysis of third-party libraries reference for developers; The site allows developers to submit their apps after verifying their identity.

1.1.5 Accessment Findings

4980 applications were collected from Google Play, and backends in 4740 of them were extracted, the others were broke down during analysis. On the whole, the most problems detected are AS, the least is OS, probably the most problems written by developers.

For OS, the problem can be divided into using a version that is no longer updated, and having an update but not installing it. The OS problems in 1st are significantly higher than those in 3rd, which suggests that many developers don't take OS updates seriously. The author mentions that OS updates may cause incompatibility between SS and AS.

For SS, problems were mainly caused by PHP, than the Top 3 in the other two kinds of combined. There is also a significant percentage of server versions that are no longer updated.

Among the problems of AS, XSS is the most common and SQLi is not uncommon. But XSS is not necessarily harmful. Language-wise, PHP has had the most problems with applications, but this is also due to the popularity of PHP.

For CS, about 18k messages were collected, of which about half were HTTP and half were HTTPS, and a little more were HTTPS. Of the 4,740 applications, 446 use HTTP only and 147 use HTTPS only. About 20% of servers that use HTTPS are downgrade or outdated. In addition, the OpenSSH Bypass vulnerability exists. The author observed HTTP messages from more than 3k applications, and found that there were personal information, device information, and even six applications reset passwords through HTTP.

1.1.6 Measurement Considerations

Depending on the role of the service provider, there are some differences in the actions that can be taken. The main difference is whether the vendor controls the operating system and software stack.

Other recommendations include delegating low-level management to reliable service providers as much as possible; Special person is responsible for maintenance; Set up emergency plans; Take defensive measures;

In order to make the experiment process reasonable and legal, the author set up a special UA and reverse DNS, which can let the scanned people know about this project and can declare to quit. An application developer contacted the author. The exploits adopted by the author are all non-invasive; The author informed the responsible party of the results of the scan.

1.2 SmartGen: Exposing Server URLs of Mobile Apps With Selective Symbolic Execution

1.2.1 Background & Introduction

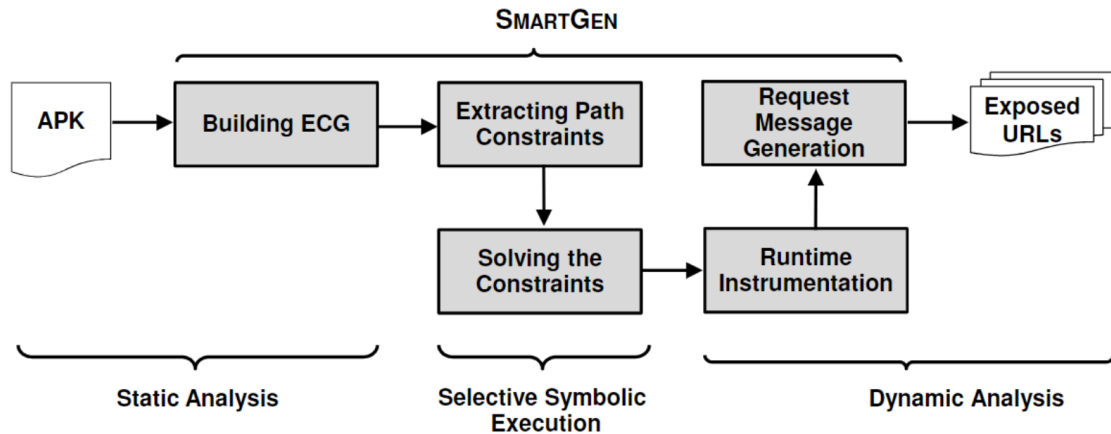
Server URLs including domain names, resource path, and query parameters are important to many security applications such as hidden service identification, malicious website detection, and server vulnerability fuzzing. Unlike traditional desktop web apps in which server URLs are often directly visible, the server URLs of mobile apps are often hidden, only being exposed when the corresponding app code gets executed. Therefore, it is important to automatically analyze the mobile app code to expose the server URLs and enable the security applications with them. We have thus developed SMARTGEN to feature selective symbolic execution for the purpose of automatically generate server request messages to expose the server URLs by extracting and solving user input constraints in mobile apps. Our evaluation with 5,000 top-ranked mobile apps (each with over one million installs) in Google Play shows that with SMARTGEN we are able to reveal 297, 780 URLs in total for these apps. We have then submitted all of these exposed URLs to a harmful URL detection service provided by VirusTotal, which further identified 8, 634 URLs being harmful. Among them, 2,071 belong to phishing sites, 3, 722 malware sites and 3, 228 malicious sites (there are 387 overlapped sites between malware and malicious sites).

contributions:

- We propose selective symbolic execution, a technique to solve input-related constraints for targeted mobile app execution. We also develop an efficient runtime instrumentation technique to dynamically insert analysis code into an executed app in real mobile devices using API hooking and Java reflection.

- We have implemented all of the involved techniques and built a novel system, SMARTGEN, to automatically generate server request messages and expose server URLs.
- We have tested SMARTGEN with 5, 000 top-ranked Android apps and exposed 297, 780 URLs in total. We found these top ranked apps will actually talk to 8, 634 malicious and unwanted web services, according to the harmful detection result from VirusTotal.

1.2.2 System Design



Build ECG

Using Soot and EdgeMiner to build Call Graph.

Extracting the Path Constraints

Identifying target APIs, and implementing taint analysis. Extracting the path constraints on dataflow.

Solving the Constraints

Using Z3 solver to deal with string APIs(e.g., length, contains, matches, startsWith, endsWith).

Runtime Instrumentation

Using dynamic hook and java reflection to inject input and get output.

Request Message Generation

Based on the UI element, using a DFS strategy to execute the activity.

1.2.3 Evaluation

we submitted all of the exposed 297, 780 URLs to harmful URL detection service at VirusTotal, which then further identified 8, 634 unique harmful URLs.

Table 3: Statistics of Harmful URLs Detected by Each Engine

Detection Engine	#Phishing Sites	#Malware	#Malicious Sites	Σ# URLs
ADMINUSLabs	0	0	4	4
AegisLab WebGuard	0	0	1	1
AutoShun	0	0	863	863
Avira	2062	941	0	3003
BitDefender	0	191	0	191
Blueliv	0	0	5	5
CLEAN MX	0	0	14	14
CRDF	0	0	150	150
CloudStat	0	0	1	1
Dr.Web	0	0	2330	2330
ESET	0	75	0	75
Emsisoft	1	43	0	44
Fortinet	8	469	0	477
Google Safebrowsing	0	13	2	15
Kaspersky	0	2	0	2
Malwarebytes hpHosts	0	1103	0	1103
ParetoLogic	0	800	0	800
Quick Heal	0	0	2	2
Quttera	0	0	6	6
SCUMWARE.org	0	8	0	8
Sophos	0	0	56	56
Sucuri SiteCheck	0	0	248	248
ThreatHive	0	0	8	8
Trustwave	0	0	80	80
Websense ThreatSeeker	0	0	56	56
Yandex Safebrowsing	0	173	0	173
Σ#Harmful URLs	2071	3818	3826	9715
Σ#Unique Harmful URLs	2071	3722	3228	8634

