A dynamic taint forensic analysis tool for Android apps

by

Zhen Xu

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering (Secure and Reliable Computing)

Program of Study Committee:
Yong Guan, Co-Major Professor
Zhenqiang Gong, Co-Major Professor
Ying Cai

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this thesis. The Graduate College will ensure this thesis is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2017

DEDICATION

I would like to dedicate this thesis to my parents who always have faith in me and support me throughout my journey, and as well to my roommate who is of extreme annoyance just by her presence especially with the cats meowing around.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to send my best regards to Professor Guan for his patient guidance. During the project, there were a time of frustration but he never lost faith in me. His guidance will long be remembered. Special thanks to my other two committee members Dr. Gong and Dr. Cai for advising me during the project.

Also, I would like to thank my colleagues for their support and acknowledge their contribution. Chen Shi helped me with her testing skill and conduct numerous testing. Wenhao Chen lent me his homebrew symbolic execution engine so that the system can have a certain feature.

Finally, I would like to thank my previous advisor Morris Chang for bring me into this field.

ABSTRACT

Mobile digital forensic faces numerous problems including a huge amount of data, growing amount of applications and usage of encryption or obfuscation. As a result, data of interest is hard to locate. The traditional method uses predefined pattern searching algorithm. Such technique can dig out much information but cannot find information embedded with normal data such as a barcode in an image or encryption. This project intends to develop a tool which facilitates the investigation process by answering what information could exist in a certain file. With the assistance, the investigator can focus on the content of some interesting files instead of enumerating all of them. The tool takes in and analyzes an application. The outputs consist of a table of files and their known content. The technique used in the project is called dynamic taint analysis and the implementation is based on Android OS 7.0. The prototype of the system has been implemented and two modes of the system are provided. One focuses on runtime efficiency and the other focuses on distinguishing as many information as possible. Experiments were conducted on testing apps, well-known social apps and the ones from an app pool. The finding indicates the system can fulfill its goal by detecting information flow to files.

CHAPTER 1 INTRODUCTION

One of the important aspects of digital forensics is to extract evidence from a device under investigation. The first step, however, is to figure out where a certain evidence could be. Traditional approach uses predefined or user-defined pattern searching algorithm which enumerates a set of data and finds what matches. Yet the data of interest might hide inside some files so that it is hard to notice via traditional method. For example, some 2-D Barcode [1] can be embedded into an image so that it is nearly impossible to tell if so by simply looking at the byte pattern or the image as a whole. Also, information like GPS can be easily mixed with other information into some integer-alphabet mix which is hard to distinguish.

Those cases require a new approach. One way to figure out what is inside a file is to examine the application which generates it. And for the examples presented previously, the taint tracking is selected because of the simplicity compared to other program analysis technique like symbolic execution. In general, there are two categories: one is static and the other is dynamic. The static approach of taint tracking analyzes the data flow by enumerating over all possible execution branches. This could lead to path explosion which affects scalability so that real-world application like Facebook which is just too large for analysis. Also, the result often sits on large quantity with high false positives rate. The dynamic approach on the other hand if implemented properly can run a large application and at least extract some result with a low false positive rate. The low false positive rate is important in the context of forensic because if an evidence with high false positive rate can simply not be taken to the court.

The goal of this project is to implement the dynamic taint tracking. The target platform is Android OS for its open source feature and, also the Android OS has the largest share of mobile device market around 65% [2]. The implementation consists of two modes. One focuses on the runtime efficiency and the other focuses on distinguishing as many types of information as possible. The system is dynamic analysis tool. It requires additional event inputs in order to explore the application. This is done either by manual input or by the machine generated events. The intended usage of the product is as following: identify the application under investigation; run the application on a host machine with the dynamic taint system; use either manual input or automatically generated event to explore the application. The result then can be used to indicate what information presents in a certain file or what information flows into a predefined place.

This thesis can be mainly divided into two parts. The first part discusses the dynamic taint analysis from the theory to the implementation. The second part talks about the event sequence generation accompanies the system.

CHAPTER 2 RELATED WORK

This chapter examines existing projects of similar approach and explains the difference between this project vs others and, also the difference of requirement. For each related work, there are several elements in common: taint representation, taint storage, and taint propagation. Those elements are the basic element of taint analysis.

2.1 TaintDroid

TaintDroid [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.

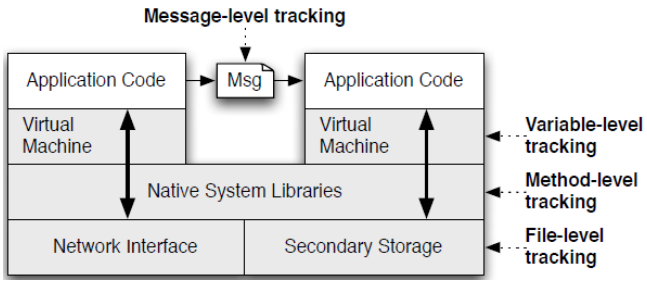

Figure 1 TaintDroid Multi-level tracking

As shown in Figure 1, 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.

Table 1 TaintDroid 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

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 TaintART

TaintART [4] shares similar goal with TaintDroid as both focus on information leakage and minimal runtime overhead. Unlike TaintDroid, TaintART builds upon Android version 7.0 which is one of latest and popular platform in the ecosystem. The implementation of TaintART alters compiler to achieve the dynamic taint analysis. 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. The propagation rules of TaintART shares some similarities with the one from TaintDroid. 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.

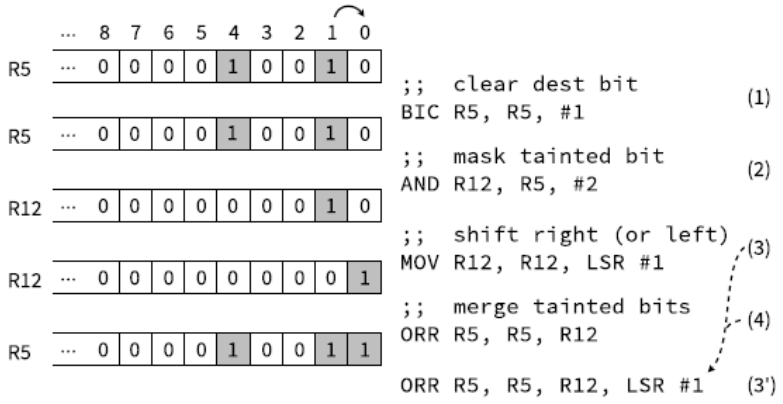

Figure 2 TaintART Propagation Example

Figure 2 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.

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 3 TaintART Multi-level Tag

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 3 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.

Overall, TaintART achieves its goal with a penalty from 1% to 30%. The design of the system does not meet our need due to the loss of accuracy of information flow. Only 4 levels or types are effectively distinguished. Thus it cannot be used by our project.

2.3 ARTist

ARTist [5] is a compiler-based instrumentation tool. Android uses Dex2oat as the on-site compiler which takes in an Android application APK and converts it into an oat file. The oat file contains both the byte code and binary code which is ready for execution. The dex2oat operates on objects called HGraph which represents a method in the application. ARTist integrates its instrumentation module with the compiler and thus allows compiler-based instrumentation.

The ARTist can be used to implement taint tracking by instrumenting the application. After the package is unzipped and the intermediate representation of the HGrpah is built, several steps occur:

1. Identify calls to global source and sink methods which are defined by the user.
2. Identify local sources and sinks which include method arguments and parameters, method invocation and return, and field read and write operations.
3. Create intra-procedural data flows. Such data flow is the result of backward tracing which starts at a sink method. During the tracing, all variables that might influence the sink are recorded.

4. Instrument the application. The ARTist handle taint propagation via a companion library. Such library provides static Java methods to record and access the taint tag for the method-level propagation. Namely the methods are taint-get, taint-set, getArgTaint and setArgTaint. When a taint-get method is invoked, the program assumes the occurrence of taint-set method. For field operations, it uses predefined constant value for the static field as an identifier while for a field of an instance, a constant plus the whole object are provided for the identification of field.

In summary, the ARTist is a light-weight multi-purpose instrumentation tool which can be used for taint tracking, policy enforcement, and others. The drawback of the solution is the potential incorrectness raised from the calculation of identifier which exists for tainted variables and is used for finding the taint tag. The calculation takes in a predefined constant which is inserted during compilation and a result from a hash function. This identifier might duplicate.

CHAPTER 3 DYNAMIC TAINT ANALYSIS

This chapter digs into the detail from theory to implementation. The first part compares different approaches like taint analysis and symbolic execution. The second part introduces the background of Android OS and the underlying components of relevance. The third part describes the theory of dynamic taint analysis. The final part lists all the implementation based on theory.

3.1 Technique Comparison

There is a variety of program analysis techniques which can help to achieve the same goal. The two main choices other than the dynamic taint analysis are symbolic execution and static taint analysis. Symbolic execution, for example, iterates all possible paths in theory and therefore is the most precise solution. Whenever an if statement is encountered, a symbolic execution engine will explore both paths. In theory, the search space increases exponentially for only a few if statements. In the same time, a typical Android application fits the event-driven model which means the order of event handler is uncertain until runtime. This nature further increases the possible search space. When the search space reaches the limit of memory and time that a system can handle, the result is called path explosion. Experiments show that for a larger application like facebook app, the soot [6] at the time writing the thesis cannot handle the large Android application. The next technique is static taint analysis. On contrary with dynamic taint analysis, the static one does not run the application. A static analysis system conducts analysis for each method first and then use graph theory to connect the dots. FlowDroid [7] for example uses such approach. The problem is that the precision is traded off for the benefit of reduced usage of resources. This increases the false positive rate which is undesirable from the perspective of finding evidential data. For the dynamic taint analysis, the benefit is low false positive rate as it

runs the program. It costs fewer resources compared to the two other approaches. There are some problems with the technique which is discussed in the last chapter.

3.2 Android Background

Understanding the Android source code is crucial for the implementation. The section documents findings about the source code during the development.

Android System just like others is very complicated. With 20GB source code, it surely takes a lot of time to understand let alone modification. Over the years, it has evolved over several generations. The latest version at the time of writing sits at version 7 with version 8 coming online in near future. The current version uses a runtime environment called Android Runtime since version 5, which is a combination of ahead-of-time compilation and interpreter. By contrast, the older version uses Dalvik VM which is an interpreter based runtime environment coupled with just-in-time compilation. A system like TaintDroid modifies the interpreter on Dalvik and achieves the taint tacking ability. However, with the system update, newer applications might not run on the older version of OS. Therefore, this gives the need for a new system.

The ART uses a compiler called Dex2oat which converts an Android application package into an oat file. Such oat file contains both dex code and binary code so that the environment can choose to run the application either in binary mode or interpreter mode according to current system status. One thing noticed during development is that, if the system notices any modification or fails to verify system integrity, then the safe mode will be used. Under such mode, the code from both app developers and Android framework is executed by interpreter while some basic classes like object class under “libcore” folder are executed in binary mode.

The interpreter does not execute the binary code for java method but operates the dex code. By default, it is used for debug mode or safe mode. There are two kinds of interpreter implemented under hood. One is switch-based implementation written in C++ code while the other involves assembly language. The switch-based implementation iterates over the instructions with a while loop and each instruction is handled by a giant switch statement where under each case some appropriate code is executed. The benefit of this switch-based interpreter is that the structure is clear and the modification can be easily done. A simple experiment that executes Facebook and Instagram applications under the mentioned interpreter was carried out and showed no obvious lag. The other interpreter which involves assembly language is called "mterp". This interpreter instead of using a loop with a switch statement inside employs assemble code to calculate the address of a chunk of code to be executed. This is done presumably for the purpose of better performance. The drawback is that the code becomes less clear and harder to modify. Also, the mterp interpreter does not work for every case. When debug mode or safe mode is enabled, the interpreter will fall back to the previously mentioned one.

