ApakoHa: AN AUTOMATED TOOL USING DYNAMIC TAINT ANALYSIS FOR ANDROID SECURITY FOCUSING ON SENSITIVE DATA

A PREPRINT

Yiwei Yang, Longwen Zhang, Kaiyuan Xu, Zhe Ye
Schools of Information and Science Technology
ShanghaiTech University
Shanghai, SH 201210
yangyw,zhanglw2,xuky,yezhe@shanghaitech.edu.cn

January 5, 2020

ABSTRACT

Privacy protection on android phones is a widely discussed topic nowadays. As the main leaking source, many tools analyzing information flow statically and dynamically. Integrating dynamic taint analysis in the development process enables early detection of potential privacy leakage, which reduces the cost of fixing them. In this paper, we present *ApakoHa*, a dynamic taint analysis tool for Android apps that interleaves bug fixing and code development in the VS-code integrated development environment. *ApakoHa* is based on the novel framework of TaintART that makes full use of android ART runtime to get information flow, and computes the more complex results and optimizes the bytecode through soot**??**. incrementally later using static analyzing tools. Unlike traditional batch-style static-analysis tools, *ApakoHa* causes minimal disruption to the developer's workflow. This video demo showcases the main features of *ApakoHa*:

***K*eywords** DTA · Android privacy · automated tool · operating system

1 Introduction

Android security has attracted much research attention from both academy and industry recently. Dynamic Taint Analysis(DTA) is a classic analysis to detect information flow problems and it has been widely adopted to detect private data leaks in Android applications. In this project, we will automate the process of taint analysis and provide more detailed information about the dataflow and ICFG on the source code. Thus providing a way for Maple IR to continue to compile.

First, in terms of the performance of the DTA, we refer to the TaintART techniques to utilize the compiler and the register allocation of android ART Runtime. Then, a taint propagation framework is proposed and the correctness of the taint propagation analysis is proved by their paper. After we obtain the information flow and what kind of method the information is leaking, we try to make highlight the code on VS-Code front end, Finally, in the backend, the function name and the taint source will be input to soot, a java static analyzing tool to output ICFG and optimize the code automatically by adopting the methods of eliminating, replacing and moving.

We're actually adopting the method of combing the advantages of dynamic and static taint analysis techniques. Through static analysis can sort out the general idea, through dynamic analysis can get the actual execution process of the program. Both of them can help each other to realize the deep penetration test of the responsible app. The pros and cons of the state of art are listed as follows:

Table 1: The comparison of the state of the art android taint analysis

Type	Representative works		Features
Static	In-component propagation analysis	LeakMiner/CHEX	Incrementally add callback function Implementing the Android semantic equivalent model Add processing callback function Virtual access point
	Inter-component propagation analysis	Klieber Heros/DroidSafe	Match with custom inference rules Transform data flow analysis to improve accuracy
	Component and library function propagation analysis	FlowDroid	Manual analysis and Implementation Automatic derivation with flowdroid
Dynamic	Multi level propagation analysis	TaintDroid Appsplaygroud...	Address multiple levels of communication strategy Optimization or application extension

2 Related workflow

2.1 Taintdroid

TaintDroid 4.3 is a system-wide taint tracking system based on Android 4.2+. It aims to minimize runtime overhead that is the amount of additional instructions needed for the implementation of tracking mechanism and, also to monitor the system for sensitive information leakage.