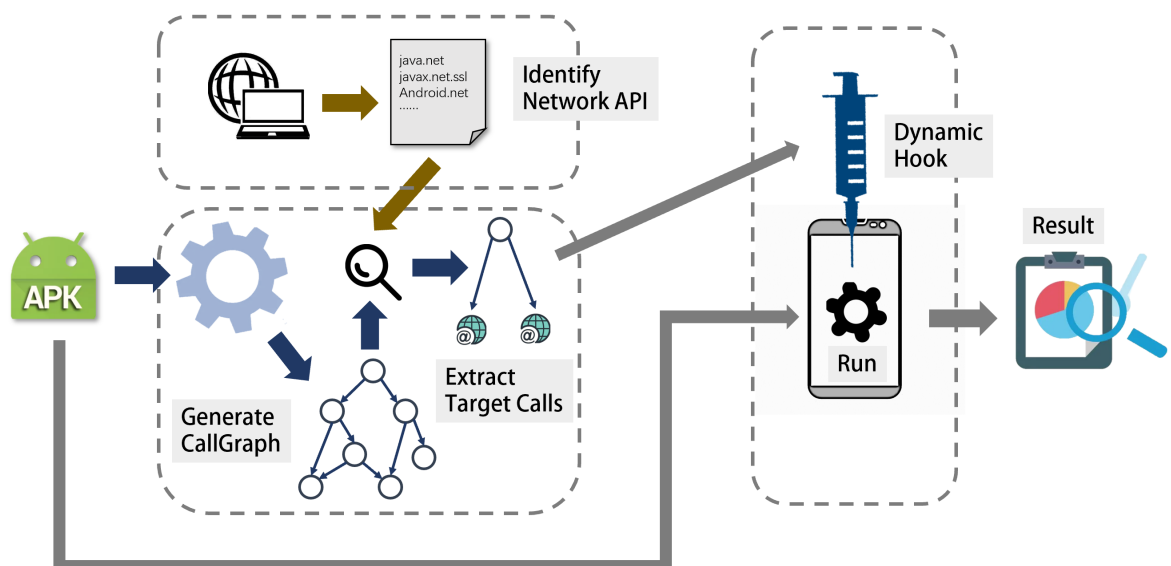
Currently, we only demonstrated how to use the exposed URLs to detect whether an app communicates with any malicious sites. There are certainly many other applications such as server vulnerability identification [31]. For instance, we can use the generated server request messages as seeds to perform the penetration testing to see whether the server contains any exploitable vulnerabilities such as SQL injection, cross-site-scripting (XSS), cross-site request forgery (CSRF), etc. We leave the study of the vulnerability fuzzing to our another future work.

2 Project Introduction

2.1 Project Overview

The main objective of this project is to partially reproduce the results of the above papers. To be specific, building a mini pipeline and perform case studies on samples.

1. Identify all network system calls/APIs available in the Android Framework.
2. Build and print the Call Graph using the Android Activity Lifecycle APIs as Entry points and find all network API calls in the malware sample.
3. Write a Frida plugin to hook the network calls and print the API name and the parameters passed to those APIs.



2.2 Environment Configuration

The development environment of this project is as follows:

OS: Ubuntu 20.04

CPU: i9-10980HK

RAM: 16G

Android Emulator: Anbox 4

Soot Version: 3.0.1

Frida Version: 16.0.11

Android Version: 7.0

JDK Version: 1.8

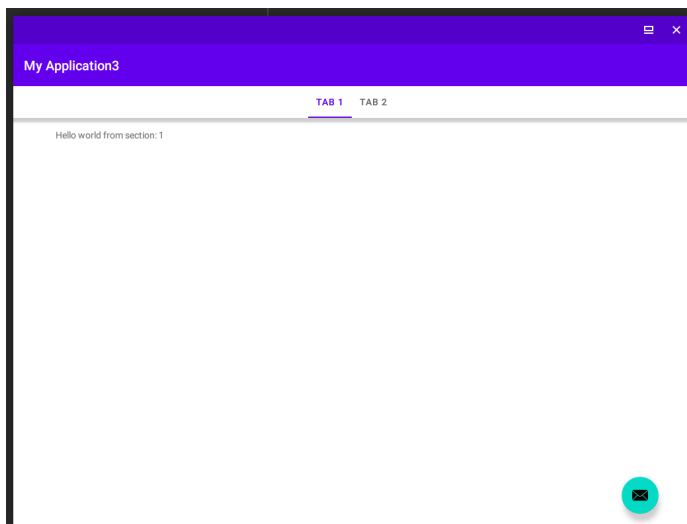
The Github Repo address of the project: <https://github.com/FlyTweety/AndroidSecurityRookie>

3 implementation

3.0 Write a Hello-World APK

In order to clearly compare the results of subsequent tests, we need a Hello-World level APK. This APK should contain certain function calls (to build the CallGraph), network calls (to find the network call API).

We use Android Studio to develop for Android 7 and start with the Basic Activity template.



This template provides a button that provides an excellent entry point for writing a test function.

Let's create a new subclass and create a network call in it. This function, `aNetworkCallFun`, sends a GET request to www.baidu.com and processes it and receives a reply. For simplicity, now our function will return a "0".

```
public class ChildClass {  
  
    public String aNetworkCallFun(String nothing){  
  
        Integer state = 0;  
        String str = "";
```

```

        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    URL url = new URL("https://www.baidu.com");

                    HttpURLConnection conn = (HttpURLConnection)
url.openConnection();

                    conn.setRequestMethod("GET");

                    Integer res = conn.getResponseCode();

                    System.out.println("Response Code : " +
conn.getResponseCode());

                    BufferedReader in = new BufferedReader(new
InputStreamReader(conn.getInputStream()));

                    String inputLine;
                    StringBuffer response = new StringBuffer();
                    while ((inputLine = in.readLine()) != null) {
                        response.append(inputLine);
                    }
                    in.close();

                    String resStr = response.toString();

                } catch (IOException e) {
                    e.printStackTrace();
                }

            }
        }).start();

        return state.toString();
    }
}

```

ATTENTION: *we must create a new thread to make the network call, not in the main thread, otherwise the process will crash*

We then call the function in the button's `onClick` method and add a counter that lets us see how many times the function has been called. Every time the button is clicked, we get a message telling us the return value of the function and the value of the counter.


```

@Override
public void onClick(View view) {

    String str = childClassInstance.aNetworkCallFun("nothing");
    count = count + 1;

    Snackbar.make(view, str + count.toString(), Snackbar.LENGTH_LONG)
        .setAction("Action", null).show();
}

```

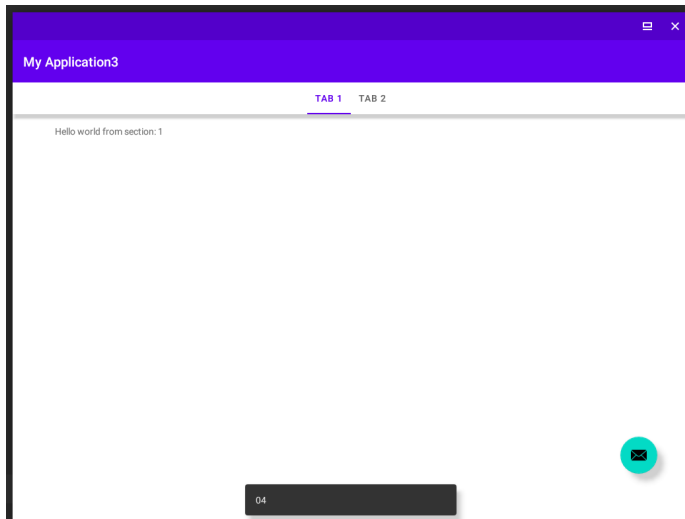
The complete code can be found in the appendix. Now use Android Studio to generate Signed APK. This requires us to create a key file of our own.

ATTENTION: Only signed APKs can be installed

Use adb to connect to the Android emulator and install it using the following command.

```
adb install app-debug.apk
```

After the installation is complete, click and see the effect.



After clicking the button four times, we can see that the message correctly displays the result of the counter. We now have a Hello-World APK!

3.1 Identify Network API

Due to my lack of knowledge of network programming, after reading some official documentation about the concrete classes (such as <https://docs.oracle.com/javase/8/docs/api/javax/net/ssl/HttpsURLConnection.html>, <https://developer.android.com/reference/javax/net/ssl/HttpsURLConnection>), I found that for beginners, to find the network calls the most insurance method, is to determine directly through whether the method is belonging to the specific class.

I first find all the common network call related classes, and then if a method belongs to one of those classes, consider it the network call I'm looking for. After the network calls are obtained through this method, we perform a manual filter to remove the API that our target does not want to do. The practice test shows that many network calls are repeated, so the number of network calls found by this method is not very large. So it can be handled manually, and this method is at least feasible.

