LibDroid: Summarizing information flow of Android Native Libraries via Static Analysis

By:
Chen Shi (Iowa State University), Chris Chao-Chun Cheng (Iowa State University), and Yong Guan (Iowa State University)

From the proceedings of
The Digital Forensic Research Conference
DFRWS USA 2022
July 11-14, 2022

Forensic Science International: Digital Investigation 42 (2022) 301405

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS 2022 USA - Proceedings of the Twenty-Second Annual DFRWS USA

LibDroid: Summarizing information flow of android native libraries via static analysis

Chen Shi*, Chris Chao-Chun Cheng, Yong Guan

NIST Center of Excellence in Forensic Sciences - CSAFE, Department of Electrical and Computer Engineering, Iowa State University, USA

ARTICLE INFO

Article history:

Keywords:
Android Native Library
Mobile App Forensics
Taint Analysis

ABSTRACT

With advancements in technology, people are taking advantage of mobile devices to access e-mails, search the web, and video chat. Therefore, extracting evidence from mobile phones is an important component of the investigation process. As Android app developers could leverage existing native libraries to implement a part of the program, evidentiary data are generated and stored by these native libraries. However, current state-of-art Android static analysis tools, such as FlowDroid (Arzt et al., 2014), Evihunter (Cheng et al., 2018), DroidSafe (Gordon et al., 2015) and CHEX (Lu et al., 2012) adopt the conservative approach for data-flow analysis on native method invocation. None of those tools have the capability to capture the data-flow within native libraries.

In this work, we propose a new approach to conduct native data-flow analysis for security vetting of Android native libraries and build an analysis framework, called LibDroid to compute data-flow and summarize taint propagation for Android native libraries. The common question app users and developers often face is whether certain native libraries contain hidden functions or utilize user private information. LibDroid aims to answer this question. Therefore, we build a precise and efficient data-flow analysis with the support of *SummarizeNativeMethod* algorithm, and pre-compute an Android Native Libraries Database (ANLD) for 13,138 native libraries collected from 2,627 real-world Android applications. The ANLD includes the taint propagation summary of each native method and potential evidentiary data generated or stored within the native library. We evaluate LibDroid on 52 open-source native libraries and 2,627 real-world apps. Our results show that LibDroid can precisely summarize the information flow within the native libraries.

1. Introduction

As of 2021, millions of Android apps have been published and updated in open markets such as Google Play Store (Google Play Store), Mi Store (Mi App Market) and ApkPure (Apkpure). Along with the increasing popularity of apps, the security and reliability have become a raising concern of the apps' usage.

In 2019, Jingdong Financial App (JD Finance Android App), with over 300 million customers, was reported and officially confirmed that cached the screenshots of other apps without users' awareness in the background by security analysts. At the beginning of 2020, another security issue that app developers leveraged vulnerable APIs and resulted in kids' privacy leakage can be observed in (30 Popular mHealth Apps Vulnerable to API Attacks). To mitigate the vulnerability of the raising privacy concern, hundreds of research studies are published yearly to improve the techniques of detecting privacy leakages in Android apps.

Android app taint analysis, one of countermeasures proposed to analyze and report the existence of data flows causing privacy leakages, is implemented by many existing static program analysis studies such as FlowDroid (Arzt et al., 2014), CHEX (Lu et al., 2012), (Calzavara et al., 2017; Wei et al., 2014; DroidBench benchmark; Gordon et al., 2015) etc. These prior studies proposed and implemented taint propagation rules over the program code extracted from the Android application package (APK) files. The taint analysis approach starts from tainting a variable by given user-defined source APIs that return the private data. After applying the data-flow propagation in the analyzed app, if the tainted variable can reach the given sink APIs that store or send data out, the privacy leakage is found.

However, one of the challenges shared by all existing Android

* Corresponding author.
E-mail addresses: cshi@iastate.edu (C. Shi), ccheng@iastate.edu (C.C.-C. Cheng), guan@iastate.edu (Y. Guan).

https://doi.org/10.1016/j.fsidi.2022.301405

taint analysis studies is the data-flow propagation inside the libraries (system and third-party). Due to the complexity and size of their code-base, analyzing data-flow propagation of system and third-party libraries generally takes more time than analyzing the app's code itself. Alternatively, existing studies (Arzt et al., 2014; Lu et al., 2012; Calzavara et al., 2017; Wei et al., 2014; DroidBench benchmark; Gordon et al., 2015) apply their specific data-flow summaries to skip analyzing the libraries' code. Since these specific data-flow summaries are manually crafted and only a small number of APIs included, over- or under-approximation is naively introduced when an API out of the summary is analyzed. StubDroid (Arzt and Bodden, 2016) was therefore proposed to automatically generate taint data-flow summaries for Java libraries programs through FlowDroid (Arzt et al., 2014).

Nevertheless, none of existing works can summarize native libraries written by C/C++, despite its heavy usage in both system libraries (Android native development kit, a.k.a NDK) and third-party libraries. Without a precise and large-scale data-flow summaries for these native libraries, the performance and efficiency of these prior Android static analysis approaches are not reliable when delivering the results of app privacy leakage. Thus, we propose LibDroid to automatically generate data-flow summary of given Android native libraries.

Goal: We proposed an online-offline approach to solve existing Android program analysis tools' challenges, i.e. precisely and completely summarize data-flow information inside the Android native libraries. In the offline phase, given a native library program, LibDroid parses it and updates the corresponding data-flow summary to our Android native libraries database (ANLD). When analyzing an Android app having various native APIs usage, the existing analysis tool can query the database and apply our data-flow summary. To bridge the gap between existing Android analysis tools and our database of native APIs' data-flow summaries, we implement a helper plugin program that can be coupled with the existing tools.

In implementing LibDroid, we leverage McSema (Framework for lifting x86) to transform an Android Native Library.*so* file to LLVM IR (the most commonly used intermediate representation) (Clang Static Analyzer) and build control flow graph and entry points. Then, LibDroid performs forward analysis of the control flow graph and construct the condition dependencies by leveraging the symbolic execution techniques. We evaluate LibDroid using 52 open source library and 13,138 native library collected from 2,627 real-world application downloaded from Google Play Store (Google Play Store). Our results show that LibDroid can precisely and accurately retrieve the information flow of native library. Moreover, we performed a best-efforts manual verification of the results for 52 open source library. Our results show that LibDroid achieves a precision of 97% at constructing the information flow and reporting potential sensitive information leakage. Finally, we discuss in detail of several cases to showcase how LibDroid can offer an in-depth look with valuable information for researcher and app developer in their quest to pinpoint the native library to integrate with.

In summary, our contributions are as follows:

[•]We develop LibDroid to automatically identify the native source and sink and construct information flow within native library.
[•]We evaluate LibDroid using both 52 open-source libraries and other popular libraries. Our results show that LibDroid achieves high precision in both open-source native libraries and native libraries collected from real-world applications.
[•]We create a Android Native Library Database (ANLD) which will provide comprehensive insights to app developers.

The rest of this paper is organized as follows. Section II introduces the background information with a motivating example. Section III describes the details of design and implementation. We discuss the evaluation results of LibDroid in Section IV, related words in Section V, limitations in Section VI, and conclude the paper in Section VIII.