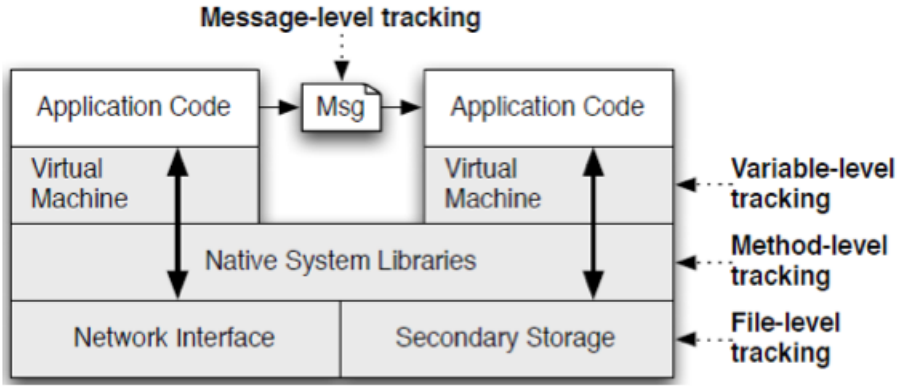

Figure 1: TaintDroid Multi-level tracking.

As shown in Figure 2, the system tracks information at multiple levels. The variable-level tracking means the information flow among memory like stack register and heap object. The method-level tracking means the whenever a method returns, the return value if any should properly propagate to the caller method. The file-level tracking means that the system store taint tag on file system permanently. The message-level tracking means the information flow between processes. An Android application typically uses the broadcast receiver and intent to communicate with each other .

TaintDroid uses technique call adjacent memory storage for the taint storage. This means taint tag locates next to the memory which the tag is associated with. In theory, this implementation should double both the total memory and the address of object. This eases the calculation of the address of the taint tag.

The taint tag used by TaintDroid is bit-wise, meaning each bit of the allocated memory represents the absence or presence of certain type information. The taint tag is stored in a 32-bit integer. As a result, only 32 types of information can be distinguished. There are several modifications made by TiantDroid developers for the accommodation of taint tag.

1. Stack. Next to each virtual register which is just a 4-byte memory, an additional 4-byte memory is allocated so that during the execution of a method, the taint tag can be stored locally for the method.

2. Calling convention. When a method is invoked or returned, a modified calling convention is used to accommodate the taint tag for the return result.

3. Parcel. This change allows taint tag travel through binder which essentially the inter-process communication procedure.

4. File system. TaintDroid extends the Linux file system so that the metadata on INode can store the taint tags. If such way, a file which receives sensitive information can be marked accordingly. In such way, taint tag can be stored permanently.

With the propagation rules, taint tag can travel through the application and system and when the information flows to the predefined method or sink, the situation is reported through a special channel and revealed by a front application. For example, a binary operation like addition which takes in virtual register A and B and output to virtual register C. The resulted taint tag of register C should the union of taint tags from register A and B. The union operation in TaintDroid is used as binary "OR" operation. The Table 1 comes from TaintDroid and shows the propagation rules.

Op Format	Op Semantics	Taint Propagation	Description
const-op v_A C	$v_A \leftarrow C$	$\tau(v_A) \leftarrow \emptyset$	Clear v_A taint
move-op v_A v_B	$v_A \leftarrow v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
move-op-R v_A	$v_A \leftarrow R$	$\tau(v_A) \leftarrow \tau(R)$	Set v_A taint to return taint
return-op v_A	$R \leftarrow v_A$	$\tau(R) \leftarrow \tau(v_A)$	Set return taint ($\emptyset$ if void)
move-op-E v_A	$v_A \leftarrow E$	$\tau(v_A) \leftarrow \tau(E)$	Set v_A taint to exception taint
throw-op v_A	$E \leftarrow v_A$	$\tau(E) \leftarrow \tau(v_A)$	Set exception taint
unary-op v_A v_B	$v_A \leftarrow \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
binary-op v_A v_B v_C	$v_A \leftarrow v_B \otimes v_C$	$\tau(v_A) \leftarrow \tau(v_B) \cup \tau(v_C)$	Set v_A taint to v_B taint $\cup$ v_C taint
binary-op v_A v_B	$v_A \leftarrow v_A \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_A) \cup \tau(v_B)$	Update v_A taint with v_B taint
binary-op v_A v_B C	$v_A \leftarrow v_B \otimes C$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
aput-op v_A v_B v_C	$v_B[v_C] \leftarrow v_A$	$\tau(v_B[\cdot]) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_A)$	Update array v_B taint with v_A taint
aget-op v_A v_B v_C	$v_A \leftarrow v_B[v_C]$	$\tau(v_A) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_C)$	Set v_A taint to array and index taint
sput-op v_A f_B	$f_B \leftarrow v_A$	$\tau(f_B) \leftarrow \tau(v_A)$	Set field f_B taint to v_A taint
sget-op v_A f_B	$v_A \leftarrow f_B$	$\tau(v_A) \leftarrow \tau(f_B)$	Set v_A taint to field f_B taint
iput-op v_A v_B f_C	$v_B(f_C) \leftarrow v_A$	$\tau(v_B(f_C)) \leftarrow \tau(v_A)$	Set field f_C taint to v_A taint
iget-op v_A v_B f_C	$v_A \leftarrow v_B(f_C)$	$\tau(v_A) \leftarrow \tau(v_B(f_C)) \cup \tau(v_B)$	Set v_A taint to field f_C and object reference taint