The final list of network call related classes is as follows:

```
Set<String> networkClassRecords = new HashSet<String>(){
    {
        // java.net
        add("java.net.URL");
        add("java.net.URLConnection");
        add("java.net.HttpURLConnection");
        add("java.net.Proxy");
        add("java.net.InetSocketAddress");
        add("java.net.InetAddress");
        add("java.net.Socket");
        add("java.net.URLEncoder");
        // javax.net
        add("javax.net.ssl.SSLContext");
        add("javax.net.ssl.SSLParameters");
        add("javax.net.ssl.SSLSocketFactory");
        add("javax.net.ssl.HttpsURLConnection");
        // android.net
        add("android.net.ConnectivityManager");
        add("android.net.NetworkInfo");
        add("android.net.NetworkRequest");
        add("android.net.Uri");
        add("android.net.wifi.WifiInfo");
        add("android.net.wifi.WifiManager");
        // org.apache.http
        add("org.apache.http.HttpResponse");
        add("org.apache.http.client.HttpClient");
        add("org.apache.http.client.methods.HttpGet");
        add("org.apache.http.impl.client.DefaultHttpClient");
        add("org.apache.http.HttpEntity");
    }
};
```

The code used to determine whether it is a network call is based primarily on string matching. First, a large number of Android and Java library functions are filtered out, and then the class to which the method belongs is compared against our records.

```
public boolean isNetworkCallLoose(SootMethod sootMethod){
    if(AndroidUtil.isAndroidMethod(sootMethod))
        return false;
    if(sootMethod.getName().equals("dummyMainMethod"))
        return false;
    if(sootMethod.toString().contains("<init>") ||
sootMethod.toString().contains("<clinit>"))
        return false;
    if(!isNotJavaLibFuns(sootMethod))
        return false;

    if(networkClassRecords.contains(sootMethod.getDeclaringClass().getName())){
        return true;
    } else {
        return false;
    }
}
```

```
}  
}
```

After reading some official documents, reviewing some real project source code, and consulting with ChatGPT, we can also attempt to compile a list of commonly used network call names, including class names and method names. When determining whether it is a network call, both the class name and method name must match. The code is as follows:

ATTENTION: Although this list covers a lot of network calls, it is not exhaustive and may result in a relatively high false negative rate. The benefit of this list is that it has a relatively low false positive rate. Consulting experts and studying real-world project code are good ways to improve it. It is still being continuously updated.

```
Map<String, String> networkCallRecords = new TreeMap<String, String>(){  
    {  
        // java.net  
        put("java.net.URL", "openConnection");  
        put("java.net.URL", "<init>");  
        put("java.net.URLConnection", "connect");  
        put("java.net.HttpURLConnection", "setRequestMethod");  
        put("java.net.HttpURLConnection", "getResponseCode");  
        put("java.net.HttpURLConnection", "getInputStream");  
        put("java.net.Proxy", "proxySelector");  
        put("java.net.InetSocketAddress", "createUnresolved");  
        // javax.net.ssl  
        put("javax.net.ssl.HttpsURLConnection", "setSSLSocketFactory");  
        put("javax.net.ssl.HttpsURLConnection", "setHostnameVerifier");  
        put("javax.net.ssl.SSLContext", "getInstance");  
        put("javax.net.ssl.SSLContext", "init");  
        put("javax.net.SocketFactory", "getDefault");  
        put("javax.net.ssl.X509TrustManager", "checkClientTrusted");  
        put("javax.net.ssl.X509TrustManager", "checkServerTrusted");  
        put("javax.net.ssl.X509TrustManager", "getAcceptedIssuers");  
        put("javax.net.ssl.TrustManagerFactory", "getInstance");  
        put("javax.net.ssl.TrustManagerFactory", "init");  
        put("javax.net.ssl.SSLSession", "getPeerCertificates");  
        put("javax.net.ssl.SSLSession", "getCipherSuite");  
        put("javax.net.ssl.SSLSocket", "startHandshake");  
        put("javax.net.ssl.SSLSocket", "getSession");  
        put("javax.net.ssl.SSLSocketFactory", "getDefault");  
        put("javax.net.ssl.X509ExtendedTrustManager", "checkClientTrusted");  
        put("javax.net.ssl.X509ExtendedTrustManager", "checkServerTrusted");  
        put("javax.net.ssl.X509ExtendedTrustManager", "getAcceptedIssuers");  
        put("javax.net.ssl.X509KeyManager", "chooseClientAlias");  
        put("javax.net.ssl.X509KeyManager", "getClientAliases");  
        // android.net  
        put("android.net.ConnectivityManager", "getActiveNetworkInfo");  
        put("android.net.ConnectivityManager", "getAllNetworkInfo");  
        put("android.net.ConnectivityManager", "requestNetwork");  
        put("android.net.Network", "getSocketFactory");  
        put("android.net.NetworkInfo", "getType");  
        put("android.net.NetworkInfo", "isConnected");  
        put("android.net.NetworkInfo", "isAvailable");  
        put("android.net.Uri", "parse");  
        put("android.net.wifi.WifiManager", "getConnectionInfo");  
    }  
}
```

```

        put("android.net.wifi.WifiManager", "getScanResults");
        put("android.net.Proxy", "getDefaultProxy");
        put("android.net.ProxyInfo", "getPacFileUrl");
        put("android.net.ProxyInfo", "getHost");
        put("android.net.Uri", "fromFile");
        put("android.net.Uri", "fromParts");
        put("android.net.ConnectivityDiagnosticsManager",
"startNattSocketTest");
        put("android.net.wifi.WifiConfiguration", "allowedAuthAlgorithms");
        put("android.net.wifi.WifiConfiguration", "allowedGroupCiphers");
        put("android.net.Uri", "parse");
        put("android.webkit.URLUtil", "isValidUrl");
    }
};

public boolean isNetworkCallStrict(SootMethod sootMethod){
    //System.out.println(sootMethod.getDeclaringClass().getName());
    if(AndroidUtil.isAndroidMethod(sootMethod))
        return false;
    if(sootMethod.toString().contains("<init>") ||
sootMethod.toString().contains("<clinit>"))
        return false;
    if(sootMethod.getName().equals("dummyMainMethod"))
        return false;
    //Travel the map, to see if match
    for (Map.Entry<String, String> entry : networkCallRecords.entrySet()) {
        String className = entry.getKey();
        String methodName = entry.getValue();
        if(sootMethod.getDeclaringClass().getName().equals(className)){
            if(sootMethod.getName().equals(methodName)){
                return true;
            }
        }
    }
    return false;
}

```

Please note that if the user overrides a library function's method, our code will not detect it. One approach to address this is to consider the method's parent class when identifying whether a method is a network call (which can be easily accomplished using Soot). However, because this possibility is relatively small, we will not consider it in this project for now.

In fact, there are many pieces of code that are not directly related to networking but are commonly used in network calls, such as the `InputStreamReader` and `BufferedReader` classes, which are often used to process strings obtained from network calls. If the goal is to achieve the most comprehensive coverage, these classes could also be considered.

3.2 Using SOOT to Identify Network Calls in APK

3.2.1 Soot Introduction

Soot is a open-source java framework for analyzing and transforming Java and Android applications. Soot transforms programs into an intermediate representation, which can then be analyzed. Soot can do Call-graph construction, Points-to analysis, Def/use chains, Template-driven Intra-procedural data-flow analysis, Template-driven Inter-procedural data-flow analysis, and Taint analysis in combination with FlowDroid or IDEal.

In our project, we need to analyze Android APKs and build a CallGraph. Although Android code is written in Java, because we are dealing with APK files and do not have the source code, we cannot use tools such as CodeQL. Therefore, the SOOT tool, which can directly analyze Android bytecode, has become the best choice. Although there are some new tools available, such as OPAL, they are far less mature than SOOT.