2. Motivating example

According to previous studies (Qian et al., 2014; Wei et al., 2018), 16.7% of apps utilize native libraries based on a large scale of app collection in 2014 and the percentage increased to 21.2% in 2018. To improve the performance of the application, increasing number of Android developers take advantage of Native Development Kit (NDK) to build their own native library or to leverage native libraries implemented by others.

On the other hand, Android malware developers also make use of NDK to develop part of the app's functionality in order to bypass the security vetting. For example, the clicker malware family - an Android malware creates ad fraud by mimicking user clicks on advertisements. It relies on loading a native library (named 'kagulib') to inject into legitimate Ad-SDKs.

We also provide a motivating example to discuss the challenges to retrieve data-flow for Android native libraries. Fig. 1 illustrates a real world app (named *com.hdtotokmessenger.v doalexchat*). It consists of code written in 1) Java code: An activity component which load a native library "*face_swap_android*" and imports a native method *mainActivity_sendData()*; 2) Native library: Export the native function which leverage NDK libraries to read Java objects.

Take this application as an example, a sequence of events will be triggered as shown in Fig. 1:

1. *MainActivity* invokes native method *mainActivity_sendData()* where it pass by visited URL via argument.
2. Native method *mainActivity_sendData()* calls a native API to obtain timestamp information.
3. String variable *str* receives the visited URL and concatenate with timestamp.
4. Native method *mainActivity_sendData()* receives the sensitive data which contain visited URL and timestamp and writes to file.

3. Design and implementation

3.1. Android Native Library Database (ANLD)

Fig. 2 demonstrates the use scenario of LibDroid and Android Native Library Database (ANLD). ANLD currently contains the information flow analysis result and evidentiary data of over 2,000 popular native libraries. Specifically, each row of ANLD represents the analysis result of a native method which presents the taint propagation logic between the return value and input parameters and the types of evidentiary data that method generated. ANLD has three columns: the first column includes the native library name; the second column includes method summary; and the third column indicates the types of potential evidence generated of the certain native method. Existing studies (Arzt et al., 2014; Lu et al., 2012; Calzavara et al., 2017; Wei et al., 2014; DroidBench benchmark; Gordon et al., 2015) make an over-approximation for each argument and return value in the native method calls. Specifically, they take the union of taint tags of all input variables and assign the union to the output variable. However, this approach overlooks the native sources and sinks, such as *AKeyEvent_getEventTime(const AInputEvent key_event)* which get the time when the event

```java
package com.hdtotokmessenger.vdoalexachat
//code snippet

public class MainActivity extend Activity{
    static{
        System.loadLibrary("Agora_Native_SDK");
        //load library face_swap_android.so
    }

    protected void onSaveInstanceState(Bundle state){
        WebView w = findViewById(R.id.webView);
        String url = w.getUrl();
        //source method
        if(url != null && url.length() > 0){
            w.saveState(state);
            state.putBoolean(WEBVIEW_STATE_PRESENT, true);
            mainActivity_sendData(url);
        }
    }
}
```

```java
Agora_Native_SDK.so
//code snippet

JNIEXPORT void JNICALL
mainActivity_sendData(JNIEnv *env, jobject thisOBJ, jstring url){
    int fd = AAssetDir_getDir();
    //Native API which return file descriptor.
    int timestamp = ASurfaceTransactionStats_getLatchTime(*env, thisOBJ);
    //Native API which get the timestamp when the current activity was latched.
    string str = url + to_string(timestamp);
    if(fd < 0)
        //handle error
    ssize_t written = write(fd, str, 100);
    if(written >= 0)
        //handle successful write
    else
        //handle error
    return;
}
//visited URL is leaked and stored in file
```

Fig. 1. Code snippet of real world app *ToTok Messenger*.

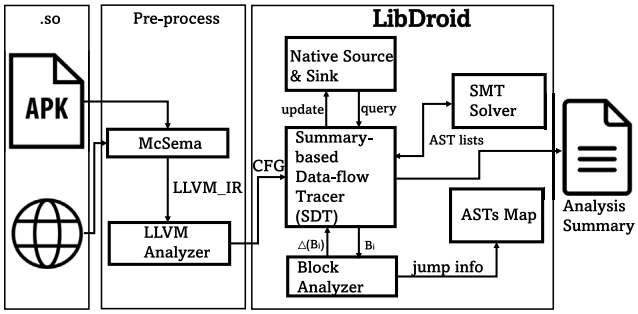

Fig. 2. The workflow of generating data flow analysis result.

occurred in the *java.lang.System.nanoTime()* time base. On the other hand, it has an over-tainting problem since it fails to disclose the true relationship between input arguments and output variable.

Given the package name and version number of an android native library, the native library matcher matches against all the records in the ANLD. It contains the taint propagation logic and native source and sink information for app developers, researchers to utilize the summary for an accurate and completed analysis result. If an app has a native library that is not included in ANLD, we will analyze the native library using LibDroid and add the summaries into ANLD. Moreover, we are going to collect more android native SDKs and update the ANLD.

Next, we discuss how LibDroid builds the ANLD.

3.2. Building the ANLD via LibDroid

Our goal is to provide an efficient native library summary to avoid potential evidence being mishandled by app developers and existing analysis tool-kits. Thus, we leverage static analysis to build the ANLD. In particular, we develop a static data-flow analysis method to build the ANLD for a large amount of Android native libraries, including widely used open-source native libraries as well as the native libraries we collected from real-world popular apps. Specifically, in our static data-flow analysis, we uncover the taint propagation rules between input parameters and return value, define a customized taint structure for local and global variables and propagate the taints through forward analysis. We leverage existing source and sink methods found by previous works (SuSi; Gordon et al., 2015), at the same time, we discover new native sources and sinks as existing lists do not include any native methods.

3.2.1. Agora Native SDK

Agora Native SDK is a well known SDK for Interactive Broadcast which has been used by several popular broadcast apps, such as *com.dongby.android.mmshow.inter* with over 10,000,000 downloads,*com.xqbl.liveoktv* with more than 5,000,000 downloads and *com.qixingzhibo.living* with 3,000,000 downloads. The Agora Native SDK supports both voice communication and live audio broadcast, it enables one-to-many and many-to-many audio or video live streaming. Different from the traditional content delivery network live broadcast, which only allows one-way communication from the hosts to the audience, the Agora SDK empowers the audience to interact with the hosts through hosting-in, like a viewer jumping onto the stage in the middle of a play to perform.Thus, we will use Agora Native SDK as an example to demonstrate the process of generating the data flow summary.

3.2.2. Pre-processing

Given an app, we first unpack the *.apk* and modify the properties in *build.gradle* file to pre-query the dynamic loading libraries. After obtaining the native library, we use McSema (Framework for lifting x86) to transform the.so file into LLVM IR (Clang Static Analyzer), which can be converted and linked into machine-dependent assembly language code. Second, we generate the method list (MList) and leverage the LLVM analyzer (Clang Static Analyzer) to build the control flow graph for each method and construct entry points. To be noted, LLVM analyzer itself is insufficient to build ANLD, it is unable to identify data-flow from sources to sinks as well as identify the file paths where the evidence is written to.

Take the Agora Native SDK as the example, after the code lifting and pre-processing through McSema and LLVM, we obtain its MList of 153 native methods and their corresponding control flow graphs which are used as the input for generating the data-flow analysis summary.