Figure 2: TaintDroid Propagation Rules.

TaintDroid is strong for the runtime efficiency. The statistic shows only 14% runtime overhead occurring during testing. The drawback, however, is that the system is based on Android version 4.2 which is outdated compared today's version 7.0. TaintDroid cannot distinguish files but only label all files under one type of information.

2.2 Cheetah

Despite the demonstrated usefulness of DTA in mobile privacy security, poor performance attained by prototypes is a big problem. A novel optimization methodology for taint tracking based on just-in-time static analysis taht discovers and reports the most relevant results to the developer fast is presented. Cheetah4.3 is a JIT taint analysis for Android applications applying this theory. It enables easy transformation of a distributive dataflow analysis to its correstponding JIT with minimal changes to its fransfer functions. A JIT analysis computes the same dataflow propagations as its base analysis, but it delays some propagations in favor of others by pausing and resuming them later at trigger $stmt$, and each of them is assorciated with a priority that determines the layer at which the JIT analysis resumes its computation.

However, their work is not compatible with android development for from android 7.0 AoT take the place of JIT.

2.3 Inspeckage

Inspection is an Xposed module for dynamic analysis of Android apps. Inspection package summarizes many common functions of dynamic analysis and builds a web server. The whole analysis operation can be carried out in a friendly interface environment.

In theory, it can get all the information by add hooks, but it requires Xposed, which is depreacated since android 7.0.

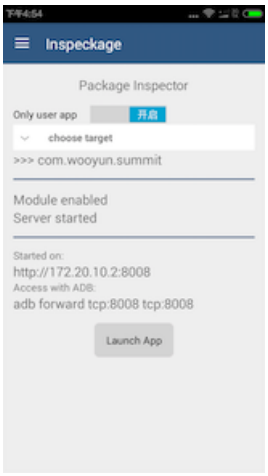

Figure 3: Inspeckage analyzing background.

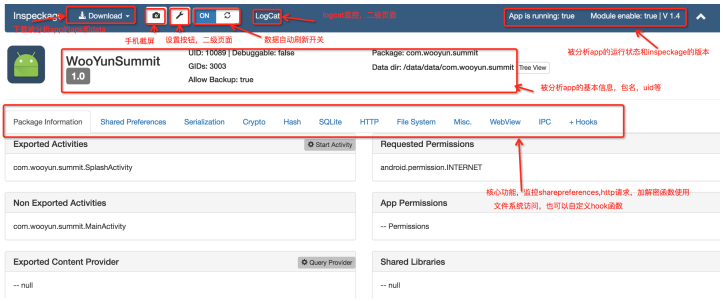

Figure 4: Inspeckage webserver.

3 Theory Principles

In this part, we'll introduce how the dynamic taint analysis work in tracking the private information and how we find the ICFG graph for those who have possibility to leak information in dynamic runtime which may save a lot of time analyzing non-triggered leakage situation.