ATTENTION: *In the real world, it is more common to decompile binary files and manually search for functions to inject. However, this approach is often used to handle a small number of functions. Because we need to search for all network calls in bulk, we use SOOT to accomplish this.*

3.2.2 Setup Soot

Basically, just follow this: <https://github.com/soot-oss/soot/wiki/Introduction:-Soot-as-a-command-line-tool>

3.2.3 Build Call Graph

We will briefly discuss the process of writing SOOT code to generate a CallGraph from scratch, using the code framework from <https://github.com/noidsirius/SootTutorial> as a starting point. I made some modifications to the original code.

Let's ignore Android for now and start by analyzing Java code.

First, we use code to set the parameters we want to parse (obviously, so many parameters are not suitable for the command line).

```
String[] sootArgs = {
    "-pp",
    "-process-dir", targetFilePath,
    "-allow-phantom-refs",
    "-no-bodies-for-excluded",
    "-whole-program",
    "-w",
    "-verbose",
    "-p", "cg.spark", "enabled:true",
    "-dynamic-dir", targetFilePath,
};
Options.v().parse(sootArgs);
```

Then, we set the classpath for SOOT and load the Necessary class.

```
String sootClassPath = System.getProperty("java.home") + File.separator + "lib"
+ File.separator + "rt.jar";
Options.v().set_soot_classpath(sootClassPath);
Scene.v().loadNecessaryClasses();
```

Next, we define the basic framework of the program. The execution of SOOT is divided into a set of different packages (**packs**), each containing different phases (**phases**). This is a concept of the life cycle. We place our code in the "jtp" phase, then run the Pack and write output.

```
PackManager.v().getPack("wjtp").add(new Transform("wjtp.myTransform", new
SceneTransformer() {
    @Override
    protected void internalTransform(String phaseName, Map<String,
String> options) {
        // Our Codes
    }
})
PackManager.v().runPacks();
PackManager.v().writeOutput();
```

Then we just need one line of code to establish the CallGraph.

```
callGraph cg = Scene.v().getCallGraph();
```

We define a function `getAllReachableMethods` that takes a method as input and returns all reachable methods from it in the form of key-value pairs of source-destination, stored in a `Map`.

First, we define a queue and add the initial method to the queue. Then, for each method in the queue, we use `callgraph.edgesOutOf` to get the edges from that method to another method and keep adding new edges to the queue. Through this breadth-first search, we can obtain all the methods reachable from the method.

```
public static Map<SootMethod, SootMethod> getAllReachableMethods(SootMethod
initialMethod){
    CallGraph callgraph = Scene.v().getCallGraph();
    List<SootMethod> queue = new ArrayList<>();
    queue.add(initialMethod);
    Map<SootMethod, SootMethod> parentMap = new HashMap<>();
    parentMap.put(initialMethod, null);
    for(int i=0; i< queue.size(); i++){
        SootMethod method = queue.get(i);
        for (Iterator<Edge> it = callgraph.edgesOutOf(method); it.hasNext(); ) {
            Edge edge = it.next();
            SootMethod childMethod = edge.tgt();
            if(parentMap.containsKey(childMethod))
                continue;
            parentMap.put(childMethod, method);
            queue.add(childMethod);
        }
    }
    return parentMap;
}
```

However, there is a serious problem here: not all Method Calls can be found in the CallGraph. In fact, there are precision issues with the CallGraph generated by SOOT, where if a function call is on the right-hand side of an assignment statement, it will not be recorded in the CallGraph. Therefore, we need to manually identify this pattern.

We retrieve the `Jimple` IR and search for cases where the right-hand side of an assignment statement is a `MethodCall`, and add these methods to the queue.

```
JimpleBody body = (JimpleBody) initialMethod.retrieveActiveBody();
for (Unit unit : body.getUnits()){
    if(unit instanceof JAssignStmt){
        value rightOp = ((JAssignStmt) unit).getRightOp();
        if(rightOp instanceof InvokeExpr){
            InvokeExpr InvokeExpr = (InvokeExpr) rightOp;
            SootMethod childMethod = InvokeExpr.getMethod();
            if(parentMap.containsKey(childMethod))
                continue;
            parentMap.put(childMethod, initialMethod);
            queue.add(childMethod);
        }
    }
}
```

Then we define the `getPossiblePath()` function. This function iterates directly in the Map returned by `getAllReachableMethods()`, starting from the initial method and continuing until the target is found.

```
public static String getPossiblePath(Map<SootMethod, SootMethod>
reachableParentMap, SootMethod it) {
    String possiblePath = null;
    while(it != null){
        String itName = it.getDeclaringClass().getShortName()+"."+it.getName();
        if(possiblePath == null)
            possiblePath = itName;
        else
            possiblePath = itName + " -> " + possiblePath;
        it = reachableParentMap.get(it);
    } return possiblePath;
}
```

After these preparations, we traverse all methods of all classes in the CallGraph. For each method, we call the `getAllReachableMethods` function to obtain all reachable functions from this method. While printing the names of the functions, we call the `getPossiblePath` function to print out the path.

```
Integer classIndex = 0;
for(SootClass sootClass: Scene.v().getClasses()){
    if (sootClass.isApplicationClass()){
        System.out.println(String.format("Class %d: %s", ++classIndex,
sootClass.getName()));
        for(SootMethod sootMethod : sootClass.getMethods()){
            System.out.println(String.format("\t[Function] Method %s",
sootMethod.getName()));
            // Get all the reachable Functions
            Map<SootMethod, SootMethod> reachableMethodMap =
getAllReachableMethods(sootMethod);
            for (Map.Entry<SootMethod, SootMethod> entry :
reachableMethodMap.entrySet()) {
```

```

        SootMethod mapKey = entry.getKey();
        SootMethod mapValue = entry.getValue();
        if((mapValue != null && !isNotJavaLibFuns(mapValue))){
            System.out.println("\t\t[Reachable] " +
mapKey.getDeclaringClass().getName() + " " + mapKey.getName() + " | " +
getPossiblePath(reachableMethodMap, mapKey));
        }
    }
}
}
}
}

```

We can obtain output similar to this.

```

Class 1: TestInputMain
[Function] Method <init>
[Function] Method main
[Reachable] java.lang.StringBuilder append | TestInputMain.main -> StringBuilder.append
[Reachable] com.example.hello.Calculator <init> | TestInputMain.main -> Calculator.<init>
[Reachable] java.lang.StringBuilder <init> | TestInputMain.main -> StringBuilder.<init>
[Reachable] java.lang.Object <clinit> | TestInputMain.main -> Object.<clinit>
[Reachable] java.lang.System <clinit> | TestInputMain.main -> System.<clinit>
Class 2: Calculator
[Function] Method <init>
[Function] Method add

```

Then, we need to transform the analysis target into an Android APK and set the entry point to an Android lifecycle function.

Although SOOT can directly analyze Android APKs, `FlowDroid` is more suitable for Android. At the same time, we don't need to make too many modifications to the code because `FlowDroid` can be used as a plugin for SOOT.