3.2.3. Native sources & sinks

There are three kinds of source and sink APIs in the native code: 1) Linux system calls; 2) JNI functions that invoke Java methods; 3) Native APIs which may generate or store sensitive information. We first combined the publicly available sources and sinks in existing tools including FlowDroid (Arzt et al., 2014), SuSi (SuSi), and DroidSafe (Gordon et al., 2015). We manually analyze all the native

APIs available for the Android system and collect 47 native source APIs and 17 native sink APIs where we define a sink as a system API that writes data to file system or sends out data through the socket, while a source is where sensitive data are created.

In this work, we focus on the types of evidence including *visited URL*, *time*, *sensor* and *textinput*. To be noted, LibDroid can be extended to other kinds of evidence simply through update the source and sinks list and extend the *TaintSet*. Next, we discuss each type of evidentiary data in detail.

1) **Visited URL:** We add 11 new native source methods for visited URL and obtained 13 Java API which return URLs from existing tools (Arzt et al., 2014; SuSi; Gordon et al., 2015). As user browsing web pages via mobile device, the visited URL can be saved by browser app or other app that using the WebView. We find that visited URL is useful evidence as searching and browsing histories have begun to appear in more criminal prosecutions (How your search history can send you to jail).
2) **Timestamp:** We include 17 new native source methods for timestamp and 20 Java source method from previous work. For example, we find that *ASurfaceTransactionStats_getLatchTime()* return the timestamp of current activity.
3) **Sensor:** We include 14 native source methods for sensors. Through our experiment, we discover several application take photos or record audio secretly, they create fake *SurfaceView* to hide the camera capturing and start the recording in background service. These are the reason that we include sensor as one category of source methods.
4) **Text input:** We include 5 new native source methods and 3 Java APIs from existing tools (Arzt et al., 2014; SuSi; Gordon et al., 2015) for user's text input. For example, the account email address and password user entered are text inputs; a search keyword is a text input.

Therefore, we extend the source and sink methods for native APIs and we will make the sources and sinks publicly available.

3.2.4. Tag for a variable

We define a Taint structure for each variable, e.g., basic type, enumerated types and derived type. Then, from the entry point in the control flow graph,we propagate variables'taint tags by applying forward data-flow analysis. We denote by $taint(v)$ the tag for a variable v. The taint tag can be used to identify the types of evidence. At meanwhile, we track where the evidence is written to by monitoring the invocation of the native file paths. Thus, we propose a taint structure that includes the following information:

[•] Evidence type (*TaintSet*): Taint tag number of the types of sensitive information that a variable is carried.
[•] File path (*Path*): File path where the variable which contain sensitive information are written to. For instance, when a native method writes data to file system via native sink API, the *Path* associated with the file descriptor is the file where data are written to.

3.2.5. Propagation rules

Propagation rules define how tags are updated when analyzing each statements in native libraries. Our rules are applied to the ARM/Thumb instructions of a.so native library. We classify instructions into two groups, i.e., explicit flow and indirect flow. We discuss the propagation rules for them separately.

Explicit flow: Table 1 shows the propagation rules for the possible explicit flow in the ARM/Thumb instructions. For example, when a statement assigns a constant to a variable, we set the (*TaintSet*) of the variable's tag to be empty; if a statement assigns one variable's value to another variable, i.e., $v_a = v_b$ which is $movR_d$, R_m in instruction format, we assign v_a 's tag to v_b 's taint set where the destination register (R_d) was assigned with union set of all taints from source registers (R_m). Next, we discuss about the statements with more complex assignment relationships.

[•]Load and Store instructions: TaintSet(mAddr) is the taint of memory at address *mAddr* and *cup* is used to combine the taint tags from registers.
[•]Binary operator: We propagate the union of the *TaintSet* of the two operands for all binary operators. We handle unary, binary and move operations.
[•]Array and String access: For array and string variable v_a, the $TaintSet(v_a)$ is carrying the union set of all tags of its elements.

Indirect flow: Indirect flow is difficult to handle and we classify indirect flow into two types.

$$v_0 = method(v_1, v_2, \ldots) \tag{1}$$

[•] Method-invoking statement: take the equation (1) as an example, we analyze data flow in the called *method*() and assign the taint tags of the return value to *TaintSet* of the variable v_0. When analyzing method calls, we discover that sometimes we could get into a loop of method calls. To avoid re-analyzing the same method twice, we create a stack to keep track of method calls and skip the same method call if the method is already on the stack. In this way, we can ensure to analyze each method in a loop only once.
[•] Conditional branches: In order to detect all conditional branches and propagate taint accordingly, we leverage the control flow graph (Clang Static Analyzer) to determine branches in the conditional structure. We design a Summary-based Data-flow Tracer (SDT) to detect flow with condition-dependencies.

3.3. Summary-based data-flow tracer (SDT)

Given the control flow graph of a native method, we apply the *SummarizeNativeMethod* to construct the data-flow summary. The benefit of this method is that we keep track of all the methods which have been analyzed, while still preserving a flow and context-sensitive data-flow analysis result. In the meantime, we update the native source and sink list as if we discover any method invoking existing source or sink APIs.

SDT takes the control flow graph of the native method as input, then we calculate the in-degree of each vertex in the control flow graph to verify that a basic block(B_i) does not have any successor that has not been analyzed, in this way, each block needs to be analyzed only once. If there is a cycle in the control flow graph, we will break the cycle arbitrarily to calculate the in-degree of the basic block. For each basic block B_i in BList, we apply a taint propagation as listed in Table 1 to update the taint set of the corresponding register and generate the summary δB_i. The called method's summary will propagate to its caller methods until the *return* is reached.

As mentioned before, we discovered 153 native methods from the Agora Native SDK through the pre-processing procedure. We take the *mainActivity_sendData* method as an example to demonstrate the process of propagating the taint and generating the data-flow summary. Taking the control flow graph of *mainActivity_sendData* method as input, we generate the BList of

Table 1
Propagation Rule for ARM/Thumb instructions.

Instruction Format	Propagation Rule	
	TaintSet	*Path*
mov R_d, *C*	$TaintSet(R_d) \leftarrow \varnothing$	$R_d.Path \leftarrow \varnothing$
mov R_d, R_m	$TaintSet(R_d) \leftarrow TaintSet(R_m)$	$R_d.Path \leftarrow R_m.Path$
unary-op R_d, R_m	$TaintSet(R_d) \leftarrow TaintSet(R_m)$	$R_d.Path \leftarrow R_m.Path$
binary-op R_d, R_m, R_n	$TaintSet(R_d) \leftarrow TaintSet(R_m) \cup TaintSet(R_n)$	$R_d.Path \leftarrow R_m.Path + R_n.Path$
binary-op R_d, R_m	$TaintSet(R_d) \leftarrow TaintSet(R_m) \cup TaintSet(R_n)$	$R_d.Path \leftarrow R_m.Path + R_n.Path$
binary-op R_d, R_m, *C*	$TaintSet(R_d) \leftarrow TaintSet(R_m)$	$R_d.Path \leftarrow R_m.Path$
store R_d, R_n, *C*	$TaintSet(mAddr) \leftarrow TaintSet(R_d)$	$R_d.Path \leftarrow R_n.Path$
load R_d, R_n, *C*	$TaintSet(R_d) \leftarrow TaintSet(R_n) \cup TaintSet(mAddr)$	$R_d.Path \leftarrow R_n.Path$
push regL, R_n, *C*	$TaintSet(mAddr) \leftarrow TaintSet(R_i, R_j)$	$R_d.Path \leftarrow R_i.Path + \cdots + R_j.Path$
pop regL, R_n, *C*	$TaintSet(mAddr) \leftarrow TaintSet(R_i, R_j)$	$R_d.Path \leftarrow R_i.Path + \cdots + R_j.Path$

mainActivity_sendData method by calculating the in-degree of each block within the control flow graph.