When launching a program, a series of operation will happen. Figure 4 shows the simplified procedure. Such procedure is modified and the modification is described in next section. When an application gets launched, the system forks the root process zygote and invokes the main method of the launched activity. The method is first checked if it is native. If not, the system checks if debug or safe mode is enabled. If so the system executes the method with the interpreter. If not, the system checks if the method has the corresponding native code. If not, the system tries to compile the application. In the case when the compilation failure, the system falls back to the usage of the interpreter. On the other hand, if the binary code is available or

compilation success, the system executes the method in native mode. Another interesting fact is that even under interpreter mode, some core methods and classes like Object class is executed in native mode. The core classes including Object, String, Array and so on. There some corresponding mirror implementation in C++. The procedure can be summarized in the following figure.

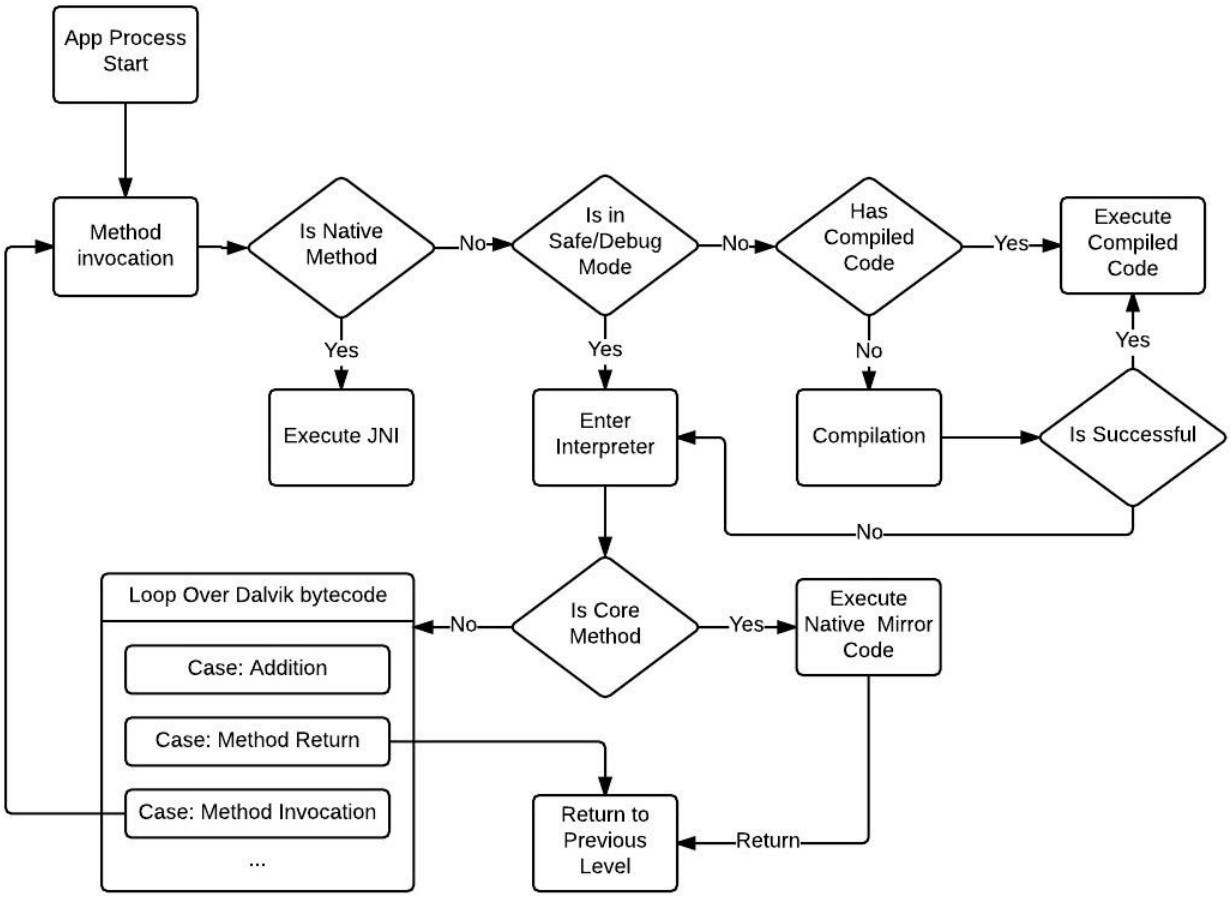

Figure 4 Simplified Execution Procedure

There are three folders that require attention: art, framework and libcore folder.

The “art” folder stores source code files for the Android runtime system which includes interpreter, compiler, class linker, memory allocator, garbage collector, JVM implementation, mirror classes and other utilities. The mirror classes refer to C++ version of some Java classes like object, string, primitive array, object array and etc. Those classes are used by other

components like interpreter and class linker. The class linker component handles the class loading during runtime. It is also used for un-started runtime which occurs during OS compilation. The class linker also attempts to check the basic loading procedure so that class size, pointer size and etc are properly initiated. For example, the class linker compares if the size of a Java object is the same as the size of an actual instance. "CHECK_EQ(java_lang_Object->GetObjectSize(), mirror::Object:: InstanceSize());" The size of an object is fixed and the value is stored in a field of the class while the instance in some case is predefined. This code statement demonstrates the internal checking procedure which occurs all over the system. Therefore, any modification must be done in a careful manner, otherwise, the basic checking won't be passed. Another two important files for checking lie in the "generated" folder. The asm_support.h and asm_support_gen.h contains purely system MACRO for values and checking.

```
#define SHADOWFRAME_LINK_OFFSET 0
ADD_TEST_EQ(SHADOWFRAME_LINK_OFFSET,
            static_cast<int32_t>(art::ShadowFrame::LinkOffset()))
#define SHADOWFRAME_METHOD_OFFSET (SHADOWFRAME_LINK_OFFSET + 1 * __SIZEOF_POINTER__)
ADD_TEST_EQ(SHADOWFRAME_METHOD_OFFSET,
            static_cast<int32_t>(art::ShadowFrame::MethodOffset()))
#define SHADOWFRAME_RESULT_REGISTER_OFFSET (SHADOWFRAME_LINK_OFFSET + 2 * __SIZEOF_POINTER__)
ADD_TEST_EQ(SHADOWFRAME_RESULT_REGISTER_OFFSET,
            static_cast<int32_t>(art::ShadowFrame::ResultRegisterOffset()))
#define SHADOWFRAME_DEX_PC_PTR_OFFSET (SHADOWFRAME_LINK_OFFSET + 3 * __SIZEOF_POINTER__)
ADD_TEST_EQ(SHADOWFRAME_DEX_PC_PTR_OFFSET,
            static_cast<int32_t>(art::ShadowFrame::DexPCPtrOffset()))
#define SHADOWFRAME_CODE_ITEM_OFFSET (SHADOWFRAME_LINK_OFFSET + 4 * __SIZEOF_POINTER__)
ADD_TEST_EQ(SHADOWFRAME_CODE_ITEM_OFFSET,
            static_cast<int32_t>(art::ShadowFrame::CodeItemOffset()))
#define SHADOWFRAME_LOCK_COUNT_DATA_OFFSET (SHADOWFRAME_LINK_OFFSET + 5 * __SIZEOF_POINTER__)
ADD_TEST_EQ(SHADOWFRAME_LOCK_COUNT_DATA_OFFSET,
            static_cast<int32_t>(art::ShadowFrame::LockCountDataOffset()))
```

Figure 5 Sample Integrity Checking

Figure 5 is an iceberg from those files. It shows the expected value for the shadow frame class where each field has expected offset from the start of the class. This is needed due to the assembly file which uses those offsets in assembly code.

The framework folder is for Android framework. The code includes input method service, hardware driver, database, animation and so on. There could be many API or static field to be considered as source. For example, the system information stored in the fields from android.os.build can be treated as sources.

The libcore folder includes core Java library, for example, Object.java, FileInputstream.java and FileOutputstream.java, and along with the native implementation for those native Java methods.

3.3 Theory

Taint analysis is a program analysis approach which adds label or tag to the variable within the application under test. During the execution of the application, variables store information and the information passes around via the instructions like assignment operations. The goal of the taint analysis is to track where the information goes to so that the flow of which can be revealed.

3.3.1 Important Concepts

a. Taint tag: Taint tag or tag refers to the type of information which travels with a variable. If a variable is tainted or has certain taint tag, then it means the variable carries a certain type of information like GPS longitude.

b. Source: Source refers to the starting point where a variable gets tainted. Typically, some system method can be treated as a source so that whenever the return value is assigned to a variable, the variable become tainted.

c. Sink: Sink refers to the checkpoint in the system. Methods like file writing can be treated as a sink so that whenever the data being sent to this method, the system will check the input and report if it is tainted.

d. Sensitive information: Not all information deserves attention because many of them are just temporary or simply of no use. Information like phone device ID and user text input may require some attention while constant in random class can usually be ignored.

e. Taint propagation: Variable is constantly under computation and reassignment. In the process, information flows from on to another and so should the flow taint tag occur. Whenever a variable is assigned to another, the same should happen to the taint tag. Take addition varA = varB + varC for example, the varA should get the taint from both varB and varC.

f. Taint tag storage: Taint tag as a kind of information requires some memory space to be stored. The size of taint tag for each variable varies based on the implementation.

g. Propagation rules: Propagation rules define how the information shall flow when an instruction gets executed. For example, the binary operation like addition shall give a result of the union set of both operands' tag. For other operation like array element access, the implementation can simply supply an individual tag associated with the element or the tag of the array object.

3.3.2 Example

```
  public class MainActivity extends FragmentActivity {
1   String local_field;

    @Override
    protected void onCreate(Bundle saveInstanceState){
2       super.onCreate(saveInstanceState);
3       setContentView(R.layout.sample_main);
    }

    public void Button_click_1(View v){
4       local_field = GetPhoneString();
    }

    public void Button_click_2(View v){
5       Send_to_Internet(local_field)
    }
  }
```

Figure 6 Example Main Activity

Figure 6 above shows a simple example of a main activity with two buttons and a onCreate method. In the first button click handler, the program accesses the phone string from the system which can be considered as a source and assigns the data to a field. The second button click handler sends the data to Internet which can be considered as a sink. In theory, the system should attach a taint tag of the phone number to return value of "GetPhoneString" method. The assignment operation carried out immediately after the method return should give the "local_field" both the phone number string and taint tag which is defined by the propagation rule. In the second button click handler, the internet sending action serves as a sink method which checks the taint tag of all input arguments. If the first button click handler was invoked before, then the "local_field" should carry the mark and the checking mechanism in the internet sending method should report such flow.

3.4 Implementation

Figure 4 shows the simplified procedure of Android runtime when an application gets launched. Upon launching an application, a new process is born, dedicated to the application. The system checks if it is necessary to execute native code either because of JNI or compiled code available under normal condition. If not, it will enter interpreter which loops over Dalvik bytecode and executes instructions accordingly. A special situation is that if the method belongs to core library like object class, the system will execute native code compiled from C++ code during OS Compilation. These code mirrors the behavior of their Java counterpart and is intended for better performance. Based on the understanding, the modification forces the system always enter interpreter and bypass the checking of trusted mirror class so that all the Java-based code executes under interpreter. Furthermore, the modification alters the interpreter instruction case handling procedure so that the taint propagation can be implemented.

Two modes are implemented for the system. Namely, they are bit-wise mode and tag-id mode. The system under bit-wise mode is similar to what was done in TaintDroid with the difference of supported version of Android OS(7.0+). This mode focuses on efficiency and serves as a milestone for the further modification. A tag is a 4-byte memory and each bit in it represents if a certain type of information is carried or not. On the other hand, the tag-id mode aims to have the capability of distinguishing a large amount of information which primarily focuses on file. The system maintains a global mapping between id and tag set. The tag in tag set ranges from 1 to 232. In comparison with Artist which faces the problem of unique id for variable and object, the system under tag-id mode avoids such problem by assigning each field a unique id during runtime and can support up to 232 tags. Another difference between two modes is the way they store the tag and how the tag union is carried. Under bit-wise mode, OR operation is used for both tags are of 4-byte long. Under tag-id mode, the system first finds the tag sets and then merge set data.