First, we introduce the relevant dependencies.

```

import soot.jimple.infoflow.InfoflowConfiguration;
import soot.jimple.infoflow.android.InfoflowAndroidConfiguration;
import soot.jimple.infoflow.android.SetupApplication;

```

Then, we adjust the way of constructing the CallGraph.

```

final InfoflowAndroidConfiguration config =
AndroidUtil.getFlowDroidConfig(apkPath, androidJar, cgAlgorithm);
SetupApplication app = new SetupApplication(config);
app.constructCallgraph();
CallGraph callGraph = Scene.v().getCallGraph();

```

At the same time, we add filtering conditions to set the Android lifecycle function as the entry point. (Please refer to the specific code at <https://github.com/noidsirius/SootTutorial>.)

```

if(androidCallGraphFilter.isLifecycleMethods(sootMethod))

```

So far, we can obtain output similar to this (due to the large number of Android library functions, our filtering for library functions may not be as efficient as before).


```

Class 1: com.example.myapplication3.ui.main.PageViewModel
Class 2: com.example.myapplication3.databinding.ActivityMainBinding
Class 3: com.example.myapplication3.ChildClass
Class 4: com.example.myapplication3.MainActivity$1
Class 5: com.example.myapplication3.ChildClass$1
Class 6: com.example.myapplication3.ui.main.PageViewModel$1
Class 7: com.example.myapplication3.ui.main.PlaceholderFragment
[Life Cycle Function] Class com.example.myapplication3.ui.main.PlaceholderFragment
[Reachable] android.widget.TextView getIncludeFontPadding | PlaceholderFragment
[Reachable] kotlin.coroutines.channels.SendElementWithUndeliveredHandler remove
[Reachable] java.lang.Character isWhitespace | PlaceholderFragment.onCreate ->
[Reachable] com.google.android.material.behavior.SwipeDismissBehavior$SettleRun
[Reachable] android.view.ViewGroup setBackgroundDrawable | PlaceholderFragment

```

3.2.4 Find Network Calls

The purpose of SOOT in this pipeline is to identify all network calls. Since we have printed all reachable functions for user methods, we only need to filter out the parts we want from the results.

When printing out all reachable functions, we also add the filtering conditions written in section 3.1.

```
if((androidCallGraphFilter.isNetworkCallLoose(mapKey)))
```

We want all such output to be collected in one place, so we create a `StringBuilder` object as the output.

```
StringBuilder allNetworkOutput = new StringBuilder();
```

For each network call that meets the condition, we check if it already exists in `allNetworkOutput`. If the answer is no, we add its class name, function name, and parameters to `allNetworkOutput`.

```

String thisCall = mapKey.getDeclaringClass().getName() + " " + mapKey.getName();
int flag = 0;
Iterator<String> it = Calls.iterator();
while(it.hasNext()){
    String inCall = it.next();
    if(inCall.equals(thisCall)){
        flag = 1;
        break;
    }
}
if(flag == 0){
    Calls.add(thisCall);
    allNetworkOutput.append(mapKey.getDeclaringClass().getName() + " " +
mapKey.getName());
    List<Type> paramTypes = mapKey.getParameterTypes();
    for (int i = 0; i < paramTypes.size(); i++) {
        allNetworkOutput.append(" arg" + (i+1));
    }
    allNetworkOutput.append("\n");
}
}

```

Finally, we print out `allNetworkOutput` separately, and we can obtain results similar to this.

```

java.net.InetAddress hashCode
java.net.Socket close
java.net.URL equals arg1
java.net.InetSocketAddress hashCode
java.net.InetSocketAddress toString

```

We will carry this result to the next section.

3.3 Using Frida to Perform Dynamic Hook

In this part, using the APK I wrote earlier, I will show how to hook functions with Frida, and give out some tips.

3.3.1 Frida Introduction

Frida is a dynamic code instrumentation toolkit. It can inject snippets of JavaScript or library into native apps on Windows, macOS, GNU/Linux, iOS, watchOS, tvOS, Android, FreeBSD, and QNX. Frida also provides some simple tools built on top of the Frida API, such as trace specific function calls.

Frida injects Google's V8 engine into target processes where JS has full access to memory, hooks functions and even calls to in-process native functions to execute. So it can greatly facilitate reverse engineering, program debugging, and dynamic instrumentation.

In this project, we will use Frida to hook these APIs found by SOOT, overriding the functions. Print out the names of these functions when they are called, along with the arguments passed to them.

3.3.2 Setup Frida

According to the guidelines of <https://wrlus.com/android-security/android-frida/>, we first need to through PIP install frida and frida - tools.

```
pip install frida frida-tools
```

We also need to download the server-side of Frida and run it on the mobile phone. First, we download the server-side from <https://github.com/frida/frida/releases>.

ATTENTION: the version of the server-side must be exactly the same as the Frida version installed in the previous step, otherwise it cannot work.

Then, we use adb to establish a shell, move the downloaded file to the mobile phone, and modify the executable permission.

```
adb push frida-server-12.5.0-android-arm64 /data/local/tmp/  
adb shell  
cd /data/local/tmp/  
chmod +x frida-server-yourversion  
su  
./frida-server-yourversion
```

ATTENTION: frida-server has to run in `su` mode, otherwise the computer-side Frida will complain that it cannot find the server.

Now we can test Frida. Let's start the emulator, run the program we wrote, and try to print out the processes running on the Android emulator.

```

zcy@msiUbuntu:~$ frida-ps -Ua
PID  Name          Identifier
3    -----
495  Adobe Flash     com.adobe.flash13
514  Calendar        com.android.calendar
360  Clock           com.android.deskclock
552  Email           com.android.email
728  My Application3  com.example.myapplication3
264  Settings        com.android.settings

```

Now we have successfully printed out a process with the package name `com.example.myapplication3`. The setup for Frida is now complete.

3.3.3 Hook User Methods

The core of hooking functions is to find the target functions in memory and overload them. In the APK we just wrote, we will try to overload the `aNetworkCallFun` function.

First, we need to check if the Java environment is available.

```
if (Java.available)
```

Then, we need to find the class to inject into.

```
var childClass = Java.use("com.example.myapplication3.ChildClass");
```

And now, the most critical part: we override the target function.

```
childClass.aNetworkCallFun.implementation = function (arg1)
```

ATTENTION: I used `arg1` to replace the first parameter of the function. The name `arg1` is not important, but the number of parameters must match the number of parameters in the original function.

We obtained the original return value `originalResult` by running the original function, and designed a new return value `modifiedResult`.

```
var originalResult = this.aNetworkCallFun(arg1);
var modifiedResult = "Modified Result";
```

We printed the arguments of the function call, `originalResult`, and `modifiedResult`. Then we returned the modified result.

```
send("args: " + arg1);
send("Original Result: " + originalResult);
send("Modified Result: " + modifiedResult);
return modifiedResult;
```

The complete code is as follows.

```
if (Java.available) {

    Java.perform(function () {
        var childClass = Java.use("com.example.myapplication3.ChildClass");
        childClass.aNetworkCallFun.implementation = function (arg1) {
```

```

    var originalResult = this.aNetworkCallFun(arg1);
    var modifiedResult = "Modified Result";
    send("args: " + arg1);
    send("Original Result: " + originalResult);
    send("Modified Result: " + modifiedResult);
    return modifiedResult;
  };
});
}

```

To find the target function, we need to find the class it belongs to. Similarly, to find this class, we need to find the target process. We first invoke Frida using the command line, with the following code:

```
frida -U -l ./MyFridaScript.js "com.example.myapplication3"
```

```

zcy@msiUbuntu:~/MalwareAnalysis/Mine$ frida -U -l ./MyFridaScript.js "com.example.myapplication3"

Frida 16.0.11 - A world-class dynamic instrumentation toolkit

Commands:
  help      -> Displays the help system
  object?   -> Display information about 'object'
  exit/quit -> Exit

More info at https://frida.re/docs/home/

Connected to Android Emulator 5558 (id=emulator-5558)
Failed to spawn: unable to find process with name 'com.example.myapplication3'

```

However, the result seems to be unsuccessful. Frida is able to start up normally, but it is not able to find the process we need.

ATTENTION: when invoking Frida via the command line, the name of the target process is not the Identifier seen in the `ps` command, but rather the name of the application as it appears in the Android system.