3.3.1. Summary of basic block (δ(B))

In order to generate the summary of each basic block, we mainly focus on three types of instructions: Data Movement Instructions, Arithmetic and Logic Instructions and Control Flow Instructions.

[•]**Data Movement Instructions:** In this category, we only track two instruction *mov* and *lea* where *mov* instruction copies the data item referred to by its first operand (i.e. register contents, memory contents, or a constant value) into the location referred to by its second operand and *lea* instruction places the address specified by its first operand into the register specified by its second operand which is used for obtaining a pointer into a memory region or to perform simple arithmetic operations. As the block B_0 in our example, the first argument arg_0 was initially push into register *r*4 and *cmp* instruction compare the value with register *r*3 where we need to propagate the TaintSet of arg_0 to register *r*4.

[•]**Arithmetic and Logic Instructions:** We will propagate the TaintSet of its first operand to second operand for all the instructions in this category, including *add*, *sub*, *inc*, etc.

[•]**Control Flow Instructions:** We focus on four instructions in this category. 1) *mov* transfers program control flow to the instruction at the memory location based on the operand. 2) *jcondition* is a conditional jump that is based on the status of one or a set of condition codes. Conditions of *jump*are stored in a special register which is called machine status word. The contents of that special register include the result information of the last performed arithmetic operation. *cmp* compares the values of the two specified operands, where the condition codes were stored in the machine status word. This instruction is equivalent to the sub instruction, the difference is that the result of the subtraction is discarded while the result is stored in the first operand for *cmp*. *cal*, *ret* correspond to call and return. The call instruction first pushes the current code location into the stack, and then performs an unconditional jump to the code location based on the label operand.

In order to identify both the types of evidence and the file paths, we propose a data structure $summary(B_i)$ which consists of TaintSet and Path to store corresponding information.

Summary(B_i): We design three custom structures to assist data-flow analysis: (1) TaintSet: It illustrates the tainted information, such as, taint type (source or sink) and taint tag. Native code can utilize JNI functions to create and manipulate Java objects, invoke Java methods, catch and throw exceptions, which means native code has the capability to pass sensitive data back to Java method and objects. Therefore, SDT adds tags to TaintSet to capture data related to Java operations in native code. (2) Path: File path associated with the target register. For instance, when an app writes data to a file system via a file descriptor, the Path associated with the file descriptor is the file where data held within the register are written to. (3)Abstract syntax tree (AST):It is challenging to analyze data-flow sensitive to all conditional branches. When performing an arithmetic comparison with variables, we model the condition to a tree of operations - an abstract syntax tree (AST). ASTs can later on be translated into constraints for an SMT solver (Moura and Bj ø rner, 2008). The control flow graph consists of nodes of basic blocks and directed edges represent the jumps, we detect the flow of condition dependencies from blocks in the control flow graph using the ASTs.

Instead of tracking a variable's concrete numerical value, when assigning a variable to another, we treat them as symbols. Later on, when performing arithmetic operations or assignment with that variable, we will propagate the symbolic variable. For example, if we have $v_1 = arg_0 + 1$ and later we have a *if* condition of *if* $v_1 > 0$. If we assign arg_0 as symbolic variable, then v_1 will transform as $arg_0 + 1$ and comparison condition will be *if* $arg_0 + 1 > 0$. In this way, we can construct condition dependencies ASTs.

We leverage the SMT solver to compute the results of two ASTs. To be noted, it can only apply to same type of ASTs. When performing merge between two different-typed ASTs, for example, int $i = 0$ which was cast to a float number by *float* $f =$ *float* (i + 1). As we discussed above, variable f will be transformed as $i + 1$, however, these two ASTs cannot be solved by SMT solver due to their different types. For the native method consists of more than one ASTs, we leverage the SMT solver to construct the final comparison condition related to method arguments.

[•] When the SDT starts, it will first add summary structure to each argument including argument index and taint information. As an example, the summary of basic block $\delta(B_0)$ is $r4 \leftarrow arg_0|\varnothing$;.

[•] As data-flow tracer perform the forward analysis, those $summary(B_i)$ will properly update and propagate TaintSet. Take the example of Fig. 1, native code concatenates the string from input parameter *url* with local variable *timestamp* and assign it to *str*. We will update the TaintSet with the combination of taint tags of input parameter *url* and local variable *timestamp*.

[•] If SDT encounters a condition jump, it will compute an AST and update it with summary structure. For example, before writing the visited URL together with timestamp to a file, native method *mainActivity_sendData* first checks whether the file descriptor is valid.

[•] When data-flow tracer encounters any method/function invocation, it will first check whether it is a source or sink API. If

so data-flow tracer will add new tag to corresponding TaintSet. For method invocation rather than source and sink API, we will check with SDT to obtain its summary δ and apply it on the summary structure of the return value.
[•] When data-flow analysis of all the blocks is over, we extract the TaintSet together with Path related to the return node to build the summary of native method which illustrate the relationship between return value and arguments, invoked source and sink APIs within the method.

We have a control flow graph of method *mainActivity_sendData*, and we calculate the in-degree of all the basic blocks. We start generating the summary δ from block with 0 in-degree which is B_0 in our case.

Native function *mainActivity_sendData* receives one input parameter. We create a summary structure $\{\varnothing, \varnothing\}|\varnothing$ which is in the format of {*TaintSet*, *Path*|*TaintCondition*}. At line 8, it invokes a native API to get the current timestamp where we create a summary structure and add timestamp taint tag to its TaintSet. Then *summary*(*str*) get *summary*(*url*) and *summary*(*timestamp*) and propagate it to variable *str*.

After finish running data-flow analysis, we collect the TaintSet, Path and TaintCondition related to each argument and return value to construct the summary $\delta(mainActivity\_sendData) = \{\{Time, arg_0\}, /data/data/com.hdtotokmessenger.vdoalexchat/| \varnothing\}$. It tells us under file path */data/data/com.hdtotokmessenger.vdoalexchat/*, we can retrieve timestamp type of evidence. If input parameter *url* carries any type of evidence, it is also stored under the same file path. Therefore, we solved the native evidence problem.

3.4. Data-flow summary result

When app developer and researcher query against ANLD of Agora Native SDK as shown in Fig. 3, our database will provide detailed analysis result of each native method within the native SDK including information of:

[•] explicit taint propagation rules for arguments and return value.
[•] detected source APIs and its evidence type.
[•] discovered sink APIs and retrieved file path.

Taking the Figs. 4 and 5 as an example, it shows the analysis result of two native methods in Agora native SDK - *mainActivity_sendData()* and *getActualVoiceType()*, <method> provides the method signature of a native method included in that native SDKs. <source> identify the source detected within the native method, it could either be one or more arguments of the method or a source method defined in the source API list. As for the source API, <source category > indicates the type of evidentiary data, such as *Time* and *Visited URL*. <source name> is the source API. <sink> could either be return value of the method containing the type of evidentiary data from <source>, or a sink API that store the type of evidence from <source>.

For example, native method *mainActivity_sendData()* in Fig. 4 demonstrate that it has two sources: 1) the first argument of the method; 2) source API *ASurfaceTransactionStats_getLatchTime()*. In addition, it identifies the file path */data/data/com.hdtotokmessenger.vdoalexchat/* where the evidentiary data is written to. While native method *getActualVoiceType()* shows an example of native method with return value where the return value should be assigned with the taint set of its first argument and add taint tag of *Time* evidence type.

4. Evaluation

In this section, we aim to evaluate the ANLD generated by LibDroid. First, we evaluate LibDroid using 52 open-source Android native libraries, including several flagship imaging processing libraries as having significant additional bundled functionality on top of the standard Android platform. In addition, we conduct a performance measurement of our system in terms of the accuracy of its *Summary-based Data-flow Analysis Tracer* based on the open-source native libraries. Second, we evaluate LibDroid on the 13,138 native libraries retrieved from 2,627 randomly selected real-world apps from Google Play Store (Google Play Store). Third, we present a case study on how forensic investigators can utilize LibDroid and ANLD to retrieve evidence from Android applications with the usage of native libraries.

We conduct experiments to answer the following questions:

Q1: How does LibDroid perform on open-source native libraries?
Q2: Is LibDroid capable of discovering critical evidentiary data generated by Android native libraries?

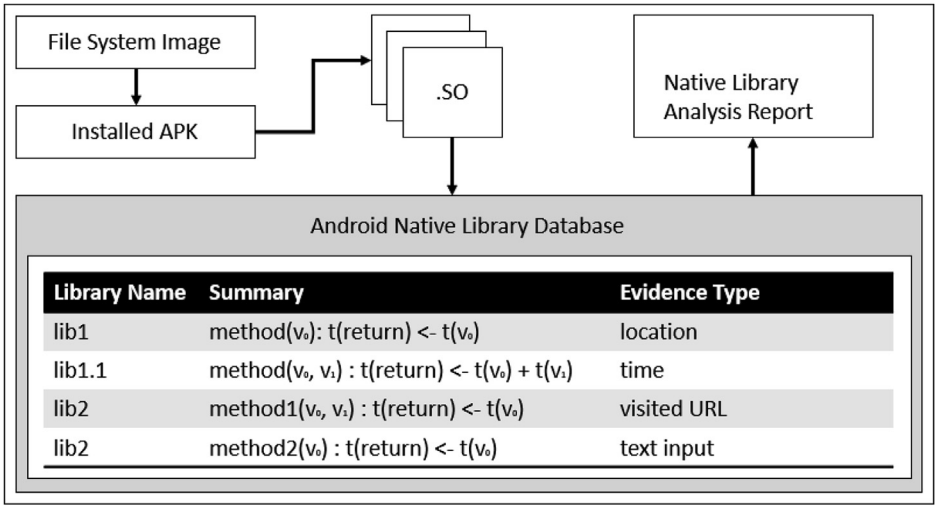

Fig. 3. Overview of ANLD

```
<library = "Agora_Native_SDK:3.0.1">
        <method = " mainActivity_sendData(arg_0)">
            <source>
                <parameter class="jString" parameter="1" />
                <source category="timestamp" |
                    source API="ASurfaceTransactionStats_getLatchTime()"/>
            </source>
            <sink "/data/data/com.hdtotokmessenger.vdoalexchat/">
        </method>
</library>
```

Fig. 4. Analysis result of native method *mainActivity_sendData*() of Agora Native SDK.

```
<library = "Agora_Native_SDK:3.0.1">
        <method ="getActualVoiceType(arg_0, arg_1)">
                <returnValue
                    <source>
                        <parameterclass="getActualVoiceType" parameter="1"/>
                        <source category="timestamp" |
                                sourcename="AKeyEvent_getEventTime"/>
                    </source>
                /returnValue>
        </method>
</library>
```

Fig. 5. Analysis result of native method *getActualVoiceType*() of Agora Native SDK.

Q3: How to use the data-flow result of native libraries to discover crucial security issues?

4.1. Results on benchmark apps

Previous studies on Android app security have designed a collection of native benchmark apps, e.g., NativeFlowBench (NativeFlowBench) provides 22 benchmark apps targeting on the challenge of native and inter-language data flow analysis.

We compare the effectiveness of LibDroid with FlowDroid (Arzt et al., 2014) and JN-SAF (Wei et al., 2018). Since LibDroid take.so file as input, we first compute the native library summary and add the summary to EviHunter (Cheng et al., 2018) to handle the taint propagation in Java for a fair comparison. We run each tool against all the benchmark apps to check if the tool can correctly report the potential data leakage and corresponding data paths. The result is shown in Table 2. We use True Positive (✔), False Positive (*) and False Negative (✘) to mark different scenarios. For apps contain more than one leakage path, we mark each of the leakage path separately. Among the 22 benchmark apps, JN-SAF (Wei et al., 2018) reports two false positive, while LibDroid (with EviHunter (Cheng et al., 2018)) only has one false alarm on *native_complexdata_stringop*. It's caused by no precise string analysis implemented by EviHunter (Cheng et al., 2018). On the other hand, FlowDroid (Arzt et al., 2014) treats all native method calls as black box and applies conservative model to taint all other arguments if one argument is tainted, thus it identifies one false positive path at *native_source_clean* and fails to detect most of data paths among the 22 benchmark apps.

4.2. Results on 52 open source native libraries

Previous studies on mobile security have developed and published Android benchmark apps, such as DroidBench (DroidBench benchmark), however, these applications are not suitable to evaluate LibDroid as their functionalities all written in Java without the usage of Android native libraries. We collected 52 open-source Android native libraries to measure the performance of LibDroid. To analyze these open-source native libraries, we include source and sink collected by Susi (SuSi) as well as the native source and sink discovered from documentation of Android native APIs. Developer and researcher can leverage the detailed data-flow analysis result as the taint propagation rule for native methods.

For the 5 native libraries, LibDroid finds 17 static file paths which store one or more type of evidentiary data. LibDroid discovers 10 *Time* type of evidence, 4 *Text Input* type of evidentiary data, 1 *Sensor* type and 3 *Visited URLs*. Among the 52 open-source native libraries, we notice that 5 of them invoke and generate *Sensor* type of evidence; 45 of these native libraries will access and store *Time* to the file system; 17 native libraries store user's *Text Input* and 14 of them obtain *Visited URLs*. In addition, we finds that many of native libraries acquire *Device ID*, *UUID*, *Contact* and *Mobile Country Code* which are listed as Others in the Table 3.

4.3. Results on native libraries retrieved from real-world apps

We randomly collected 2,627 real-world app from Google Play Store (Google Play Store) where we unpack the *.apk* and obtain the native libraries file *.so*.We use LibDroid to build the ANLD for these Android native libraries. For a large scale of analysis, we created an automated testing script where we set a 60-min timeout for each Android native library. To be more specific, we force the analysis to stop when it reaches the time limit and report the result. According to the result, only 486 native libraries stop early (3.7% of the total number). Note that the 60-min timeout does not include the preprocessing where we transform the *.so* to LLVM IR via McSema (Framework for lifting x86).

Table 4 summarizes our analysis results for each type of the evidentiary data on 13,138 Android Native libraries where we

Table 2
NativeFlowBench evaluation results.

Benchmark App	JN-SAF	FlowDroid	LibDroid
native_source	✔	✘	✔
native_nosource			
native_source_clean		*	
native_leak	✔	✘	✔
native_leak_dynamic_register	✔	✘	✔
native_dynamic_register_multiple	✔	✘	✔
native_noleak			
native_noleak_array	*		
native_method_overloading			
native_multiple_interactions	✔	✘	✔
native_multiple_libraries	✔	✘	✔
native_complexdata	✔	✘	✔
native_complexdata_stringop	*		*
native_heap_modify	✔	✘	✔
native_set_field_from_native	✔✔	✘✘	✔✔
native_set_field_from_arg	✔✔	✘✘	✔✔
native_set_field_from_arg_field	✔✔	✘✘	✔
native_pure	✔	✘	✔
native_pure_direct	✔	✘	✔
native_pure_direct_customized	✔	✘	✔
icc_javatonative	✔	✘	✔
icc_nativetojava	✔	✘	✔

Table 3
Summary of the analysis results for each type of evidentiary data on 52 Android open-source native libraries.The column "Native Library" indicates the number of native libraries. "Others" indicates a file that include the other than the four specified types of evidentiary data.

Evidence Type	Native Library	Evidence File Path	
		Static File Path	Dynamic File Path
Sensor	5	12	0
Time	45	67	2
Text Input	17	16	1
Visited URL	14	32	2
Others	22	40	7

collected from 2,627 real-world popular apps. A reported record contains at least one type of evidentiary data including *Time*, *Text Input*, *Sensor* and/or *Visited URLs*. We categorize a file path as dynamic file path if the file path includes the patterns <*timestamp*>, <*UUID*>, <*android version*> and <*intent*>, otherwise, we treat it as static file paths.

From our analysis result, we find that 87.5% of the file path are static file path which shows static file paths are used more frequently by Android native libraries. Among the four types of evidence, *Time* is the most common evidence that can be retrieved from a mobile device, and 36.6% of Android native libraries store *Time* in the local file system. As for *Sensor* type of evidentiary data, *camera* is the most prevalent *Sensor* requested by application.

Table 4
Analysis results for each type of evidentiary data on the 13,138 native libraries collected from 2,627 real-world apps. The column "App" indicates the number of apps. "Others" indicates a file that include the other than the four specified types of evidentiary data.

Evidence Type	Native Library	App	Evidence File Path	
			Static File Path	*Dynamic File Path*
Sensor	145	35	12	0
Time	1745	398	267	2
Text Input	203	157	161	1
Visited URL	52	14	155	2
Others	2893	514	1723	327

Current state-of-art Android static analysis tool FlowDroid (Arzt et al., 2014) bypass the analysis of native method invocation and apply the conservative approach to propagate the taint. Specifically, for each native method call, FlowDroid (Arzt et al., 2014) make an over-approximation of the taint set for each argument and return value where they take all taint tags from all input variables and assign the union set to each input variables and output variables. On the other hand, FlowDroid (Arzt et al., 2014) only treat a limited number of Java APIs as source and sink methods without the recognition of any native source and sink APIs. In conclusion, FlowDroid (Arzt et al., 2014) is suffered from both under-tainting problem and over-tainting problem. We summarize the results comparing with FlowDroid (Arzt et al., 2014) based on 2,627 real-world apps. 745 of 2,627 apps suffer the under-tainting problem and have a certain type of evidence undocumented. On the other hand, 16.48% of apps are over-tainted due to FlowDroid (Arzt et al., 2014) applying the over-approximation tainting strategy when dealing with Native APIs invocation. Without the data-flow summary of native methods, this approach generates more false positives.

Manual verification: It is very challenging to evaluate our results for the native libraries collected from real-world apps without the *ground truth*. Thus, we perform a best-efforts manual verification. Specifically, we randomly selected 10 applications that utilize 34 native libraries in total. We installed these 10 applications on a real device and leverage Monkey (UI/Application Exerciser Monkey), which can generate pseudo-random streams of user events, such as clicks, touches as well as a number of system-level events, to test each application for 2-h. Then we manually examine each file generated by the apps. In total, we discovered 493 files created by the application and LibDroid reported 23 of them containing evidentiary data.

For a file reported by LibDroid that contain a certain type of evidence, we consider the data-flow summary is a false positive if

that file does not include the type of evidentiary data reported; Take the Fig. 4 as an example, if we manually examine the file */data/data/com.hdtotokmessenger.vdoalexchat/* and cannot find *Time* type of evidence, we will mark this data-flow summary as false positive. Since the evidence may be generated through the Java code of the app itself, it is very hard to define the false negative. Therefore, we only compute the precision for each type of evidence considered in this paper. We find that LibDroid achieves an average precision of 97% over 34 native libraries.

4.4. Case study

We use a case study to demonstrate that our Android Native Library Database can help to identify evidentiary data. To bypass the app security vetting process, malware developers use Command and Control (C&C) server to conceal the malware command and control information generation process into network communication. LibDroid detected a app named *com.gooders.pdfscanner.gpmalware* from *Joker* malware family which hide its C&C communication in the native advertisement libraries.

com.gooders.pdfscanner.gpmalware launches a thread to invoke the native methods in *userServiceStatus* class. While the native methods get access to the user Mobile Country Code (MCC) and send it to the server to get the malicious payload and then subscribe user with premium service.

In addition, it gains the IP address of device through native API *android_getaddrinfofornetwork()*. Then it send the IP address taint source to the third argument of Java method *networkInfoFrom()*, where it store the information at */data/data/com.gooders.pdfscanner.g pmalware/databases/Idata.db*.

LibDroid detects the native source API which queries user MCC code and IP address and generates data-flow summary. LibDroid models the Linux system calls that can execute shell command in which way to track the behaviors of communicating with the C&C server.

5. Related works

5.1. Forensics analysis by static analysis

Among the static analysis tools, Flowdroid (Arzt et al., 2014)is a well known data-flow analysis framework for detecting potential privacy leakage of Android application. It creates a dummy main method for an app, detects and propagates the taints through a flow and context-sensitive algorithm. However, FlowDroid (Arzt et al., 2014) did not cover the native method invocation and treat all native libraries as black box settings. IccTA (Li et al., 2015) extends FlowDroid (Arzt et al., 2014) framework and models regular Intent calls and returns to track data flows.

Chex (Lu et al., 2012) is designed to detect component hijacking problem on Android platform and is the first static analysis tool to consider different types of entry points of an Android app. It is built on top of Wala (Wala) where it first parses app code and constructs app-splits. Each app-splits is a code segment that can be reached from an entry point. Then it builds the data-flow summary for each of the app-split utilizing the data-flow engine from Wala (Wala). Finally, it constructs the possible information flows by linking app-splits summaries in all possible permutations.

DroidSafe (Gordon et al., 2015) proposed a technique to model the Android framework and adopted a flow-insensitive points-to analysis algorithm to handle all possible run-time event ordering.However, DroidSafe (Gordon et al., 2015) fails to analyze real-world apps which cannot be applied to native libraries.

EviHunter leverage Soot (Vall é e-Rai et al., 1999) to perform code lifting from an Android app to Jimple code, IC3 (Li et al., 2015) to build inter-connected component communication models, and FlowDroid (Arzt et al., 2014) to construct call graphs and entry points. It could discover both the types of sensitive data and associated file paths where the data are written to which is suitable for our research purpose. However, it make an over-approximation of evidence types for each input and output variable involved in the native method call.

JN-SAF (Wei et al., 2018) utilize Amandroid (Wei et al., 2014) and Angr (Yan et al., 2016) to capture inter-language data-flow. However, JN-SAF (Wei et al., 2018) only support a limited number of Native source and sink APIs. In addition, it cannot identify the file path where the evidence is written to.

5.2. Forensics analysis by dynamic analysis

TaintDroid (Enck et al., 2014) modified DVM to carry out dynamic taint propagation and uses variable level tracking within the DVM interpreter. However, it could not work with newer version of Android system and only support up to 32 different tags, which is insufficient for the forensic purposes.

NDroid (Qian et al., 2014) is built on the top of QEMU and contains a customized OS-level view reconstructor to obtain information from processes and memory. It instruments important JNI-related, such as JNI entry, JNI exit, object creation functions, and performs dynamic taint analysis tracks information flows through JNI. NDroid re-use the modules implemented by Taindroid and follow their format to store taint tags. However, NDroid suffers from path coverage issue and it does not track control flow.

While TaintART (Sun Tao et al., 2016) applies dynamic taint tracking by instrumentation the ART compiler and runtime since the newer version of Android system replaced DVM since Android Lollipop and uses Ahead of Time Approach (AOT) instead of JIT.

Dyn-EviHunter (Xu et al., 2018) modified the ART platform with taint propagation implementation which force the Android system enter interpreter mode and bypass the checking of trusted mirror class. It support two different modes of operation: Bit-wise mode and Tag-id mode which can be scale up to 2^{32} types of taint tags.

In conclusion, in order to treat file system as a type of sink and further discover the file paths where data are written to, we decided to implement LibDroid which addresses these challenges.

5.3. Library and framework analysis

A series of work has demonstrated the importance of third-party libraries for managed code of Android apps (i.e., Dalvik code) and their security effects and implications (Telecommunication Union, 2017; Alyahya and Kausar, 2017; Satrya et al., 2016; Lessard and Kessler, 2010). (Akarawita et al., 2015; Daryabar et al., 2016; Lee et al., 2009) investigated the outdated libraries in Android apps by conducting a survey with more than 200 app developers. They reported that a substantial number of apps use outdated libraries and that almost 98% of 17K actively used library versions have known security vulnerabilities (Bayu Satrya et al., 2016). report, for managed code-level libraries, that app developers are slow to update to new library versions— discovering that two long-known security vulnerabilities remained present in top apps during the time of their study. None of these studies examined native third-party libraries in Android apps nor did they look at the security impact of vulnerable libraries or whether these vulnerabilities are on the attack surface. LibRARIAN now explores the attack surface of native libraries, closing this important gap and calling platform providers to action.

6. Discussion and limitations

At this point, native library matching is based on the package

name and version number. Several researches (Li et al., 2017; Zhang et al., 2019; Ma et al., 2016) proposed and implemented different approaches to identify specific versions of third-party libraries from apps through comparison of call graph or static analysis of app binaries coupled with a database of third-party libraries. Since the library matching is beyond the scope of our project, and existing works can easily solve this matching problem, we will work on integrate LibDroid with library matching tool to achieve better accuracy on library identification.

As many other Android static analysis tools, LibDroid has limitations on dynamic file path construction. Specifically, if a dynamic file path includes intent from inter-component communication, LibDroid uses <intent> instead of the concrete value as part of the file path.

LibDroid only supports constant string propagation, if the string is manipulated, LibDroid will not be able to construct the precise resolution of a string value. As precise string analysis is expensive in binary analysis, we leave this for future research.

Moreover, LibDroid only has basic support for system native APIs. In particular, we manually summarize the data flows for system native APIs that are related to file path constructions and commonly used data structures. It would be an interesting future work to model all system native APIs.

There are many methods to obfuscate native libraries in Android in order to evade the security vettings, such as string encryption and control flow obfuscation. The latter re-organizes the control flow of the native method, injectes dead/ dummy code and removes functions' prototypes, and uses proxy methods to redirect the flow of execution. LibDroid currently does not provide a solution for such obfuscations. We will apply anti-obfuscation techniques (Baumann et al., 2017) in the future.

7. Conclusion

In this work, we design LibDroid to automatically identify the evidentiary data generated and stored on the file system by the Android native library. LibDroid builds an App Native Library Database (ANLD) of a large number of Android native libraries collected from real-world apps by applying the forward static data flow analysis. In addition, app developers and researchers of the existing static analysis tools could leverage the data-flow summary to compute the analysis sensitive to the native method invocation. We summarize the data-flow of commonly used system native APIs, and extend the current publicly available sources and sinks with native APIs. Our evaluations, based on both open-source native libraries and native libraries collected from real-world apps, show that LibDroid can precisely and accurately identify the evidentiary data that a native library could store to file systems and the file paths where the evidentiary data are written to.

Acknowledgment

This work was partially supported by NIST CSAFE under Cooperative Agreement No. 70NANB20H019, NSF under grants No. CNS-1619201, CNS-1730275, DEB-1924178, ECCS-2030249, and Boeing Company. We appreciate anonymous reviewers for their valuable suggestions and comments.