3.1 Principles of finding Information flow in DTA

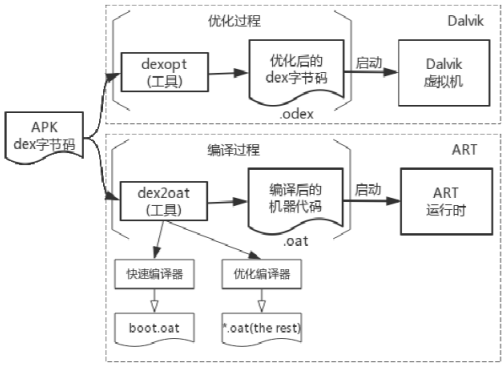

Figure 5: TaintART ART runtime.

The main difference between DTA and STA is that DTA has to deal with the runtime. Somtimes hardware support is required. But in android, all the register allocation is based on ART runtime.

For DTA, the sensitive data to be tracked is called taint source, and the tag added to track sensitive data is called taint tag. When sensitive data is copied or moved to another new location, or data is cleared, its tag will also be copied, moved or cleared, which is called taint logic In taint logic, the status of taint tags that track sensitive data will be stored in taint tag storage. DTA will track the tainted sensitive information and detect whether it is sent out of the system through certain functions, which are called taint sink.

3.1.1 Runtime support

The feature of TaintART is that it uses CPU register to store the taint tags. The taint tag is either 1 or 2 bits wide and represents the taint status of other processor registers. During the compilation, the program reserves register 5 and treats it as storage. The register 12 is used for taint propagation.

3.1.2 Taint Source

Level	Leaked Data	Source	Class/Service
0 (00)	No Leakage	N/A	N/A
1 (01)	Device Identity	IMSI	TelephonyManager
		IMEI	TelephonyManager
		ICCID	TelephonyManager
		SN	TelephonyManager
2 (10)	Sensor Data	Accelerometer	SensorManager
		Rotation	SensorManager
	Location Data	GPS Location	LocationManager
		Last Seen Location	LocationManager
		Network Location	LocationManager
3 (11)	Sensitive Content	SMS	ContentResolver
		MMS	ContentResolver
		Contacts	ContentResolver
		Call log	ContentResolver
		File content	File
		Camera	Camera
		Microphone	MediaRecorder

Figure 6: TaintART Taint Source.

3.1.3 Taint Propogation

The propagation rules of TaintART shares some similarities with the one from TaintDroid 2. The major difference is TaintART's rule focus on the instruction at compilation level. HInstruction is a class used by Android for the representation of each Dalvik instruction. The TaintART have rules for every HInstruction so that the resulted binary code can achieve taint propagation as desired. The second major difference is that the unlike binary or operation used by TaintDroid, the TaintART uses a special information merging operation. Take 2-bit wide storage for example. Suppose a taint tag of 0x01 needs to be merged with a taint tag of 0x010, the result would be 0x10. This is because the TaintART uses a level design. When a low-level taint tag encounters a higher-level taint tag, the result would simply be the higher one. The choice is made due to the privacy and runtime efficiency focus.

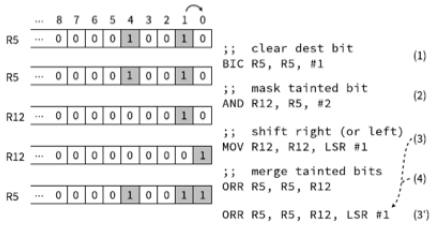

Figure 7: TaintART Taint Propogation.

Figure 7 comes from TaintART and shows the operations done for the instruction "MOV R0, R1". The initial taint status represented by R5 is that register 1 and 4 are tainted. The result of MOV operation should leave register 0 tainted.

The instruction #1 masks bit on index 1 which represents the taint tag for register 1 and store the value into temporary register 12 in Instruction #2. The instruction #3 shifts the bit to the appropriate position according to the index of result register, which is 0 in the case. Finally, a binary OR operation with input from R5 and R12 gives the result which is then put back to R5. The TaintART modifies the heap to accommodate the taint tag from R5. When a method is invoked, the content of register 5 is saved to stack and then loaded with values according to the taint tags of argument registers.