`com.example.myapplication3` is the process name we just printed using the `ps` command, but in the Android system, this application has its own name. We use the interface provided by Frida to print out the names of all processes in the Android system.

```

process = frida.get_usb_device().enumerate_processes()
print(process)

```

In the output, we found:

```
Process(pid=718, name="My Application3", parameters={})
```

We adjust our command to:

```
frida -U -l ./MyFridaScript.js "My Application3"
```

```

zcy@msiUbuntu:~/MalwareAnalysis/Mine$ frida -U -l ./MyFridaScript.js "My Application3"

Frida 16.0.11 - A world-class dynamic instrumentation toolkit

Commands:
  help      -> Displays the help system
  object?   -> Display information about 'object'
  exit/quit -> Exit

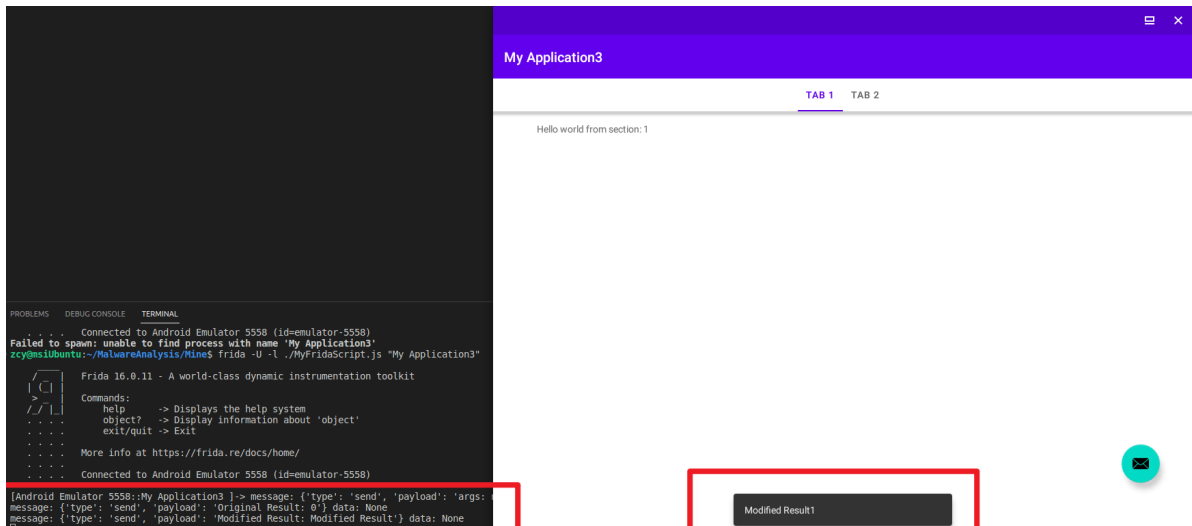
More info at https://frida.re/docs/home/

Connected to Android Emulator 5558 (id=emulator-5558)

[Android Emulator 5558::My Application3 ]->

```

Now we have successfully hooked the target function! Let's click the button and see what happens.



As we can see, when we click the button, the function's return value is modified to "Modified Result", and we also get a printed message telling us that the function was called. We have successfully hooked a user-written function!

3.3.4 Hook Library Functions

Hooking library functions is different from hooking user-written functions: the actual class to which a library function belongs may change, so we need to find it again.

Before discussing this issue, let's first modify the previous code to make it more scalable and capable of hooking multiple functions (which is what we want: Soot provides a bunch of APIs, and then we use Frida to hook all of them). We will no longer use the command line to invoke Frida, but instead use Python for interaction.

First, we need to inject Frida using Python code. We first need to connect to the device, and then use the package name to get the process PID, and attach our session to the target process with that PID.

```

package_name = "com.example.myapplication3"
device = frida.get_usb_device()
pid = device.spawn([package_name])
session = device.attach(pid)

```

ATTENTION: When using this method, the package name here is the one printed using the `ps` command, not the name of the application.

Then we define a list of functions we want to hook. Here, we use "ClassName MethodName Args(if exist)" to identify a method to be injected. We will hook our own `aNetworkCallFun` as well as a library function called `setRequestMethod` in the `HttpsURLConnection` class that is called within `aNetworkCallFun`.

```
public String aNetworkCallFun(String nothing){  
  
    Integer state = 0;  
    String str = "";  
  
    new Thread(new Runnable() {  
        @Override  
        public void run() {  
            try {  
                URL url = new URL( spec: "https://www.baidu.com");  
  
                // 创建 HttpsURLConnection 对象  
                HttpURLConnection conn = (HttpURLConnection) url.openConnection();  
  
                // 设置请求方法  
                conn.setRequestMethod("GET");  
            }  
        }  
    }).start();  
}
```

We put these two methods to be hooked in a txt file. It looks like this:

```
com.example.myapplication3.ChildClass aNetworkCallFun arg1  
javax.net.ssl.HttpsURLConnection setRequestMethod arg1
```

Next, we need to refactor the Frida code so that it can read the txt file and adjust the number of functions to be hooked flexibly.

First, we define the framework of the Frida code:

```
js_code = ""  
if (Java.available) {  
    Java.perform(function () {  
        // hook functions  
        %s  
    });  
}
```

Then, we read the function to be injected.

```
with open('HookList.txt', 'r') as f:  
    functions = f.readlines()
```

"Then we start constructing the code for hooking each function. A simple `for` loop is used to iterate over each function, and we use `strip().split()` to split out the class_name, method_name, and args. The process then proceeds identically to what was done previously.

```
hook_code = ""  
for func in functions:  
    class_name, method_name, *args = func.strip().split()  
    shortClsName = class_name.split(".")[1]  
    hook_code += f""  
        var {shortClsName} = Java.use('{class_name}');  
        if({shortClsName}) send('find' + {shortClsName});  
    """
```

```

        {shortClsName}.{method_name}.implementation = function ({arg_list}) {{
            var originalResult = this.{method_name}({arg_list});
            var modifiedResult = 'Modified Result';
            send('Class: {class_name}, Method: {method_name}, Args: ' +
{arg_list});
            send('Original Result: ' + originalResult);
            send('Modified Result: ' + modifiedResult);
            return originalResult;
        }};
    """

```

ATTENTION: Here we define `shortClsName`. When using the `Java.use()` function to retrieve the target class, we need the full class name. However, since the full class name contains ".", we cannot use it directly as a variable name. Here, we use simple string manipulation to obtain the short class name (without the package name) as the variable name.

Then we quickly realize that although this code elegantly combines Python variables and strings to enable printing of any number of arguments, it cannot handle cases where the arguments do not exist. Therefore, we need to separately handle cases where the arguments do not exist. The modified code is as follows.

```

hook_code = ""
for func in functions:
    class_name, method_name, *args = func.strip().split()
    shortClsName = class_name.split(".")[1]

    argFlag = 0
    if args:
        argFlag = 1
        arg_list = ', '.join(args)
    else:
        argFlag = 0
        arg_list = 'noarg'

    if(argFlag == 1):
        hook_code += f"""
        var {shortClsName} = Java.use('{class_name}');
        if({shortClsName}) send('find' + {shortClsName});
        {shortClsName}.{method_name}.implementation = function ({arg_list}) {{
            var originalResult = this.{method_name}({arg_list});
            var modifiedResult = 'Modified Result';
            send('Class: {class_name}, Method: {method_name}, Args: ' +
{arg_list});
            send('Original Result: ' + originalResult);
            send('Modified Result: ' + modifiedResult);
            return originalResult;
        }};
    """
    else:
        hook_code += f"""
        var {shortClsName} = Java.use('{class_name}');
        {shortClsName}.{method_name}.implementation = function () {{
            var originalResult = this.{method_name}();

```



```

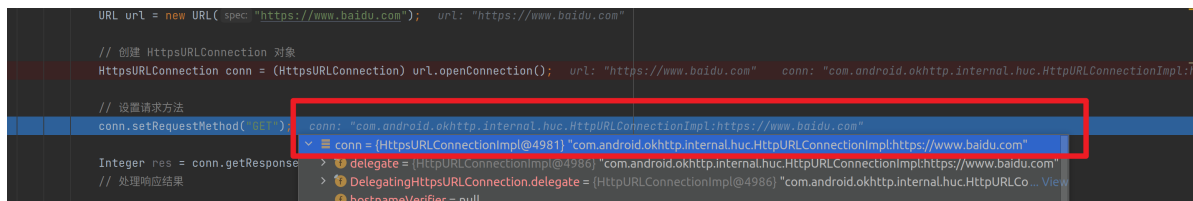
12 1 usage
13 public String aNetWorkCallFun(String nothing){
14     Integer state = 0;
15     String str = "";
16
17     new Thread(new Runnable() {
18         @Override
19         public void run() {
20             try {
21                 URL url = new URL( spec: "https://www.baidu.com");
22
23                 // 创建 HttpURLConnection 对象
24                 HttpURLConnection conn = (HttpURLConnection) url.openConnection();
25
26                 // 设置请求方法
27                 conn.setRequestMethod("GET");
28

```

After clicking the button once, the program stops at the breakpoint.