3.4.1 Taint tag storage on stack

Whenever a method is invoked, a chunk of memory is allocated on the stack. In the source code, the method frame is represented by the class ShadowFrame. The ShadowFrame mimics a conventional method frame used in other architecture like ARM. It has a 4-byte integer array, return value and reference to art method. The array acts as virtual registers and holds the values used by instructions. For example, the instruction like v1 = v2 + v3 specify the addition of virtual register 2 and 3 and the result is put into virtual register 1. The implementation of bit-wise mode introduces an array of 4-byte integers so that each virtual register has its own correspondent 4-byte tag. Meanwhile, the implementation of tag-id mode adds a vector of integer set whose content is saved on the heap. Figure 7 and Figure 8 illustrate the tag storage on stack under the

bit-wise mode and tag-id mode. A result tag or tag set is added so that there is room for the taint tag of return value.

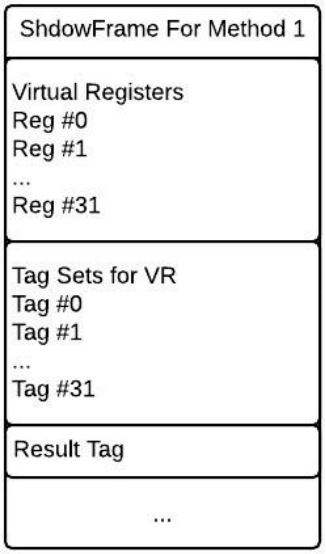

Figure 7 Tag Storage on Stack (Bit-wise mode)

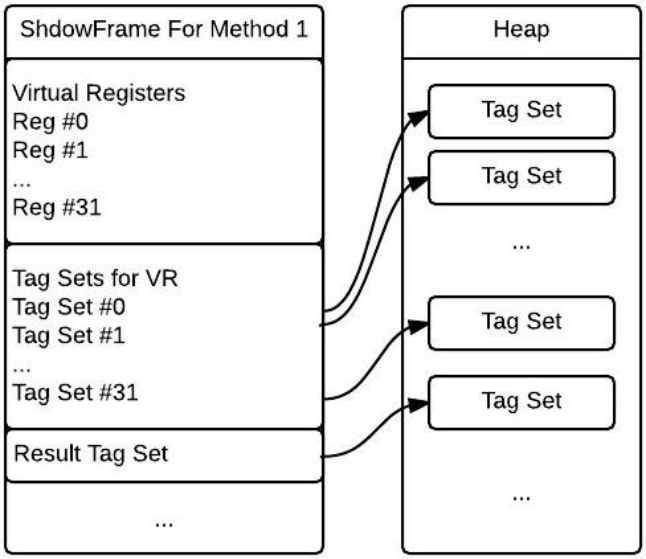

Figure 8 Tag Storage on Stack(Tag-id mode)

3.4.2 Class modification

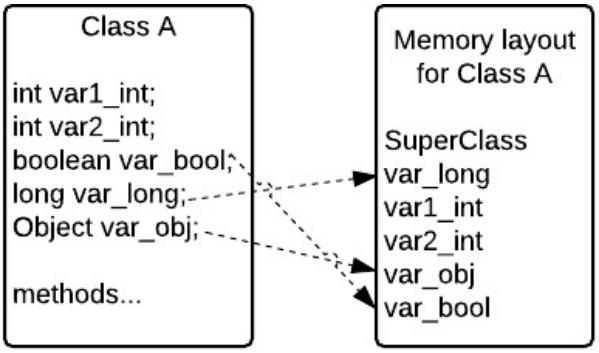

Figure 9 Memory layout for Class A

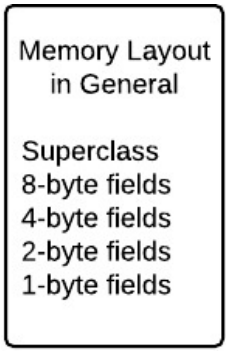

Figure 10 General Memory Layout

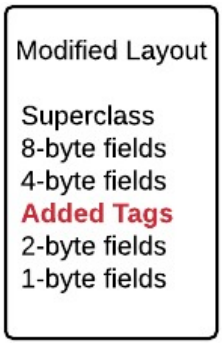

Figure 11 Modified Memory Layout

When a class is loaded into the system, a memory layout is generated. Figure 9 gives an example of how a class “class A” gets translated into a memory layout. The class A contains several variables of different sizes and types. The memory layout does not follow the same order as the one in the source code but follows the order given in Figure 10. The layout starts with the superclass and then continued with fields in descending-size order. The modification adds taint tags after the 4-byte fields. By comparison, the TaintDroid adds a tag next to every variable. The

choice is made due to the problem of memory alignment and memory gap. By default, an X-byte field should have an address which is divisible by the integer X. For example, a 4-byte field can have an address like 0b11111100 but cannot have 0b11111101. The memory fetching on some CPUs requires this rule. Therefore it is best choice to follow this setup. However, if the rule is followed, memory gaps could be created when a taint tag is added next to a field. Take a one-byte field as an example. The one-byte field could have an address of 0b0001 and its taint tag has an address of 0b0100. This setup follows the memory alignment but as a result, a gap of 3-byte is created between the one-byte field and its taint tag. This is not desirable because of the wasted memory. Therefore the modification ends up putting tags after 4-byte fields.

3.4.3 Taint tag storage on heap

Heap stores instances of each class. According to the modified memory layout from the previous section, the taint tags or tag ids are added after the 4-byte variable during the loading procedure. This is implemented via modifying files like class_linker.h and class_linker.cc. A method called ComputeClassSize calculates the total size an instance needs. Altering this method allows the memory expansion for an object. The extra memory space is treated as memory gap instead of fields from the perspective of normal execution. Another method called LinkFields enumerates fields and calculates the offset of each field related to the starting point of the object. Altering this method allows the calculation of the offset for taint tags. A new field called taint_offset is added to art_field.h. This field stored the taint tag offset which will be used during runtime in order to find the location of taint storage for a field.

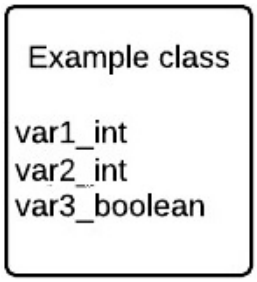

Figure 12 Example Class

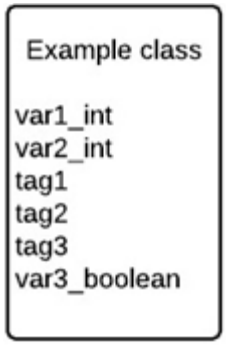

Figure 13 Example Class on heap(bit-wise mode)

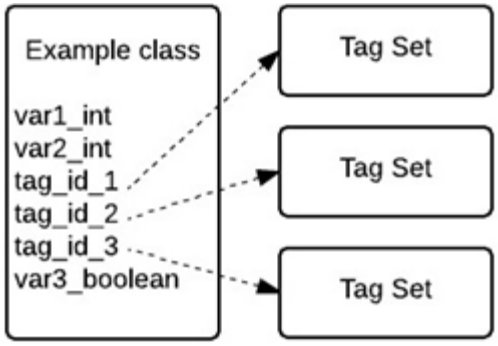

Figure 14 Example Class on heap(tag-id mode)

Figure 13 and Figure 14 shows how the example class looks like on heap under both modes. Under the tag-id mode which is on the right, the added fields are treated as id and being used to find the tag set. The mapping information is maintained in a global structure in the interpreter. The current implementation does not have explicit control over the memory management.

3.4.4 Taint propagation

There are 4 types of propagation. They are local, stack-heap, inter-method and memory-file propagation. Table 2 is the propagation rules used at the instruction level.

Table 2 Propagation Rules

Operations		**Propagation Rules** Bit-wise mode	Tag-id mode
Unary Operation: $v_1 = v_2$ op: move, move-return, …		$T(v_1) \leftarrow T(v_2)$	$T(v_1) \leftarrow T(v_2)$
Binary Operation: $v_1 = v_2$ op v_3 op: add, shift, xor, …		$T(v_1) \leftarrow T(v_2)$ or $T(v_3)$	$T(v_1) \leftarrow T(v_2)$ union $T(v_3)$
Array	aget v_1, v_2, v_3 $v_1 = v_2$ [v_3]	$T(v_1) \leftarrow T(v_2$ [v_3]) or $T(v_2)$	$T(v_1) \leftarrow T(v_2$ [v3]) union $T(v_2)$
	aput v_1, v_2, v_3 v_2 [v_3] = v_1	$T(v_2) \leftarrow T(v_2)$ or $T(v_3)$	$T(v_2) \leftarrow T(v_2)$ union $T(v_3)$
Static Field	sget v_1, f v_1 = f	$T(v_1) \leftarrow T(f)$	$T(v_1) \leftarrow T(f)$
	sput v_1, f f = v_1	$T(f) \leftarrow T(v_1)$	$T(f) \leftarrow T(v_1)$
Instance Field	iget v_1, v_2, f $v_1 = v_2$.f	$T(v_1) \leftarrow T(v_2$.f)	$T(v1) \leftarrow T(v_2$.f)
	iput v_1, v_2, f v_2.f = v_1	$T(v_2$.f) $\leftarrow T(v_1)$	$T(v_2$.f) $\leftarrow T(v_1)$

a. Local propagation happens if a virtual register is assigned a value. Take addition, for example, the result register gets the taint tag of both operands. This propagation follows the propagation rules which can found in Table 1 which is from TaintDroid. This project is inspired by TaintDroid and borrows its propagation rules from TaintDroid. The implementation modifies the instruction handling cases in interpreter_common.cc. For each instruction case, some taint propagation procedure is added.

b. The stack-heap propagation happens for field-related instructions like instance-get and instance-set. Java object and its field are all stored on the heap. And the data must be loaded onto the stack before being processed. The system first finds the ArtField instance for the target field. From the ArtField, the taint offset can be known. The system accesses the taint tag data based on the offset related to the start of the object.

c. The inter-method propagation happens for the method invocation and return. For method invocation, the implementation modifies the argument setup procedure so that the taint tags travel with the arguments. For the method return, the current method frame will be popped and destroyed. Before the final point, the taint tag of the return value is put on a data holder in caller method frame. The instruction move-result move the return value to a virtual register and implementation also move the taint tag accordingly.

d. Memory-file propagation happens for file writing and reading. The current implementation only handles file to memory propagation. When a variable gets data from a file, the variable also gets a special taint tag which represents the file.

3.4.5 Source and Sink

There are currently 11 sources which include telephone, device ID, text input, GPS data and so on. Those sources are assigned static tags which do not change over system. On the other

hand, during execution, whenever a file is opened, a tag is dynamically generated for the file. Such tag is used to distinguish files and a global mapping between tag and file path is maintained which is used for reporting at sink method. The sink includes the LoggingPrintStream.println and file writing.

3.4.6 File IO and dynamic taint tags

An application could open numerous files during its lifetime including the temporary ones. The predefined tags cannot handle the dynamic nature of file opening. The solution is to use dynamic tags which are generated during runtime. Whenever a file is opened, the file is labeled with a new integer starting from a base one. Such integer increments over time. The implementation then uses the tag throughout the propagation like file writing. A master storage of file information exists in the interpreter for each process. This storage contains tag, file descriptor, and file path. The information is accessed whenever a sink is encountered.