References

[n. d.] 30 Popular mHealth Apps Vulnerable to API Attacks. Posing PHI risk. https://30-popular-mhealth-apps-vulnerable-to-api-attacks-posing-phi-risk.
Akarawita, Indeewari U., Perera, Amila B., Atukorale, Ajantha, 2015. ANDROPHSY-forensic framework for Android. In: Advances in ICT for Emerging Regions (ICTer), 2015 Fifteenth International Conference on. IEEE, pp. 250–258.
Alyahya, Tadani, Kausar, Firdous, 2017. Snapchat analysis to discover digital forensic artifacts on android smartphone. Procedia Comput. Sci. 109, 1035–1040, 2017.
[n. d.] Apkpure. https://apkpure.com/.
Arzt, Steven, Bodden, Eric, 2016. StubDroid: automatic inference of precise data-flow summaries for the android framework. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pp. 725–735. https://doi.org/10.1145/2884781.2884816.
Arzt, Steven, Rasthofer, Siegfried, Fritz, Christian, Bodden, Eric, Bartel, Alexandre, Klein, Jacques, Le Traon, Yves, Octeau, Damien, McDaniel, Patrick, 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. SIGPLAN Not 49 (6), 259–269. https://doi.org/10.1145/2666356.2594299 (June 2014).
Baumann, Richard, Protsenko, Mykolai, Müller, Tilo, 2017. Anti-ProGuard: towards automated deobfuscation of android apps, pp. 7–12. https://doi.org/10.1145/3099012.3099020.
Bayu Satrya, Gandeva, Tobianto Daely, Philip, Arif Nugroho, Muhammad, 2016. Digital forensic analysis of telegram messenger on android devices. In: Information & Communication Technology and Systems (ICTS), 2016 International Conference on. IEEE, pp. 1–7.
Calzavara, Stefano, Grishchenko, Ilya, Maffei, Matteo, 2017. HornDroid: practical and sound static analysis of android applications by SMT solving. CoRR abs/1707.07866. arXiv:1707.07866. http://arxiv.org/abs/1707.07866, 2017.
Cheng, Chris Chao-Chun, Shi, Chen, Gong, Neil Zhenqiang, Guan, Yong, 2018. EviHunter: identifying digital evidence in the permanent storage of android devices via static analysis. CoRR abs/1808.06137. arXiv:1808.06137 URL. http://arxiv.org/abs/1808.06137, 2018.
[n. d.] Clang Static Analyzer. https://clang-analyzer.llvm.org/.
Daryabar, Farid, Tadayon, Mohammad Hesam, Parsi, Ashkan, Sadjadi, Hadi, 2016. Automated analysis method for forensic investigation of cloud applications on Android. In: Telecommunications (IST), 2016 8th International Symposium on. IEEE, pp. 145–150.
[n. d.] DroidBench benchmark. https://blogs.uni-paderborn.de/sse/tools/droidbench/.
Enck, William, Gilbert, Peter, Han, Seungyeop, Tendulkar, Vasant, Chun, Byung-Gon, Cox, Landon P., Jung, Jaeyeon, McDaniel, Patrick, Sheth, Anmol N., 2014. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans. Comput. Syst. 32 (2). https://doi.org/10.1145/2619091. Article 5 (June 2014), 29 pages.
[n. d.] Framework for lifting x86. amd64, aarch64, sparc32, and sparc64 program binaries to LLVM bitcode. https://github.com/lifting-bits/mcsema-legacy.
[n. d.] Google Play Store. https://play.google.com/store.
Gordon, Michael I., Kim, Deokhwan, Perkins, Jeff H., Gilham, Limei, Nguyen, Nguyen, Rinard, Martin C., 2015. Information Flow Analysis of Android Applications in DroidSafe. NDSS, Citeseer.
[n. d.] How your search history can send you to jail. https://www.startpage.com/privacy-please/startpage-articles/how-your-search-history-can-send-you-to-jail.
[n. d.] JD Finance Android App. https://jd-finance-android-app-stored-users-screenshots-without-permission.
Lee, Xinfang, Yang, Chunghuang, Chen, Shihjen, Wu, Jainshing, 2009. Design and implementation of forensic system in Android smart phone. In: The 5th Joint Workshop on Information Security.
Lessard, Jeff, Kessler, Gary, 2010. Android Forensics: Simplifying Cell Phone Examinations, 2010.
Li, Li, Bartel, Alexandre, Bissyandé, Tegawendé F., Klein, Jacques, Le Traon, Yves, Arzt, Steven, Rasthofer, Siegfried, Bodden, Eric, Octeau, Damien, McDaniel, Patrick, 2015. IccTA: detecting inter-component privacy leaks in android apps. In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, Piscataway, NJ, USA, pp. 280–291. http://dl.acm.org/citation.cfm?id=2818754.2818791.
Li, Menghao, Wang, Wei, Wang, Pei, Wang, Shuai, Wu, Dinghao, Liu, Jian, Xue, Rui, Huo, Wei, 2017. LibD: scalable and precise third-party library detection in android markets. In: *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, pp. 335–346. https://doi.org/10.1109/ICSE.2017.38.
Lu, Long, Li, Zhichun, Wu, Zhenyu, Lee, Wenke, Jiang, Guofei, 2012. CHEX: statically vetting android apps for component hijacking vulnerabilities. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security (Raleigh, North Carolina, USA) (CCS '12). ACM, New York, NY, USA, pp. 229–240. https://doi.org/10.1145/2382196.2382223.
Ma, Ziang, Wang, Haoyu, Guo, Yao, Chen, Xiangqun, 2016. LibRadar: fast and accurate detection of third-party libraries in android apps. In: 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C), pp. 653–656, 2016.
[n. d.] Mi App Market. https://m.app.mi.com/.
Moura, L.D., Bjørner, N., 2008. Z3: an Efficient SMT Solver. TACAS.
[n. d.] NativeFlowBench. https://github.com/arguslab/Argus-SAF/tree/master/benchmarks/NativeFlowBench.
Qian, Chenxiong, Luo, Xiapu, Shao, Yuru, Chan, Alvin, 2014. On tracking information flows through JNI in android applications. In: Proceedings of the International Conference on Dependable Systems and Networks, pp. 180–191. https://doi.org/10.1109/DSN.2014.30.
Satrya, G.B., Daely, P.T., Shin, S.Y., 2016. Android forensics analysis: private chat on social messenger. In: Ubiquitous and Future Networks (ICUFN), 2016 Eighth International Conference on. IEEE, pp. 430–435.
Sun, Mingshen, Tao, Wei, John, C., Lui, s, 2016. TaintART: A Practical Multi-Level

Information-Flow Tracking System for Android RunTime, pp. 331–342. https://doi.org/10.1145/2976749.2978343.
[n. d.] SuSi. Source and sink. https://blogs.uni-paderborn.de/sse/tools/susi/.
Telecommunication Union, International, 2017. Measuring the Information Society Report 2017, p. 1, 2017.
[n. d.] UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.
Vallée-Rai, Raja, Co, Phong, Gagnon, Etienne, Hendren, Laurie, Lam, Patrick, Sundaresan, Vijay, 1999. Soot - a java bytecode optimization framework. In: Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (Mississauga, Ontario, Canada) (CASCON '99), vol. 13. IBM Press. http://dl.acm.org/citation.cfm?id=781995.782008.
[n. d.] Wala. Static analysis capabilities for java bytecode and related languages. https://researcher.watson.ibm.com/researcher/view_page.php?id=7238.
Wei, Fengguo, Roy, Sankardas, Ou, Xinming, Robby, 2014. Amandroid: a precise and general inter-component data flow analysis framework for security vetting of android apps. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. ACM, New York, NY, USA, pp. 1329–1341. https://doi.org/10.1145/2660267.2660357.
Wei, Fengguo, Lin, Xingwei, Ou, Xinming, Chen, Ting, Zhang, Xiaosong, 2018. JN-SAF: precise and efficient NDK/JNI-Aware inter-language static analysis framework for security vetting of android applications with native code. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, pp. 1137–1150. https://doi.org/10.1145/3243734.3243835.
Xu, Zhen, Shi, Chen, Cheng, Chris, Gong, Neil, Guan, Yong, 2018. A dynamic taint analysis tool for android app forensics, pp. 160–169. https://doi.org/10.1109/SPW.2018.00031.
Yan, Shoshitaishvili, Wang, Ruoyu, Salls, Christopher, Stephens, Nick, Polino, Mario, Audrey Dutcher, Grosen, John, Feng, Siji, Hauser, Christophe, Kruegel, Christopher, Vigna, Giovanni, 2016. SoK: (state of) the art of war: offensive techniques in binary analysis. In: IEEE Symposium on Security and Privacy.
Zhang, Jiexin, Beresford, Alastair R., Kollmann, Stephan A., 2019. LibID: Reliable Identification of Obfuscated Third-Party Android Libraries. Association for Computing Machinery, New York, NY, USA, pp. 55–65. https://doi.org/10.1145/3293882.3330563.