Clicking StepOver, something magical happens: we get an instance of the `com.android.okhttp.internal.huc.HttpURLConnectionImpl` class!



Similarly, Frida also provides an interface that allows us to find the class to which a method belongs. Let's add this line to our Frida code.

```
send(JSON.stringify(Java.enumerateMethods("*!setRequestMethod")));
```

This line of code will search for all methods that end with `setRequestMethod` in the process. We get this output:

```

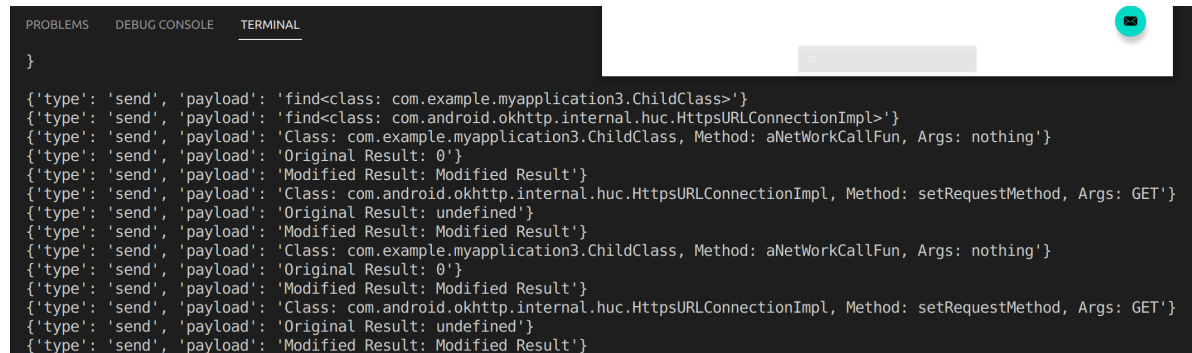
{"name": "com.android.okhttp.internal.huc.HttpURLConnectionImpl", "methods": ["setRequestMethod"]}
{"name": "com.android.okhttp.internal.huc.HttpURLConnectionImpl", "methods": ["setRequestMethod"]}
{"name": "com.android.okhttp.internal.huc.DelegatingHttpsURLConnection", "methods": ["setRequestMethod"]}
{"name": "java.net.HttpURLConnection", "methods": ["setRequestMethod"]}

```

It looks like there is some kind of delegation going on here, which causes the object we are actually generating to become an instance of the `com.android.okhttp.internal.huc.HttpURLConnectionImpl` class. Let's modify the function we want to inject.

```
com.example.myapplication3.ChildClass aNetworkCallFun arg1
com.android.okhttp.internal.huc.HttpURLConnectionImpl setRequestMethod arg1
```

Then we run our script again and click the button in the application twice.



```
}
{'type': 'send', 'payload': 'find<class: com.example.myapplication3.ChildClass>'}
{'type': 'send', 'payload': 'find<class: com.android.okhttp.internal.huc.HttpURLConnectionImpl>'}
{'type': 'send', 'payload': 'Class: com.example.myapplication3.ChildClass, Method: aNetworkCallFun, Args: nothing'}
{'type': 'send', 'payload': 'Original Result: 0'}
{'type': 'send', 'payload': 'Modified Result: Modified Result'}
{'type': 'send', 'payload': 'Class: com.android.okhttp.internal.huc.HttpURLConnectionImpl, Method: setRequestMethod, Args: GET'}
{'type': 'send', 'payload': 'Original Result: undefined'}
{'type': 'send', 'payload': 'Modified Result: Modified Result'}
{'type': 'send', 'payload': 'Class: com.example.myapplication3.ChildClass, Method: aNetworkCallFun, Args: nothing'}
{'type': 'send', 'payload': 'Original Result: 0'}
{'type': 'send', 'payload': 'Modified Result: Modified Result'}
{'type': 'send', 'payload': 'Class: com.android.okhttp.internal.huc.HttpURLConnectionImpl, Method: setRequestMethod, Args: GET'}
{'type': 'send', 'payload': 'Original Result: undefined'}
{'type': 'send', 'payload': 'Modified Result: Modified Result'}
```

We did it! We printed out the information when `setRequestMethod` was called.

Although it is likely that library functions may encounter this situation, making it impossible for us to directly use the results exported by Soot. But a good news is that the replaced class names are often fixed: if we can sort out a list, we can also quickly implement injection into library functions. Of course, we can also achieve what we want through the two methods described above.

4 Case Study

4.1 My APK

We modified the parameter passed when clicking the button to be the current system time, in order to simulate user input. Then we reinstalled the modified APK onto the Android emulator.

ATTENTION: You cannot install two APKs with the same name. So please uninstall the previous one first.

```
public void onClick(View view) {

    String str =
childClassInstance.aNetworkCallFun(String.valueOf(System.currentTimeMillis()));

    count = count + 1;

    Snackbar.make(view, str + count.toString(), Snackbar.LENGTH_LONG)
        .setAction("Action", null).show();
}
```

Then we ran SOOT to get the network calls contained in APK.

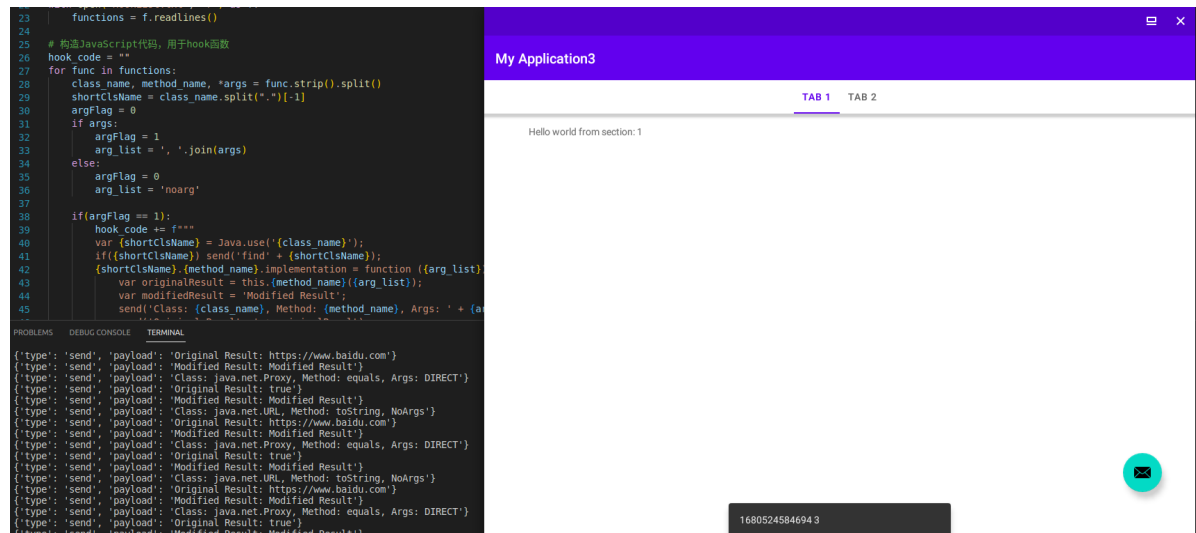
```

-----
java.net.InetAddress hashCode
java.net.URL openConnection
java.net.Socket close
java.net.URL equals arg1
java.net.InetSocketAddress hashCode
java.net.InetSocketAddress toString
java.net.URL hashCode
java.net.URL openStream
java.net.Proxy equals arg1
java.net.InetAddress toString
java.net.InetAddress equals arg1
java.net.Socket toString
java.net.URL toString
java.net.InetSocketAddress equals arg1
-----

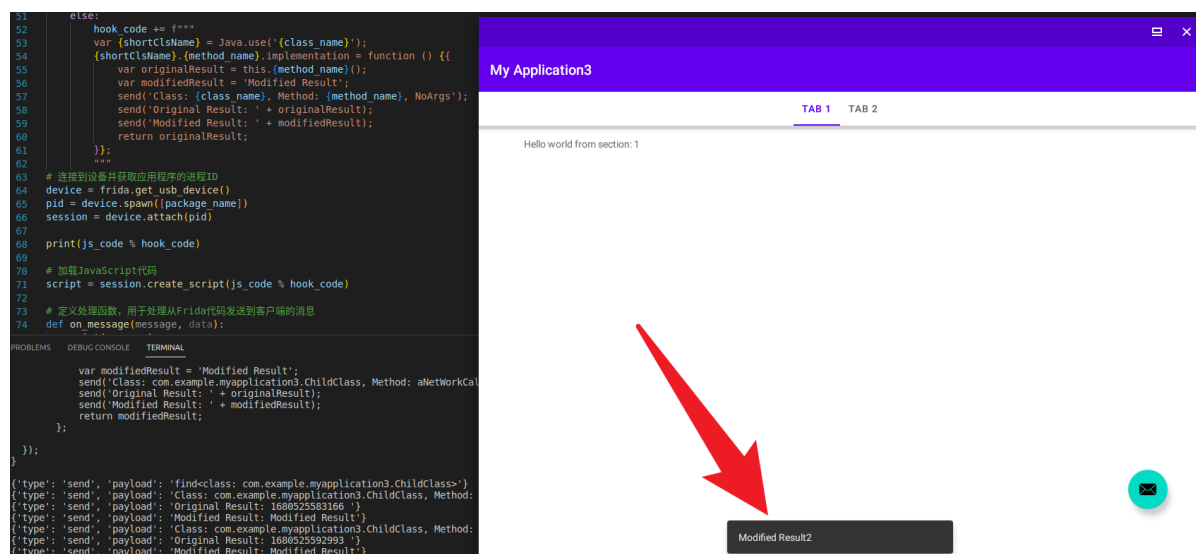
```