Under the taint propagation logic, each taint propagation statement is associated with a taint value, which refers to the abstract representation of all pollution state sets that can be observed at some place. The set of all possible taint values is called the taint propagation value range, which is studied abstractly in a holistic way. The taint value range is a product lattice, and the lattice corresponding to each taint variable is shown in Figure 8.

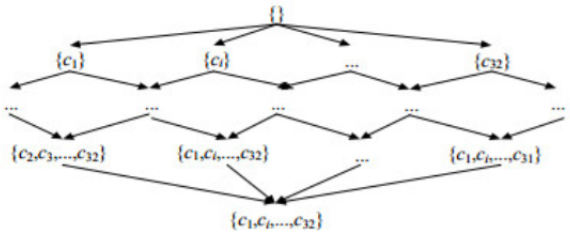

Figure 8: Lattice of taint propagation range.

For any value x belongs to the value set V, the intersection operation $\cup$ takes the set combination set $\cup$, which includes:

(1) x $\wedge$ x = x, which satisfies the idempotency;

(2) x $\wedge$ y = y $\wedge$ x, which satisfies the exchangeability;

(3) x $\wedge$ (y $\wedge$ z) = (x $\wedge$ y) $\wedge$z, which satisfies the Associativity.

The top element is the empty set $\emptyset$, which is expressed as t, which has t$\wedge$x = x for all x in V; the bottom element is the complete set u, which is expressed as $\perp$, which has $\perp \wedge$ x = $\perp$, for all x in V.

In the taint propagation logic, each statement stat corresponds to a transfer function FS, and the transfer function of a block containing multiple statements can be constructed by combining the transfer functions corresponding to each statement. The function set F is composed of a set of transfer functions F_S, whose input is the mapping $in[S]$ of the stain variable to the element in the lattice, and the input is a new mapping $out[S]$ after the stain propagation In forward stain propagation analysis, the transfer function F_S takes $in[S]$ before the statement as the input and outputs $out[S]$ after the statement; the transfer function f is closed for combination operation, that is, for any function f and G in F, the function h with $H(x) = g(f(x))$ is also in F. there is a unit function I in the transfer function family F, which accepts a mapping as the same mapping returned by the input and output, that is, for All x in V, $I(x) = x$

Monotonicity is defined as:

(1) for all F and X and Y in all V, $f(x \wedge y) \leq f(x) \wedge f(y)$.

Monotonicity can also be equivalently defined as:

(2) for all F and X and Y in all V, $X \leq y$ implies $f(x) < f(y)$. It is proved that the two definitions are equivalent

It is proved that monotonicity(2) can be derived from monotonicity(1): since $x \wedge y$ is the maximum lower bound of X and y,$X \wedge y \leq X$ and $X \wedge y \leq y$, from monotonicity(2), $f(x \wedge y) \leq f(x)$ and $f(x \wedge y) \leq f(y)$; at the same time, $f(x \wedge y)$ is the maximum lower bound of $f(x)$ and $f(y)$, monotonicity(1) is proved

Then, it is proved that monotonicity(1) can deduce monotonicity(2): suppose $x \leq y$, from monotonicity(1), $f(x \wedge y) \leq f(x) \wedge f(y)$, according to the definition, $x \wedge y = x$. therefore, $f(x) < f(x) \wedge f(y)$. Because $f(x \wedge y)$ is the maximum lower bound of $f(x)$ and $f(y)$, $f(x) \wedge f(y) < f(y)$, that is, $f(x) < f(y)$, monotonicity(2) is proved

The lattice8 shown clearly satisfies monotonicity (2) definite For any set X and Y, X belongs to $X \cup Y$.

3.1.4 Taint sink

The main choice of the taint slot is the exit function related to the network, and it can provide the log function. The log function is implemented by JNI.