There are several locations where file get opened and IO could happen.

a. IOBridge. The name indicates the input and output. The class contains file-related, network-related IO method. However, only the write and read methods are used for running applications. The file-open method is invoked only during some library loading according to test. The implementation puts taint checking method at the start of file-write method.

b. Libcore.java. The file fills the role of OS API along with Java files like BlockGuardOs and Posix. The set of files handle commands like file IO and chmod. The underlying native code including io_util_md.c is used to open a file if FileOutputStream is used. The implementation inserts taint tracking code into the c file so that a taint tracking Java method can be invoked.

c. FileChannel. The FileChannel class provides another way of communicating with file system. The subroutine uses UnixFileSystemProvider to manages file opening. It is currently labeled as a source.

3.4.7 Taint APIs

The taint APIs includes source-sink setup and other methods which help taint propagation when the existing one cannot handle. Those APIs are added in Object.java so that they can be used system wide. The list can be found in Appendix B. Those methods can be broken down into following categories:

a. Assistance methods. This group of APIs helps bypass some native methods which prevent taint propagation. The native methods do not go through the interpreter and are hard to instrument so that they prevent the taint propagation. With those assistance methods, the problem can be solved but manual placement of APIs is needed. In the AbstractStringBuilder and StringFactory class, the assistance APIs are used to carry out the taint propagation when some native methods are in the way. Also, those methods can provide a shortcut for taint propagation. The first argument of APIs is most for debugging purpose, and can be supplied with null. This group has a name which starts with TAINT_ASSIST.

b. Paint methods. This group taints field, return variable and object, which turns the variable into a source. For example, if a certain object like a string field needs to be treated as a source, use the TAINT_PAINT_object to taint the object. The first argument is most for debugging purpose and can be supplied with null. The name of the group starts with TAINT_PAINT.

c. Sink methods. This group checks input argument and report if any input is tainted. The first argument can be used as a label for reporting. The name of the group starts with TAINT_CHECK.

d. File methods. The file operations involve open, read, write and close, whose subroutine requires the support from native methods. The implementation modifies the underlying native code and invokes the correspondent method from this group. The group is intended for internal use and has the name starting with TAINT_FILE.

3.5 Experiment

Three phases of experiments were conducted: testing apps, real-world app with manual inputs, and application pool with auto tests. The system was using tag-set mode and was installed to Nexus 6p device. No obvious lagging was observed.

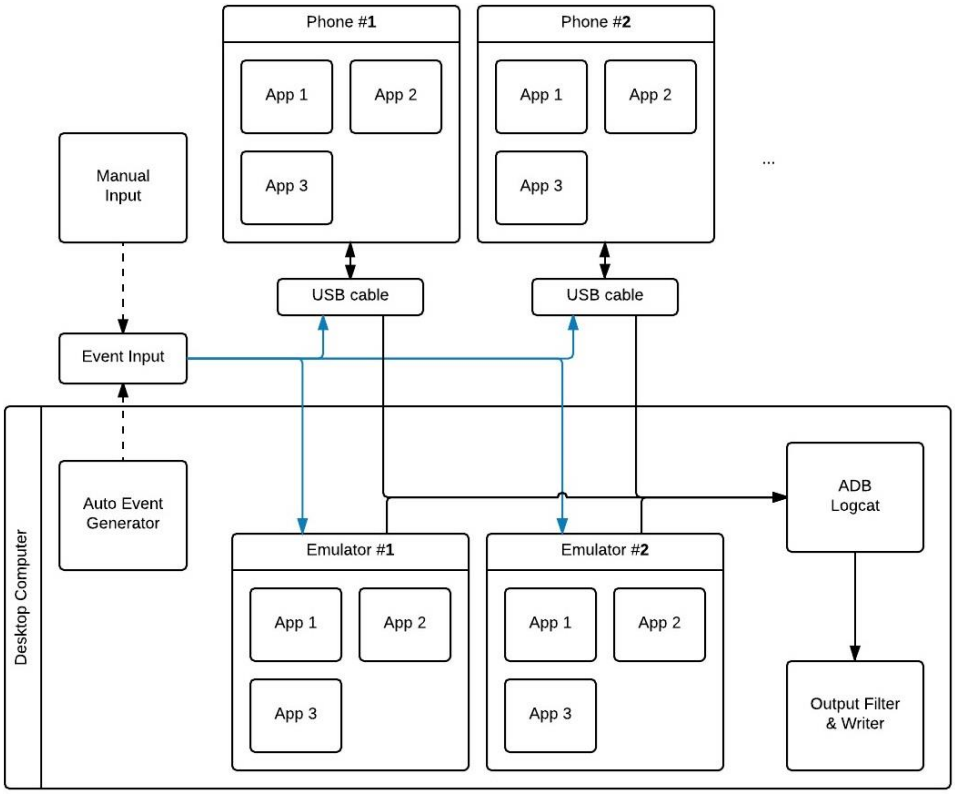

Figure 15 Experiment Setup

Figure 15 shows the setup of experiments in general. This setup allows multiple devices and emulators to be operated at the same time. The application receives events either from manual input or auto event generator. The system monitors the runtime and dumps any output to the logcat. A filter and writer was used so the output is in a neat format. In the experiment conducted, only one real nexus was used for issues which will be discussed in Chapter 5.

Testing apps: The system was tested with numerous testing apps in order to verify the basic capability. In each app, some sensitive data like contact is assigned to a variable and the variable experience simple manipulation and re-assigned to another variable. Finally, the 2nd variable is sent to a certain sink including system.out and file writing. The experiment shows that the system works as expected.

Real-world app: 25 real-world applications were tested with manual input. The group includes facebook, Line, taobao, Sideline, alipay and etc. Some applications cannot run due to the absence of Google play on customized ROM. For these applications which can run without problem, the system detected in total 10k+ information flow between files. There are duplicates in the statistic and most of them happened between temporary files. One interesting case in the app wechat shows that information from some configuration file was written into png file. Another case shows that the app sideline writes data from preference and setting files to a mp3 file. Those cases are what this system intended for. However, what exactly is written cannot be revealed at this point because reverse-engineering the applications require more time. The test also reveals that the app alipay relies on Java-based code to rename jar files.

Figure 16 Sample Output

Figure 16 shows the sample output from the system. A “cfg” file was opened and read. The data then flew to another “png” file.

Application pool with automatic test: A testing script was created for auto testing. The applications come from PlayDrone[8]. Over 400 apps were tested before the system restarted. The underlying procedure of the script takes advantage of monkey which is an auto testing tool. The monkey is capable of firing tons of events continuously to the application. For each application, a total of 5000 random events were fired with a gap of 50 milliseconds between two events. The events include click, sliding, volume up&down, screen capture and etc. The system detected 600 flows but all of them are harmless. For example, the FrameBlue app writes data from zip file to png file. In terms of amount of tracking record, the auto testing gives only 10% of what is given under manual test. The comparison is made on the same set of applications. The reason the auto test gives less result is due to following: 1) The events may not be meaningful compared to manual test. Some button or one of the execution path behind hood requires some sequence to trigger. For example, in order to log in, the username and password must not be empty. 2)Log-in based application usually requires valid credential in order to proceed. During manual testing, the tester registered account so that the testing can proceed. Some action is not supported by system right now. 3)Around 20 PlayDrone [8] apps were selected at random and tested manually but only 8 apps could run on Android 7.0 which the current system is built upon. Some partial result can be found in Appendix C.

CHAPTER 4 EVENT SEQUENCE GENERATION

One problem with the dynamic taint tracking is that it requires event inputs during runtime otherwise the application will do nothing after launch. Therefore, some inputs like button clicking and text entering are needed. The simple solution, of course, is to have human operator. With manual input, the application can be exercised with reasonably good event sequence and a result can be obtained as shown in the previous experiment. The drawback is the additional human resources. The other approach is to automatize the process by having an event sequence generator. This chapter discusses in detail about how this can be achieved. The first solution of automatic process is to use random testing which is shown in the previous experiment. It uses Monkey which is on-site component to generate random events. However, the result is less ideal. Other than the log-in mechanism, the major hindrance is how to generate meaningful events. The Monkey-based approach is fast but very limited. Observation shows that the tests often fall into loop and do not exercise the application thoroughly enough. This gives the need for a second approach of smarter guidance. In such approach, the application under test is being constantly monitored and analyzed. New events and event sequences are generated based on current status of the application. This chapter introduces the algorithm and a prototype program called APEX or Android path explorer.

The APEX explores the application under test first with random event input and then with generated event sequences. Such procedure loops overtime until no new events or sequences can be generated. The program employs the idea of the concolic execution which stands for concrete and symbolic execution. The concrete execution is just the normal running of the application

while the symbolic execution is a static analysis approach which treats very variable as a symbol rather than a concrete value. As a result, each variable has a range of values represented by some mathematic inequations.

Symbolic execution is a program analysis method which tries to determine what input triggers which part of the given program. The symbolic execution starts from the entry point of the given program and traces all encountered variables. Instead of assuming a concrete value for each variable, the program treats a variable as a symbol and typically represents it as a mathematical expression. When execution encounters an if statement, constraint(s) are added to related variables. The constraint means that if and only if the variables satisfy some mathematic condition could the branch takes place. For each of the following branches, the program executes separately and the process keeps repeating until the end or some limitation is reached. Symbolic execution faces some limitations primarily the path explosion. Each condition creates two branches so that the search space would go exponentially. In the implementation, some APIs might be unresolvable so that the symbolic execution could be incorrect due to the black box.

An effective symbolic execution program will try to eliminate as many paths as possible. However, a more headache problem occurs for event-driven platform like Android OS. The next path is uncertain. A typical GUI application like many in the Android platform requires user inputs so that the next action is determined. Such simple action is seen as an obstacle to symbolic execution because the order of methods is not certain. The naive approach is to iterate through all possible events on the entire screen but it might lead to the path explosion.

Concolic execution is abbreviated for concrete and symbolic execution. It is a hybrid testing methodology for determining what input triggers what part of a program. It is similar to symbolic execution in concept but focuses more on execution of the program with real input. In our

context, the dynamic taint analysis system is a dynamic approach and therefore the concolic execution is chosen. In each round, the program tries to execute the given program with some input and monitor the execution path. When encountering a branch, the runtime will only execute one branch while the other branch and its constraint are recorded by the concolic system. After the execution is finished, the concolic system tries to invoke the other branch by feeding the program with different input. The input is generated based on the reversed constraint of the previous execution path. For example, if the current branch has a constraint x > 10, the reversed constraint is x <= 10 and one possible new input is x = 1. The system keeps repeating the process until all discovered branched are tried. However, this approach faces a problem similar what symbolic execution faces, which is the event handler. The event handler gives some challenges: 1) In order to invoke an event handler, one or more events need to be feed to the program because between the starting UI of the application and the UI where a specific event handler could be executed, there might be multiple intermediate UI. By comparison, non-interactive program only accepts input at the start of the execution. 2) At each UI screen, a set of events exist but need to be discovered first. 3) The order of a sequence of events decides the order of methods executed. For a small non-interactive program, the change of input could directly affect the variable in a branch statement. On the other hand, change the event like button clicking might not change the variable at all. This adds some difficulty to the problem.

4.1 APEX approach

To perform Concolic execution properly on an event-driven application, it is necessary to analyze the application and determine what events are possible at each application state. Events are the driving source for the application to move from one state to another. Taking Android application as an example, an Android activity responses to events by calling event handler. In

such process, the state of the application could change. The real goal here is to discover the event handler by exercising some events so that new branch can be discovered. Therefore, the events should be the input from the Concolic execution program. However, this along is not good enough. One sole event can help discover different branches but does not help execute them. The Concolic execution needs a way to reach a specific branch otherwise it is meaningless so that the issue of input generation for target path solving raises. For example, a branch is guarded by x>0 where x is an integer variable as shown in Figure 17. What needs to be done is clear to human: find a sequence of events so leads to a concrete value like 1 or 2 for variable x before triggering the event so that the desired branch can be executed. One of the following sections will discuss this in detail.

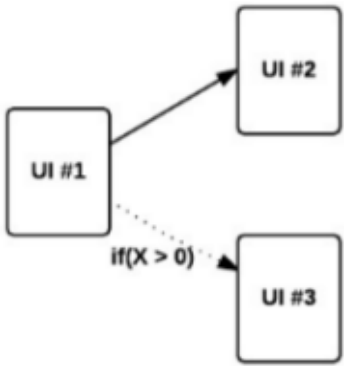

Figure 17 UI Branch

The APEX repetitively performs the concolic execution in the hope that some update could be made to the application model for each round. One of feature compared to others is UI discovery. Initially, the APEX knows nothing about the given program. With each round update, it slowly discovers new UI along with new potential events. The branch which leads to a certain UI could be guarded by an if statement so that only if a proper event sequence could go through.

Some static analysis is performed prior to the concolic execution for purpose of preparation. The APEX use APKTool to reverse engineer the APK and yield the Smali code version of the

application along with other resource files. Static analysis is performed immediately afterward. A typical Android application includes Davilk byte code, XML resource file and various image and text files. The XML files contain the layout defined by the developer. These layouts will be dynamically loaded for the activity with only an id appearing in the code. It is possible to identify the resource file which is called R.java in source code by default. If the resource file is located, the layout for each activity can also be identified. However, with code obfuscation, it becomes harder to identify the resource file statically. Instrumentation usually means the insertion of code into the application so that additional feature can be added. The APEX instruments the application for purpose of monitoring and tracing. There are lots of ways to keep track of the application, which includes instrumentation on Android OS, Java debugging bridge, and instrumentation. The APEX chooses to instrument on the application so the amount of information is limited (compared to instrumentation on OS level) and the lower IO (compared to JDB type tracing). For each branch (if statement), a trace label is inserted so that when a branch gets executed, the Logcat will print out some information and the APEX can detect the execution in such manner.

4.2 Overall procedure

Figure 18 illustrates the overall procedure employed by APEX. The procedure is repetitively executed until some limitation is reached. The image serves as a concise demonstration for one round of the concolic execution.

At the starting of each round, the program checks the current states and determines what step to take. The first choice is to execute a new event. The new event is defined as an event which is never executed before and the consequence is unknown. A new event is generated when a new UI is encountered. Exercising the new event allows the program to collect more

information out of the application which is discussed later. When the APEX starts to test an application, a launch activity event is created, which leads the program to the main Activity. The second choice is to validate an event summary pair which is the combination an event and summary for the underlying event- handler.

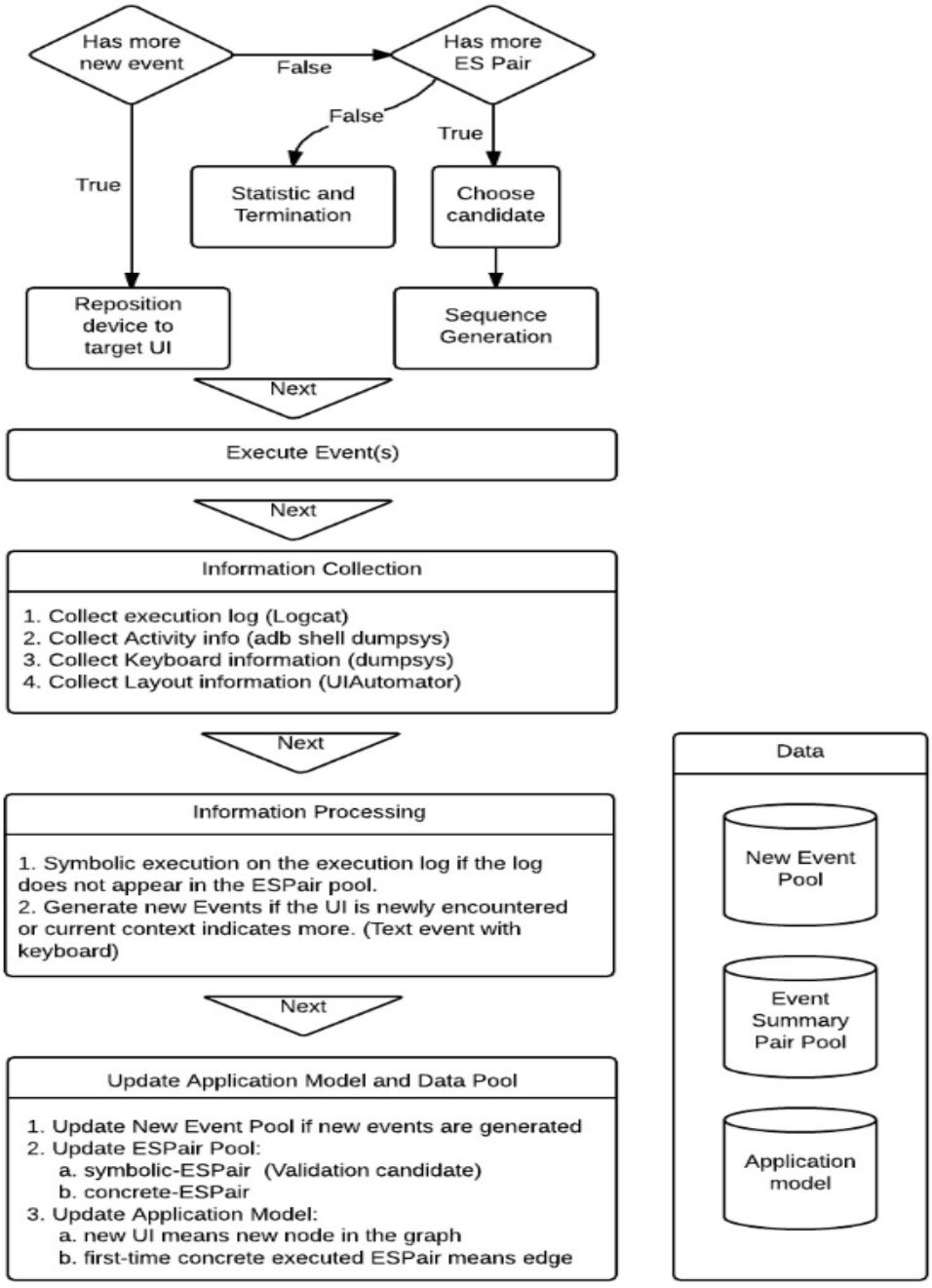

Figure 18 Solution Overview

To invoke a certain event handler, the corresponding event needs to be executed on the right UI layout. However, the application under test might not sit on the right UI. A sequence of events shall be generated for purpose of reaching a target UI. At the same time, the final event sequence should prepare the program so that the variable value can satisfy the constraints of the target path. The two requirements which are UI reaching and constraints satisfaction should be met at the same time.

The sequence generation involves the core idea of APEX. Essentially, it tries to play as a detective and find crucial leads among a crime scene. And this detective tries to connect those leads and completes the puzzle. Just before the execution of the selected event, some preparation is made. This includes buffer clearing, logcat setup, and JDB setup. The event execution is through ADB input which allows events of all kinds to be applied to the device. APEX uses a real device so that the result stands more confident.

4.3 Information collection

After the event execution, information is collected. The goal is to answer what is the consequence of the executed event by acquiring the execution trace and current application state. The trace is the execution path which consists of a list of statements at the code level. The application state includes the UI layout, keyboard, and activity information. These information helps determine if the current state was encountered and helps build the application model. The execution of an event is reflected primarily via the executed statements. APEX retrieves such information via the logcat. As mentioned before, the application is instrumented and the execution of each branch will print out some traces though logcat. In such manner, the transaction at code level could be known to APEX.

The second important piece of information is UI layout. This is done through UIAutomator which is available on the Android device by default. The UIAutomator provides UI information as a layout tree where each node is a view object which is either a view group like LinearLayout or a widget like TextView. The UIAutomator also tells if the view object can receive events like click (though it is not accurate all the time) along with some basic attributes from class type, view id to view pixel location and size. In practice, the structure of a layout tree, however, remains the primary factor in the comparison of two layouts. The text input to an EditText view object could lead to the change of size of both the EditText view and nearby objects.

Keyboard Information can be retrieved from ADB dumpsys. The presence of virtual keyboard indicates the previous event invoke a text input request. In such way, it can be known when a text event is available. Moreover, the dumpsys on keyboard provides the type of keyboard whether if the application is accepting numeric value only or an email string. It is not 100% reliable as such information is determined by the author of the application but it is sufficient in many cases. Activity information includes like class name, package, window size to activity stack can be retrieved by the ADB dumpsys. Class name and package name is the major factor.

4.4 Information Process

From the previous phase, a bundle of information is ready to use. Now it is time to extract useful information out of them. The most important analysis is the symbolic execution on the execution log. However, the execution log contains only the branch label information. Before it can be further used, it must be inflated into full path log which contains all statements from the start of the event handler to the end. After such preprocess, it is then passed to the symbolic execution program. The program performs the symbolic execution and forms a list of path

summary. Path summary serves as a conclusion of what has happened. It consists of the execution log, the constraints, and the resulted symbolic states. The constraints on each variable are the conditions which need to be satisfied so that such path can be taken. The symbolic states are seen as an update for the current application state. The information will be used in later sequence solving procedure.

In practice, the symbolic execution often leads to path explosion. APEX also faces a similar problem. The large quantity of code base in the Android framework represents a clear signal of path explosion. Also, the object-oriented aspect of Java along with huge library integrated into Android Framework makes the work even harder. The vast reuse of API, the countless use of callback methods stand strong against any symbolic execution program. Therefore, some compromise must be made so that a workaround is possible. The solution is to trade precision and accuracy for lower complexity. A large chunk of processing power is used to update the GUI in Android application and many of them only get invoked by a few statements. Those GUI APIs are written by Google and can be considered as harmless. Thereby in the symbolic execution program of APEX, the GUI API is ignored, saving a large amount of time. On the other hand, some API model is used in order to simplify the process of symbolic execution. Taking List class as an example, it is an interface of Java library with no associated code directly. However, the underlying functionality for all implementation remains largely the same. ArrayList, LinkedList and others all implements List and can be viewed as the same from the top level. In short, instead of creating a model for an individual class, one can create a model for an interface, which can be reused for many occasions. Also, the model primarily serves as a simplification for the original version of a class. The total lines in ArrayList for openJDK1.8 reach 1500 which is unfeasible for symbolic execution if the usage is high and the environment is complex. By the contrast, the

model used by APEX only contains a few lines, ignoring many security checks and redundant procedures.

Some statistic is performed so that the most popular API can be known for both the Android and Java. This can serve as a guideline for creating a model. In such way, the effort spent on building model can be reduced.

The second information gets processed is the GUI layout. As mentioned before, the layout is represented by a layout tree along with some other information like activity name and keyboard visibility. The comparison is performed in terms of tree structure against the previously encountered layout tree so that conclusion can be made if the current is new. A new layout indicates new possible events and new possible paths. The implementation tries to recognize the click event on each leaf view. Typically, events like clicking happen only on buttons.

Currently, only the click event is taken into full consideration due to its existence nearly in all application. The button push event like the power and sound button can be simulated but is usually ignored due to a low occurrence. The other events like swipe and drag are not taken into consideration. If this is to be done, some analysis needs to be performed on the code level. In order to receive events, the code needs to register an event handler in either runtime programmatically or statically in the XML. Under a certain class, as long as there exists some indication of event registration could append such event to the waiting new event list.

The APEX intends to keep the amount of work at a minimum. New events are only generated for the Layout which belongs to the application. Sometimes the application can start a new application like the email application. If this happens, the implementation will simply try to bring the execution back to application under test.

It is possible that events generated are meaningless (triggering nothing) but this won't affect the result. What is more troubling is the customized widget which could occupy a large area on screen but still appear to be one widget. The Sudoku app is the very example.

Figure 19 shows the Sudoku application on the Android device. The upper side hosts a large customized view which labeled by a red rectangle. This area consists of a gird of small rectangles and clicking each of them resulted into different consequence. Given the limitation of UIAutomator, it can only be seen as a large widget.

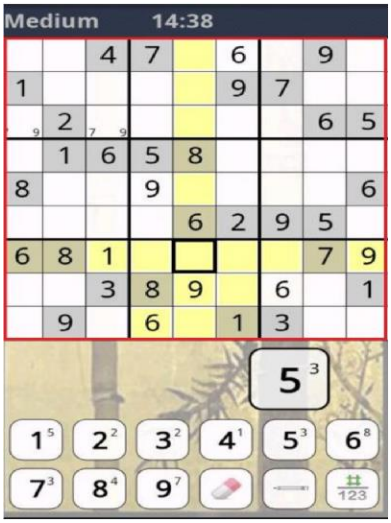

Figure 19 Sudoku Screenshot

The keyboard visibility is the indicator of whether the current event triggers a text input request. The APEX uses this information to determine if a text event is possible.

The application model is the core date for the APEX. The entire concolic execution program relies on it. The application model on the surface is the GUI model where the node is each unique UI which consists of a layout tree, activity related information and other attributes and the directed edges are the event path summary pair. The event path summary is a unique concept and it tries to answer what could happen if an event is executed. The edge starts on one of the UI and ends with another. The transition under the hood is caused by the code executed which is the

reason for the presence of the path summary. The application model constantly gets updated with new UI and new event path summary pair. The model only takes in the concrete event path summary pair while the symbolic one is stored as a candidate.

4.5 Sequence Solving

The core idea is to find leads and connect them as the final event sequence. The sequence solving dives into the mathematical world where the solver is used. The information is represented in path summary which includes constraints and symbolic states. The leads are called Anchor in the following section. An anchor event is said has major influence along the sequence which leads to the satisfaction of constraints for the target event path summary. The program finds a list of anchor sequence and then finds the connector between each anchor so that the sequence can start with an entry and ends as desired.

4.6 Anchor Solving

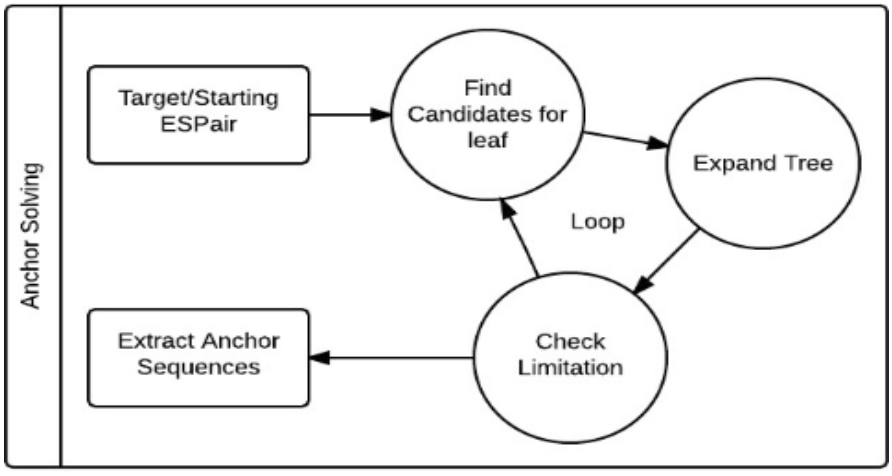

Figure 20 Anchor Solving Overview

Figure 20 serves as the framework for anchor solving. serves as the framework for anchor solving. The solving procedure is basically to build a tree structure where each branch represents

a potential anchor sequence. The root of the tree is the target event path summary pair. At each round, the program search for potential Anchor candidate in the application model.

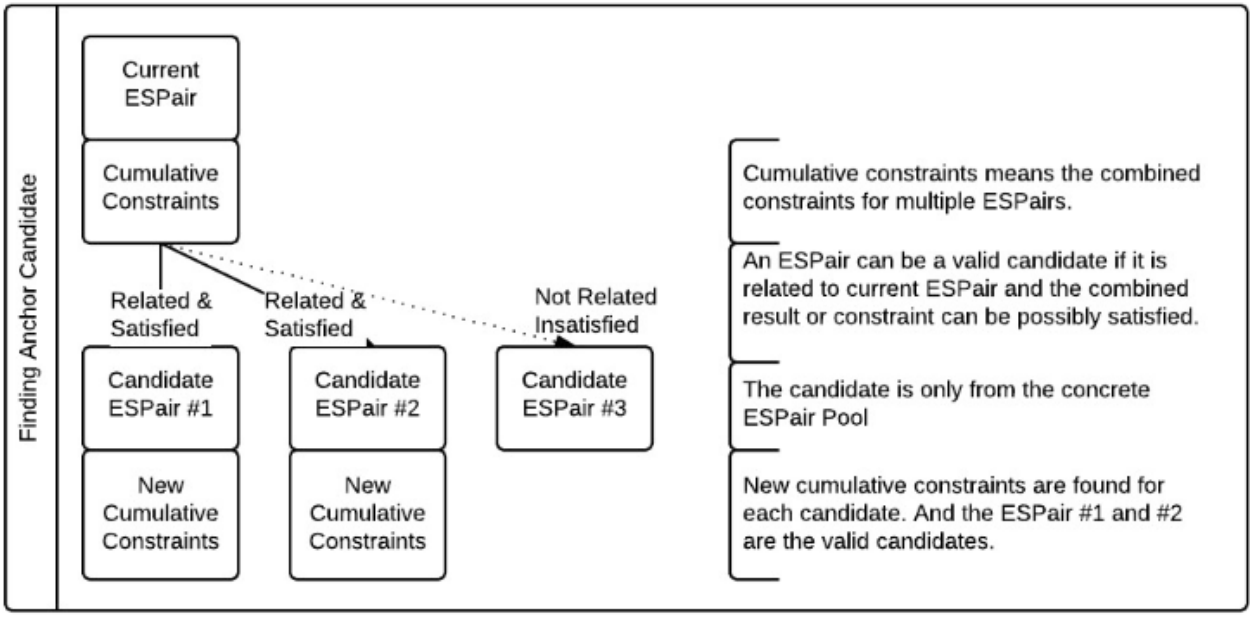

Figure 21 Anchor Candidates Finding

Figure 21 illustrates how candidates are selected. The ESPair stands for event-symbolic states pair which consists an event like button click and resulted symbolic states if the event is executed. The current ESPair show in the figure is the program working on. Among all possible candidates in the application model, the program tries to select some based on two criteria.

1. Related: An ESPair 1 is related to ESPair 2 if the set of variables in the symbolic states of ESPair #1 intersects with the variable set of the constraints from the ESPair #2. For example, if $x > 0$ is the constraints for current ESPair, an ESPair whose symbolic states should at least attempt to change the value of x. Otherwise, it is meaningless to assume such candidate could happen before the current one.

2. Satisfaction. The symbolic states of candidate should satisfy the cumulative constraints of current ESPair. For example, if $x > 0$ is the constraint then a symbolic state like "$x = 1$" is

considered satisfied. The satisfaction checking is done by the solver. The current solver used is CVC4 [9] which supports from numeric type to String. In figure 12, ESPair #1 and #2 is related and satisfy the cumulative constraints while ESPair #3 does not. Therefore ESPair #1 and #2 is added to the candidate list.

As illustrated in Figure 20, the loop continues until some limitation like max search space reaches. In the end, a tree structure is formed. From the root to each leaf lies a potential valid anchor sequence. An anchor sequence is considered valid if 1) it starts from the target ESPair and ends up with an ESPair where all the constraints are satisfied, or 2) such sequence starts from the target ESPair and ends up with an entry point like the launch application event with only the variables in the constraints of entry point remaining unsolved.

4.7 Using A Solver

As mentioned CVC4 is used to answer the mathematical problem. A class called Expression is used to hold the constraint and symbolic information. The APEX first transforms the Expressions into a script and then feed it to the CVC4 solver. Due to this nature, the ability to understand the application depends on the capability of the underlying solver. CVC4 is a powerful solver which can resolve some String problem. However, it requires additional attention due to the mixed use of String and number. Many functions like substring need additional constraints for the index value. Similar to the String, when solving the satisfaction for array related problem, the index constraint also needs to be added in order to prevent exceptions.

4.8 Finding Connectors

An anchor sequence is only a list of leads but not yet a complete event sequence. It only focuses on the code level and ignores everything happens in the GUI world. The connector, on

the other hand, intends to fix this issue. In Figure 22, two anchors ESPair A and ESPair B remain to be connected. ESPair A takes the program to UI #1 while ESPair B must be invoked on UI #2. The goal of connectors is to create a bridge from UI#1 to UI#2 so that ESPair B could happen.

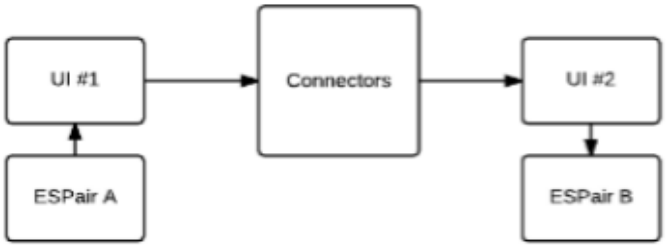

Figure 22 Finding Connectors

An event can be a connector candidate if its symbolic states do not interfere with the cumulative symbolic states of the combination from the entry point to the current ESPair. Otherwise, it should be considered as an anchor. The interference might cause the change of symbolic states and as a consequence, the next event might not trigger the desired path. The simplest way of finding the connectors is to use graph theory like the shortest path on a subgraph with a filter of some selected variables. Such approach is sufficient in many cases but may lead to the wrong solutions. The key issue is that the constraints of connector might not be satisfied.

Instead of using the graph theory directly, the connectors can be found by the same method as anchor events. To reach UI #3 like what is in Figure 23, a set of inward edges can be found. Treat each of them as a target and solve them like anchors until an outward edge is encountered of UI #2, check if the cumulative symbolic states satisfy the current constraints. If not, backward trace to the previous inward edge of UI #2. At this point, a new cumulative constraint can be formed from the target ESPair to inward edge of UI #3 with currently selected connectors to the inward edge to UI#2. The cumulative constraints can be checked and if not satisfied, the program can choose to redo the sequence solving for this part.

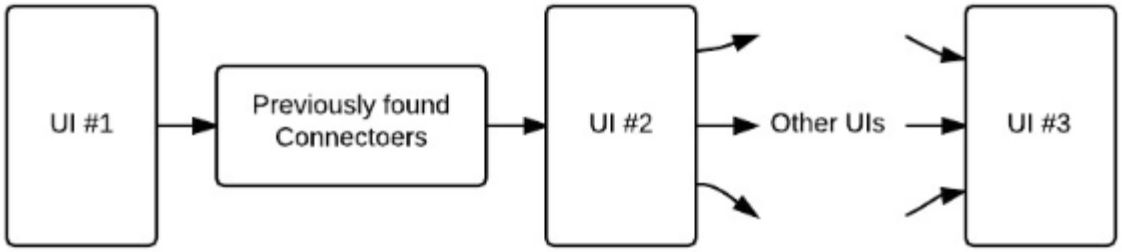

Figure 23 Connectors for list of UIs

Although this approach is more accurate the complexity and time-cost increase dramatically. In practice, if the application is simple, there is no need for this method. If the application is complex, then the time consumed goes well around exponential. Another approach is sequence merging. Due to the fact that each edge (event path summary pair) could only happen properly if the constraints are satisfied or in another word, the order of the events matter, sequence merging is very limited in this aspect.

4.9 Experiment

The effectiveness of APEX lies on the code coverage and the target reaching. The code coverage is the standard for a Concolic execution program while the target reaching ability is reflected through the code coverage.

10 applications were tested. The performance is in accordance with the capability of a symbolic execution program. The file system and various APIs are ignored by the symbolic execution program due to complexity. And therefore, applications like WordHelper only reach 2% of the total code. Secondly, some applications which use the encryption mechanism also prevent APEX from reaching a large chunk of code, like the MorseCode. The symbolic execution program is simply not powerful enough to analyze the code and produce meaningful outputs. Thirdly, some applications require an Internet connection and require input like user and password. Due to the implementation issue, those network flow is out of control and cannot be

simulated. The sequence length can reach 20. This is largely because of the new event generation and the application constantly traverse through the application with known path instead of shortest ones.

Figure 24 Application model example

Figure 24 is the application model generated by APEX. This model on the surface only shows the UI relation within the application. The model is a graph where each node is considered as a unique UI and each edge is the event with path summary information. These models can be later reused. For example, it facilitates further investigation or allows continuation of target reaching if new algorithm is equipped.

CHAPTER 5 SUMMARY

In summary, a dynamic taint analysis system tuned for forensic analysis accompanied with auto event sequence generation is implemented. The taint tracking system has two modes. One is the bit-wise mode which uses the same mechanism of TaintDroid 7.0. The other mode is the tag-id mod which aims at distinguishing as many types of information as possible. The latter one has the capability of capturing information flows between all kinds of files. The system allows either manual input, monkey-based auto testing or APEX guided exploration. The APEX system is capable of exploring the application with the assistance of symbolic execution engine which is created by my lab mate.

During the experiment of dynamic taint analysis system, numerous problems were encountered which delayed progress of testing. The first and foremost is the CPU architecture. Although the system does use a modified interpreter, there are still some applications which cannot run on a certain platform. Facebook, for example, only provides APK on ARM platform. This prevents the execution on an emulator where the host machine is of the x86 architecture. The second problem is the Google play. The google play framework does not get installed on modified Android OS so that many applications cannot simply run on a device.

5.1 Future Work and Possible Approach

a. Tag of permanent storage. TaintDroid is capable of storing tag on disk storage because the system uses extended file attribute. Current implementation does not have this capability and this feature is left for future work. A similar approach of storing tags in extended attribute can be used. For the bit-wise mode, it is straight forward as all tags are

predefined which simply means a tag number always represents the same thing. On the contrast, in the tag-id mode, due to the presence of dynamic generated tag, some additional process is needed. Instead of storing one integer under the bit-wise mode, the system under tag-id mode should store a set of tag in binary format somewhere on file system. One way of implementation is that the extended file attribute stores an id which is used to mapped to another file which store the tag set or a column in database.

b. Tag type. Currently there are only 11 static tag types. More can be added. The previous focus on implementation and verification of the basic capability.

c. Support for networking. Currently, the implementation focuses on information flow regarding file and does not put taint checking API for the network API. The simplest solution is just to put the checking API to java API of internet sending and source API for the internet reading. The tag in this case is just predefined. A more complicated solution is to adopt a similar technique used for the file IO. Whenever a socket is allocated, a dynamic taint tag is associated with it. See the section 3.4.4 for further detail.

d. Google Play installation. The google play is not by default installed on the device if the system is modified. This creates some problem for testing as many popular applications take advantage of Google play which is usually on site. This leads to poor testing result for some applications.

e. Native code. Native code has always been the obstacle for analysis on Android as java is the primary language used on this platform. The current implementation modifies the interpreter and therefore ignores all the native code. This is a common problem for the analysis tool for java-based program. For some crucial native system API, a potential workaround is to create a wrapper which encapsulates the original native API and add

assisted taint propagation in it. However, this approach requires manual modification and may not be straight forward. Other technique used in NDroid [10] may be used to further improve the analysis on JNI.

f. Further testing. Basic capability has been verified. However, in cases when large application is being tested, some expected information flow is not captured like text input. Some further investigation is needed in order to pin point to the cause. Large application is hard for reverse engineering due to large set of code and, also obfuscation.

g. Inter process communication. The implementation right now does not support inter process communication. The Android uses binder for communication and it goes through kernel level. Approach similar to TaintDroid can be considered in order to implement IPC.

h. Memory walker. When an object other than string or array goes through sink, the sub content like fields could carry taint tags. The ideal approach is to walk through the object and locate all taint tag. Because the modification is done at interpreter level, the actual class type and structure of which could be accessed whenever needed. The solution roughly requires walk through all field and each field access the taint tag by first finding the taint tag offset under art_field class and second accessing the taint tag via calculating the actual memory address of taint tag.

i. Event sequence further integration. Without some event input, applications cannot be explored properly so that no information flow can be detected simply because there is none in the first place. The manual test can generate reasonable good result but it is only feasible for a small set of applications. For large quantity, the automatic test can only give some limited results. To solve the problem, the second component, APEX, serves

the purpose of automatic exploration of the applications with guidance. It heavily relies on the underlying symbolic execution which functions as the brain of the program. The current symbolic execution program is homebrew and developed by my labmate. It is tuned for symbolic execution on Android. In the future, some more powerful symbolic execution engine could be used so that the effectiveness of the program can be improved.

REFERENCES

[1] C. Chen, W. Huang, B. Zhou, C. Liu and W. H. Mow, "PiCode: A New Picture-Embedding 2D Barcode," *IEEE TRANSACTIONS ON IMAGE PROCESSING*, vol. 25, no. 8, p. 3444, 2016.

[2] "Mobile/Tablet Operating System Market Share," NetMarketShare, [Online]. Available: https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=8&qpcustomd=1. [Accessed 5 10 2017].

[3] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel and A. . N. Sheth, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *USENIX OSDI'10,* p. 393, 2010.

[4] M. Sun, T. Wei and J. C. Lui , "TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime," *Proceeding CCS '16 Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security,* pp. 331-342 , 2016.

[5] M. Backes, S. Bugiel, O. Schranz, P. von Styp-Rekowsky and S. Weisgerber, "ARTist: The Android Runtime Instrumentation and Security Toolkit," *eprint arXiv,* 2016.

[6] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam and V. Sundaresan, "Soot - a Java bytecode optimization framework," *Proceeding CASCON '99 Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research,* p. 13, 1999.

[7] S. Arzt , S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau and P. McDaniel, "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," *PLDI*, pp. 259-269, 2014.

[8] N. Viennot, E. Garcia and J. Nieh , "A measurement study of google play," *SIGMETRICS '14*, pp. 221-233 , 2014.

[9] "CVC4," [Online]. Available: http://cvc4.cs.stanford.edu/web/. [Accessed 5 10 2017].

[10] C. Qian, X. Luo, Y. Shao and A. T. S. Chan, "On Tracking Information Flows through JNI in Android Applications," *DSN '14*, pp. 180-191, 2014.

[11] "Android version market share distribution among smartphone owners as of September 2017," statusta, [Online]. Available: www.statista.com/statistics/271774/share-of-android-platforms-on-mobile-devices-with-android-os/. [Accessed 5 10 2017].

APPENDIX A SAMPLE CODE

```
#define MERGE_XOR(first, second) first | second
    #define CLEAR_REG_TAG(shadow_frame, reg) shadow_frame.SetVReg_Taint(reg, 0);
    #define MOVE_TAG_TO_regA(shadow_frame, regA, regB) \
        {shadow_frame.SetVReg_Taint(regA, shadow_frame.GetVReg_Taint(regB));}
    #define MERGE_TAG_TO_regA_3(shadow_frame, regA, regB, regC) \
        { \
        uin32_t tagB = shadow_frame.GetVReg_Taint(regB); \
            uin32_t tagC = shadow_frame.GetVReg_Taint(regC); \
            shadow_frame.SetVReg_Taint(regA, MERGE_XOR(tagB,tagC)); \
        }
    #define MERGE_TAG_TO_regA_2(shadow_frame, regA, regB) \
        { \
        uin32_t tagA = shadow_frame.GetVReg_Taint(regA); \
        uin32_t tagB = shadow_frame.GetVReg_Taint(regB); \
        shadow_frame.SetVReg_Taint(regA, MERGE_XOR(tagA,tagB)); \
        }
    #define REPORT_TAINTED_TAG(tag) TAINT_LOG_CHANNEL << "Discover
tainted: " <<tag;
    #define UNPACK_TAG(val) (1 << val);
    #define HANDLE_TAINT_CHECK_argument(obj_vector, shadow_frame,
reg_with_primitive) \
        { \
            for(size_t i = 1; i<obj_vector.size();i++){ \
                art::ObjPtr<art::mirror::Object> p_obj = obj_vector[i]; \
                if( p_obj == nullptr) continue; \
                uint32_t tag = p_obj->getTag(); \
                if(tag != 0) REPORT_TAINTED_TAG(tag); \
            } \
```

```
                for(size_t i =0; i<reg_with_primitive.size(); i++){ \
                        uint32_t reg = reg_with_primitive[i]; \
                        uint32_t tag = shadow_frame.GetVReg_Taint(reg); \
                        if(tag != 0) REPORT_TAINTED_TAG(taag); \
                } \
        }
    #define HANDLE_TAINT_PAINT_return(shadow_frame, tag)
shadow_frame.AddExecutionTaintTag(UNPACK_TAG(tag));
    #define HANDLE_TAINT_PAINT_object(p_obj, tag) \
        { \
                tag = UNPACK_TAG(tag); \
                if(p_obj == nullptr){ \
                        TAINT_LOG_CHANNEL << "taintObject_taint: null"; \
                }else{ \
                        TAINT_LOG_CHANNEL << "taintObject_taint: " << tag; \
                        if(p_obj != nullptr) { \
                                p_obj->XOR_merge(tag); \
                        } \
                } \
        }
    #define HANDLE_TAINT_PAINT_primitive() {}
    #define HANDLE_TAINT_ASSIST_transfer(p_dst_obj, p_src_obj) \
        { \
                if(p_src_obj != nullptr && p_dst_obj != nullptr){ \
                        p_dst_obj->setTag(p_src_obj->getTag()); \
                } \
        }
    #define HANDLE_TAINT_ASSIST_merge(obj_vector, shadow_frame,
reg_with_primitive) \
        { \
```

```
                art::ObjPtr<art::mirror::Object> p_dstObj = obj_vector[1]; \
                if(p_dstObj != nullptr){ \
                        uint32_t res_tag = 0; \
                        for(size_t i = 2; i<obj_vector.size();i++){ \
                                art::ObjPtr<art::mirror::Object> p_obj = obj_vector[i]; \
                                if( p_obj == nullptr) continue; \
                                uint32_t obj_tag = p_obj->getTag(); \
                                res_tag = MERGE_XOR(res_tag, obj_tag); \
                        } \
                        for(size_t i =0; i<reg_with_primitive.size(); i++){ \
                                uint32_t reg = reg_with_primitive[i]; \
                                uint32_t reg_tag = shadow_frame.GetVReg_Taint(reg); \
                                res_tag = MERGE_XOR(res_tag, reg_tag); \
                        } \
                        p_dstObj->XOR_merge(res_tag); \
                } \
        }
    #define PROPAGATE_StackToField_FieldPut(shadow_frame, regA, p_obj, p_field,
field_type, transaction_active) \
        { \
                if(field_type != Primitive::kPrimNot){ \
                        uint32_t offset = p_field->getTaintTagOffset(); \
                        if(offset != 0){ \
                                uint32_t stack_tag =
shadow_frame.GetVReg_Taint(vregA); \
                                MemberOffset off(offset); \
                                p_obj->SetField32<transaction_active>(off, stack_tag); \
                        } \
                } \
        }
```

```
#define PROPAGATE_FieldToStack_FieldGet(p_obj, p_field, field_type, shadow_frame, regA) \
        { \
                uint32_t offset = p_field->getTaintTagOffset(); \
                if(offset != 0){ \
                        MemberOffset off(offset); \
                        uint32_t field_tag = p_obj->GetField32(off); \
                        shadow_frame.SetVReg_Taint(regA, field_tag); \
                }else shadow_frame.SetVReg_Taint(regA, 0); \
        }

#define PROPAGATE_ArrayGetPrimitive(p_arr, shadow_frame, regA) \
   { \
        uint32_t tag = p_arr->getTag(); \
        shadow_frame.SetVReg_Taint(regA, tag); \
   }

#define PROPAGATE_ArrayGetObject(p_arr, p_element, shadow_frame, regA) \
        { \
        }
#define PROPAGATE_ArrayPutPrimitive(p_arr, shadow_frame, regA) \
        { \
                uint32_t tagA = shadow_frame.GetVReg_Taint(regA); \
                p_arr->XOR_merge(tagA); \
        }
#define PROPAGATE_ArrayPutObject(p_arr, p_obj) \
        { \
      if(p_obj != nullptr){ \
        uint32_t tag = p_obj->getTag(); \
        p_arr->XOR_merge(tag); \
```

```
        } \
          }
    #define PROPAGATE_ArrayGetLength(p_arr, shadow_frame, regA) \
      shadow_frame.SetVReg_Taint(regA, p_arr->AsArray()->GetLengthTag());

    #define PROPAGATE_MovePrimitiveResult(shadow_frame, regA) \
          { \
              uint32_t tag = shadow_frame.GetResultTag(); \
              shadow_frame.SetVReg_Taint(regA, tag); \
          }
    #define PROPAGATE_MethodArgumentCopy(shadow_frame, src_reg,
p_new_shadow_frame, dst_reg) \
          { \
              uint32_t tag = shadow_frame.GetVReg_Taint(src_reg); \
              p_new_shadow_frame->SetVReg_Taint(dst_reg, tag); \
          }
    #define PROPAGATE_MethodReturnPrimitiveSetup(shadow_frame, regA) \
      shadow_frame.SetResultTag(shadow_frame.GetVReg_Taint(regA));
```

APPENDIX B JAVA TAINT API

public static void TAINT_ASSIST_transfer(String msg, Object dst, Object src)

public static void TAINT_ASSIST_merge(String msg, Object dst, Object src)

public static void TAINT_ASSIST_merge(String msg, Object dst, byte src)

public static void TAINT_ASSIST_merge(String msg, Object dst, char src)

public static void TAINT_ASSIST_merge(String msg, Object dst, short src)

public static void TAINT_ASSIST_merge(String msg, Object dst, int src)

public static void TAINT_ASSIST_merge(String msg, Object dst, long src)

public static void TAINT_ASSIST_merge(String msg, Object dst, float src)

public static void TAINT_ASSIST_merge(String msg, Object dst, double src)

public static void TAINT_ASSIST_merge_WRAPPER(String msg, Object dst, Object src)

public static void TAINT_PAINT_return(String msg, int val)

public static void TAINT_PAINT_object(String msg, Object obj, int val)

public static boolean TAINT_PAINT_primitive_assign(String msg, boolean field, int val)

public static byte TAINT_PAINT_primitive_assign(String msg, byte field, int val)

public static char TAINT_PAINT_primitive_assign(String msg, char field, int val)

public static short TAINT_PAINT_primitive_assign(String msg, short field, int val)

public static int TAINT_PAINT_primitive_assign(String msg, int field, int val)

public static float TAINT_PAINT_primitive_assign(String msg, float field, int val)

public static long TAINT_PAINT_primitive_assign(String msg, long field, int val)

public static double TAINT_PAINT_primitive_assign(String msg, double field, int val)

public static void TAINT_CHECK_argument(String msg, Object input)

public static void TAINT_CHECK_argument(String msg, boolean input)

public static void TAINT_CHECK_argument(String msg, int input)

public static void TAINT_CHECK_argument(String msg, long input)

public static void TAINT_CHECK_argument(String msg, double input)

public static void TAINT_CHECK_argument(String msg, float input)

public static void TAINT_FILE_open_WRAPPER(String path, int fd, int flags)

```
public static void TAINT_FILE_close_WRAPPER(int fd)
public static void TAINT_FILE_open(String path, int fd, int flags)
public static void TAINT_FILE_close(int fd)
public static void TAINT_FILE_read_byteArray(int fd, byte[] bytes, int ret)
public static void TAINT_FILE_write_byteArray(int fd, byte[] bytes, int ret)
public static void TAINT_FILE_read(int fd, Object obj, int ret)
public static void TAINT_FILE_write(int fd, Object obj, int ret)
public static native void reportMsg_taint(String msg);
```

APPENDIX C AUTO TESTING SAMPLE RESULT

File:/home/zhenxu/Desktop/shared_work/Output/autoTestReport/Round4/com.vng.labansecurity.flow

DST:/data/user/0/com.vng.labansecurity/files/.TwitterSdk/cl/com.crashlytics.sdk.android/59BB5E2A0391-0002-6D29-ABEDC81056A5.cls_temp

SRC:
/data/user/0/com.vng.labansecurity/files/.TwitterSdk/cl/com.crashlytics.sdk.android/59BB5E2A0391-0002-6D29-ABEDC81056A5SessionDevice.cls

/data/user/0/com.vng.labansecurity/files/.TwitterSdk/cl/com.crashlytics.sdk.android/59BB5E2A0391-0002-6D29-ABEDC81056A5SessionOS.cls

/data/user/0/com.vng.labansecurity/files/.TwitterSdk/cl/com.crashlytics.sdk.android/59BB5E2A0391-0002-6D29-ABEDC81056A5SessionApp.cls

/data/user/0/com.vng.labansecurity/files/.TwitterSdk/cl/com.crashlytics.sdk.android/59BB5E2A0391-0002-6D29-ABEDC81056A5SessionCrash.cls

/data/user/0/com.vng.labansecurity/files/.TwitterSdk/cl/com.crashlytics.sdk.android/59BB5E2A0391-0002-6D29-ABEDC81056A5BeginSession.cls

DST:/data/user/0/com.vng.labansecurity/files/.TwitterSdk/cl/com.crashlytics.sdk.android/59BB5E2A0391-0002-6D29-ABEDC81056A5.cls_temp

SRC:

/data/user/0/com.vng.labansecurity/files/.TwitterSdk/cl/com.crashlytics.sdk.android/59BB5E2A0391-0002-6D29-ABEDC81056A5SessionCrash.cls

DST:/data/user/0/com.vng.labansecurity/files/.TwitterSdk/cl/com.crashlytics.sdk.android/59BB5E2A0391-0002-6D29-ABEDC81056A5.cls_temp

SRC:

/data/user/0/com.vng.labansecurity/files/.TwitterSdk/cl/com.crashlytics.sdk.android/59BB5E2A0391-0002-6D29-ABEDC81056A5SessionDevice.cls

/data/user/0/com.vng.labansecurity/files/.TwitterSdk/cl/com.crashlytics.sdk.android/59BB5E2A0391-0002-6D29-ABEDC81056A5SessionOS.cls

/data/user/0/com.vng.labansecurity/files/.TwitterSdk/cl/com.crashlytics.sdk.android/59BB5E2A0391-0002-6D29-ABEDC81056A5SessionApp.cls

/data/user/0/com.vng.labansecurity/files/.TwitterSdk/cl/com.crashlytics.sdk.android/59BB5E2A0391-0002-6D29-ABEDC81056A5SessionCrash.cls

/data/user/0/com.vng.labansecurity/files/.TwitterSdk/cl/com.crashlytics.sdk.android/59BB5E2A0391-0002-6D29-ABEDC81056A5BeginSession.cls

File:/home/zhenxu/Desktop/shared_work/Output/autoTestReport/Round4/dangsonmai.lwp.thunderStorm.flow

DST:/data/user/0/dangsonmai.lwp.thunderStorm/files/close_button.png

SRC:

/data/user/0/dangsonmai.lwp.thunderStorm/files/drawable.zip

DST:/data/user/0/dangsonmai.lwp.thunderStorm/files/close_button.png

SRC:

/data/user/0/dangsonmai.lwp.thunderStorm/files/drawable.zip

DST:/data/user/0/dangsonmai.lwp.thunderStorm/files/info_s.png

SRC:

/data/user/0/dangsonmai.lwp.thunderStorm/files/drawable.zip

DST:/data/user/0/dangsonmai.lwp.thunderStorm/files/info_s.png

SRC:

/data/user/0/dangsonmai.lwp.thunderStorm/files/drawable.zip

DST:/data/user/0/dangsonmai.lwp.thunderStorm/files/info_s.png

SRC:

/data/user/0/dangsonmai.lwp.thunderStorm/files/drawable.zip

DST:/data/user/0/dangsonmai.lwp.thunderStorm/files/info_ex_s.png

SRC:

/data/user/0/dangsonmai.lwp.thunderStorm/files/drawable.zip

DST:/data/user/0/dangsonmai.lwp.thunderStorm/files/info_ex_s.png

SRC:

/data/user/0/dangsonmai.lwp.thunderStorm/files/drawable.zip

DST:/data/user/0/dangsonmai.lwp.thunderStorm/files/info_ex_s.png

SRC:

/data/user/0/dangsonmai.lwp.thunderStorm/files/drawable.zip

DST:/data/user/0/dangsonmai.lwp.thunderStorm/files/info_ex_s.png

SRC:

/data/user/0/dangsonmai.lwp.thunderStorm/files/drawable.zip

DST:/data/user/0/dangsonmai.lwp.thunderStorm/files/info_ex_s.png

SRC:

/data/user/0/dangsonmai.lwp.thunderStorm/files/drawable.zip

DST:/data/user/0/dangsonmai.lwp.thunderStorm/files/info_ex_s.png

SRC:

/data/user/0/dangsonmai.lwp.thunderStorm/files/drawable.zip

DST:/data/user/0/dangsonmai.lwp.thunderStorm/files/info_ex_s.png

SRC:

/data/user/0/dangsonmai.lwp.thunderStorm/files/drawable.zip

DST:/data/user/0/dangsonmai.lwp.thunderStorm/files/info_l.png

SRC:

/data/user/0/dangsonmai.lwp.thunderStorm/files/drawable.zip

DST:/data/user/0/dangsonmai.lwp.thunderStorm/files/info_l.png

SRC:

/data/user/0/dangsonmai.lwp.thunderStorm/files/drawable.zip

DST:/data/user/0/dangsonmai.lwp.thunderStorm/files/info_l.png

SRC:

/data/user/0/dangsonmai.lwp.thunderStorm/files/drawable.zip

DST:/data/user/0/dangsonmai.lwp.thunderStorm/files/info_ex_l.png

SRC:

/data/user/0/dangsonmai.lwp.thunderStorm/files/drawable.zip

DST:/data/user/0/dangsonmai.lwp.thunderStorm/files/info_ex_l.png

SRC:

/data/user/0/dangsonmai.lwp.thunderStorm/files/drawable.zip

DST:/data/user/0/dangsonmai.lwp.thunderStorm/files/info_ex_l.png

SRC:

/data/user/0/dangsonmai.lwp.thunderStorm/files/drawable.zip

DST:/data/user/0/dangsonmai.lwp.thunderStorm/files/info_ex_l.png

SRC:

/data/user/0/dangsonmai.lwp.thunderStorm/files/drawable.zip

DST:/data/user/0/dangsonmai.lwp.thunderStorm/files/info_ex_l.png

SRC:

/data/user/0/dangsonmai.lwp.thunderStorm/files/drawable.zip

DST:/data/user/0/dangsonmai.lwp.thunderStorm/files/info_ex_l.png

SRC:

/data/user/0/dangsonmai.lwp.thunderStorm/files/drawable.zip

DST:/data/user/0/dangsonmai.lwp.thunderStorm/files/info_ex_l.png

SRC:

/data/user/0/dangsonmai.lwp.thunderStorm/files/drawable.zip