Copy this part into a txt file that the Frida code will read, and then call Frida through python

Our Python code automatically launches the application, and after we click the button three times, we can see that the application successfully displays the time and number of calls. Frida also prints out the name and arguments of the called function. The number of functions printed here is lower than SOOT before because we asked SOOT to find some unnecessary library functions in order to ensure the completeness of the result.



We tweak the Frida code to inject `aNetworkCallFun` alone and change the return value to "Modified Result"



We successfully modified the return value and left the counter intact.

4.2 Dendroid

Get some malware samples from <https://github.com/ytisf/theZoo/tree/master>. Of the few Android apps that have the source code, only `Dendroid.apk` contains network calls.

This is a very interesting software. When I analyzed it at first time, I got blank CallGraph.

After a lot of checking, I found that when I ran SOOT, its package name was identified as `com.adobe.flash`. In fact, it contains two packages in the source code, one is `com.adobe.flash`, which is used to fool the operating system, while the main code is in the other package, and SOOT can only identify one package, resulting in a blank CallGraph. To do this, we need to ask SOOT to find the correct package name.

My approach is to take the package that the `onCreate()` method belongs to as the package to handle. Write the following code

```
public static String getMainPackage(CallGraph callGraph){
    Iterator<SootClass> classes = Scene.v().getApplicationClasses().iterator();
    while (classes.hasNext()) {
        SootClass cls = classes.next();
        Iterator<SootMethod> methods = cls.getMethods().iterator();
        while (methods.hasNext()) {
            SootMethod method = methods.next();
            if(method.getName().equals("onCreate")){
                String packageName =
method.getDeclaringClass().getPackageName();
                return packageName;
            }
        }
    }
    return "Error";
}
```

Then run SOOT again to get some output.

```
Class 36: com.connect.MyService$callNumber
Class 37: com.connect.MyService$sendText
Class 38: com.connect.CameraView
[Life Cycle Function] Class com.connect.CameraView, Method onCreate
[Reachable] android.view.SurfaceView getHolder | CameraView.onCreate -> SurfaceView.getHolder
[Reachable] android.app.Activity setContentView | CameraView.onCreate -> Activity.setContentView
[Reachable] android.util.Log e | CameraView.onCreate -> Log.e
[Reachable] android.app.Activity onCreate | CameraView.onCreate -> Activity.onCreate
[Reachable] android.app.Activity findViewById | CameraView.onCreate -> Activity.findViewById
[Life Cycle Function] Class com.connect.CameraView, Method onResume
[Reachable] android.util.Log e | CameraView.onResume -> Log.e
[Reachable] android.app.Activity onResume | CameraView.onResume -> Activity.onResume
[Life Cycle Function] Class com.connect.CameraView, Method onStop
[Reachable] android.content.SharedPreferences$Editor putBoolean | CameraView.onStop -> SharedPreferences$Editor.putBoolean
[Reachable] android.app.Activity onStop | CameraView.onStop -> Activity.onStop
[Reachable] android.view.ContextThemeWrapper getApplicationContext | CameraView.onStop -> ContextThemeWrapper.getApplicationContext
[Reachable] android.preference.PreferenceManager getDefaultSharedPreferences | CameraView.onStop -> PreferenceManager.getDefaultSharedPreferences
[Reachable] android.util.Log e | CameraView.onStop -> Log.e
[Reachable] android.content.SharedPreferences edit | CameraView.onStop -> SharedPreferences.edit
Class 39: com.connect.MyService$deleteSms
Class 40: com.connect.CameraView$2
Class 41: com.connect.CameraView$1
Class 42: com.connect.MyService$getSentSms
Class 43: com.connect.MyService$deleteFiles
Class 44: com.connect.MyService$sendContactsText
Class 45: com.connect.MyService$uploadPictures
Class 46: com.connect.MyService$getUserAccounts
Class 47: com.connect.MyService$mediaVolumeDown
Class 48: com.connect.MyService$openWebpage
Class 49: com.connect.CaptureCameraImage

-----

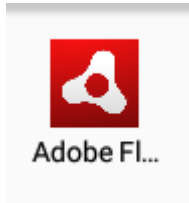
java.net.URL openStream
java.net.URL openConnection
java.net.URL toString
java.net.Socket getOutputStream
org.apache.http.HttpEntity getContent
java.net.URLEncoder encode arg1 arg2
```

Now we successfully guided SOOT to analyze the target code and get the target API.

Then we install this APK on the Android emulator.

```
adb install Dendroid.apk
```

After installing APK, we can also see that its icon and software name disguise themselves as Adobe Flash. This is indeed a common trick for malware: to disguise itself as a harmless piece of system software.



We use the `Frida-ps` command to look at the process, find its package name is `com.adobe.flash13`, and try injection

```
zcy@msiUbuntu:~/MalwareAnalysis/Mine$ frida-ps -Ua
PID  Name          Identifier
3    -----
454  Adobe Flash    com.adobe.flash13
486  Calendar       com.android.calendar
363  Clock          com.android.deskclock
514  Email          com.android.email
268  Settings       com.android.settings
```

Unfortunately, its window remained a black screen (specifically, it didn't show any window at all), no matter how much I rebooted or clicked.



Then I tried a lot of other APK, but without success. Some normal APKs have trouble finding X86 versions (the actual architecture of anbox running on Linux is X86, not arm), and the few that do have problems working for various reasons. Since I don't have a ROOT Android phone, I can't do more empirical research for the time being. But I believe that the pipeline that we have set up achieves the purpose of this program. I hope I can buy an Android phone suitable for development in the future and further carry out the case study

5 Appendix

All of the code can be found at <https://github.com/FlyTweety/AndroidSecurityRookie>