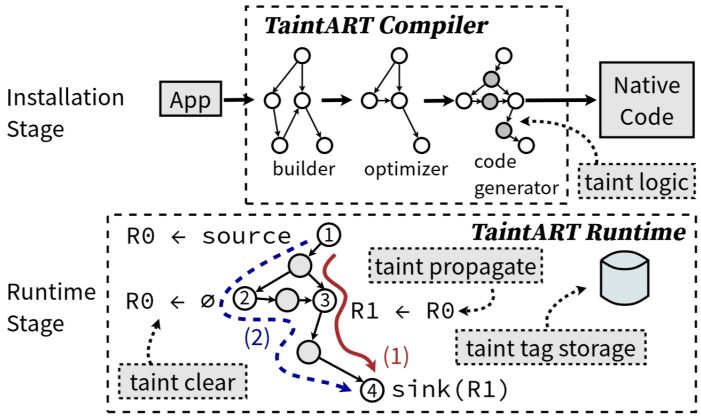

Figure 9: Overview of TaintART.

3.1.5 Taint tag storage

Another feature of TaintART is its taint tag representation. Instead of the taint tag used by TaintDroid, the TaintART uses the concept of level. Figure **??** shows the table from TaintART. There are in total 4 levels where each of them consists some kind of data. Take level 2 as example. It represents both sensor data and location data. The system does not tell what exactly the information a variable carries. If a variable with level 2 information merge with a variable with level 1 information, the result will carry a level 2 tag. This is not desirable for the forensic purpose because of the need to tell the information flow as accurate as possible.

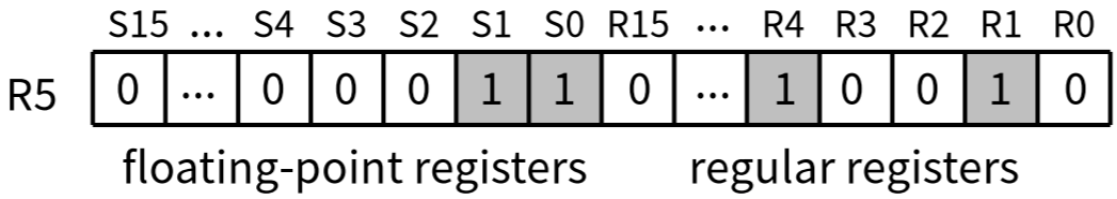

Figure 10: Taint tag storage using Register R5.

3.2 Principles of finding ICFG

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

3.2.1 Taint sink

3.3 Principles of finding IFDS

4 Application Implementation

4.1 Useful Tools introduction

4.1.1 Soot

Soot is a project with a long history. Its main purpose is to translate Java bytecode and DEX bytecode into an intermediate language. There are four ways to express the output results. Here, we mainly use jimple, which is very similar to Java. It uses less information to express the relationship between classes and variables. It can be described with only 15 opcodes.

Soot is the foundation of the whole static analysis framework. It can disassemble APK files. How to realize it in the middle doesn't need to be concerned for the time being. What the code finally gets is the middle representation. Through the API, you can access the members of each jimple, the call relationship, etc. according to this relationship, you can also generate a visible picture of CFG (control flow graph), which looks comfortable.

The documentation for `Soot` may be found at `https://github.com/Sable/soot`

4.2 Workflow

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

4.2.1 benchmark

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

4.3 References

[Sun2016CCS] A Practical Multi-level Information-Flow Tracking System for Android RunTime

[ABDE2019WCNCW] Android Malware Detection Based on System Calls Analysis and CNN Classification

[Xu2018SPW] A Dynamic Taint Analysis Tool for Android App Forensics

[Yerima2014NGMAST] Android Malware Detection Using Parallel Machine Learning Classifiers

[Hsiao2014IMIS] PasDroid: Real-Time Security Enhancement for Android

[Ngu2017ICSE] Cheetah: Just-in-Time Taint Analysis for Android Apps

[Arzt2014SIGPLAN] FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps