Capability Leakage Detection Between Android Applications Based on Dynamic Feedback

Mingsong Zhou
School of Computer Science and Technology
University of Science and Technology of China
Hefei, Anhui, China
mingsong@mail.ustc.edu.cn

Fanping Zeng
School of Computer Science and Technology
University of Science and Technology of China
Hefei, Anhui, China
billzeng@ustc.edu.cn

Zhao Chen
School of Computer Science and Technology
University of Science and Technology of China
Hefei, Anhui, China
chen95@mail.ustc.edu.cn

***Abstract*— The capability leakage of Android applications is one kind of serious vulnerabilities. It can cause other applications to leverage its functions to achieve their illegal goals. In this paper, we propose a tool which can automatically detect and confirm capability leakages of Android applications with dynamic-feedback testing. The tool utilizes context-sensitive, flow-sensitive inter-procedural data flow analysis to find key variables and instrumentation points, then it tests the application continuously by test cases generated from test log. We have made experiments on 607 most popular applications of Wandoujia in 2017, and found a total of 6,070 in 16 kinds of capability leakages. Compared with the famous IntentFuzzer, our tool is 19.38% better on the average ability to detect permission capability leakage.**

Keywords—Android, capability leakage, inter-procedural, data flow analysis, dynamic-feedback testing

I. Introduction

Capability leakage is also known as redistribution of authority [1]. It occurs when privileged applications are exploited by non-privileged malicious applications, which enables malicious applications to perform privileged actions. Communication between Android components is widely used, and many Android application developers share the functions of their applications by exposing components (components that can be invoked by external APPs). However, many Android developers do not fully understand the rules of communication between Android components, resulting in unintentionally exposing the components that should not be exposed, or forgetting to check the permissions of calls between components [2], thus resulting in the leakage of application capabilities.

There are a lot of research work on vulnerabilities between Android components, mainly divided into static analysis and dynamic testing. The main drawbacks of static analysis work (ComDroid [3], PCLeak [4], Yi He [5], AutoPatch Droid [6], Mr-droid [7]) are that it is impossible to determine whether the vulnerabilities exist. Developers need to confirm the vulnerabilities manually, which greatly increases the development cycle of APP. The existing dynamic testing methods such as Intent Fuzzer [8] and AWiDe [9] also have some shortcomings, which lead to a high rate of missed reports. Intent Fuzzer will be described in detail later, which will be selected for comparison with our method in this paper. AWiDe works for the similar purposes as our paper, but it only considers capability leakages related to input data from external components. When constructing test cases, it only uses the intent-filter information of exposed components in Android Manifest file to construct test cases, but does not use the information in code. For example, intent extra attribute information will not appear in intent-filter, so there are shortcomings. In this paper, a test case generation method based on dynamic feedback mechanism is proposed, which combines static analysis and dynamic testing technology. Compared with the existing capability leakage dynamic testing work, it has lower false positive rate.

We define the capability leakage vulnerability between Android applications as follows.

Assuming that there is Android application A, the set of privileges it owns is set to PSet, and the set of mapping relations between privileges and the statements it protects (briefly described as tgtAPI later) is set to PUMap (permission →unitSet). The set of exposed components owned by A is ECSet, and the set of root-method owned by exposed components (the first method to be executed: root-method) is set to ECMethodMap (export-component→methodSet). The set of executable paths of the root method to the unit protected by permission is RMUPathMap (root-method, unit→pathSet).

$$if\ PUMap \neq \emptyset, ECMethodMap \neq \emptyset\ ,$$

$$\exists\ intent \neq null, \text{s.t. } RMUPathMap \neq \emptyset$$

Note: intent object is the only input for inter-component communication. It mainly contains five attributes: Component, Action, Data, Category and Extras, which represent the name of the component to be started (String), the type of operation to be executed (String), the type of data to be executed (Uri), a collection of component types that can handle this intent object (Set<String>), and additional key-value pair information set (Set < key→type value >). This paper calls intent objects from other APP components external intent.

The formula is that when the PUMap and ECMethodMap of application A are not empty, there exists an intent that is not empty, so that RMUPathMap of application A is not empty, then application A has a capability leakage vulnerability. And the capability leakage corresponds to authority of the unit in RMUPathMap.

There are many APIs without parameters in Android applications, and many APIs can cause great harm even though they can't control their data inflow. Therefore, this paper considers all TGT APIs in APP, even if they don't flow into external intent data. It should be noted that there are many normal interactions between applications that require user operation. We shouldn't think these leaking paths with UI interaction capabilities as illegal, because they are user-aware. For example, to share the content of a news APP to a friend by short message, this sharing operation involves the user to click to confirm the sending of short messages, we shouldn't think that there is a leakage of the ability to send short messages, because it is ultimately up to the user to decide whether to send or not. However, it is illegal to disclose the ability of sending short messages without UI method, which has serious harmfulness. So this paper considers that UI method will not cause a capability leakage.

II. SYSTEM OVERVIEW

As shown in Figure 1, the tool includes two parts: static analysis and dynamic testing.

(1) Static analysis of the detected APP is carried out to find the control statements related to the intent data flow in and out of the detected APP, to find the set of variables (briefly described as key variables) used in the control statements, to generate Log instrumented statements that print key variables, and to insert the Log instrumented statements before the control statement blocks. At the same time, this paper finds statements protected by Android privileges, insert the Log statement before it, record the statement information protected by Android privileges, and then repackage the signed APP to get the instrumented APP (Figure 1 instrumented APP).

(2) The testing APP (Figure 1 testing APP, without any privileges) will dynamically test the instrumented APP by sending intent objects. According to the value of key variables in Log, new intent test cases are generated, which can trigger more code and improve the code coverage. If the statement information protected by Android privileges appears in the Log, it indicates that the privilege capability is leaked. Next, we will elaborate on two parts.

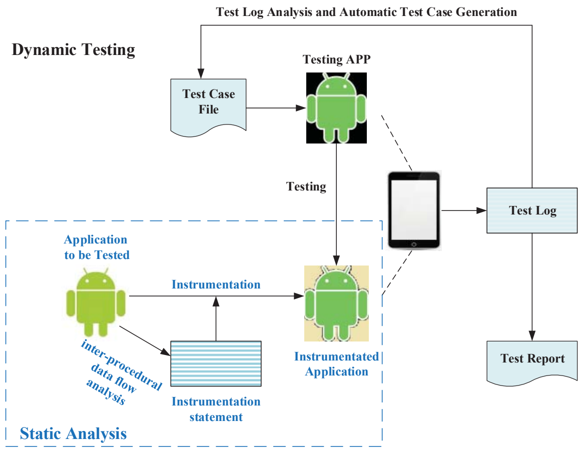
Test Log Analysis and Automatic Test Case Generation
Dynamic Testing
Testing APP
Test Case File
Testing
Test Log
Application to be Tested
Instrumentation
Instrumentated Application
Test Report
inter-procedural data flow analysis
Instrumentation statement
Static Analysis

Figure 1 Flow chart

A. Static Analysis

Our tool builds method call diagram and control flow diagram of each method based on Soot [10]. Soot is a Java bytecode [11] analysis and optimization framework, which supports the conversion of Java bytecode into multiple intermediate languages. This paper uses Soot framework to transform the application to be detected into Jimple [12] intermediate code with three address codes for analysis.

There are many implicit calls in Android applications, as shown in Figure 2. StartActivity (intent) is a calling method between Android components. Its function is to start activityA. First, StartActivity (intent) calls the Android system API, and finally the Android system API calls the activityA.onCreate () method. But we can't get the call relationship between startActivity (intent) and activityA.onCreate when we statically analyze Android APP and build method call graph by Soot alone. Therefore, in the process of constructing method call graph, this paper identifies hidden callback methods in Android APK and adds them to the Android method call graph until the Android method call graph no longer changes (that is, all callback callbacks in the current method call graph have been added to the method call graph). The proposed method is similar to FlowDroid [13] and IccTa [14]. Let n be the number of nodes in the complete method call graph and K() be the number of methods with callbacks. The algorithm complexity of constructing a complete call graph is O(k*n).

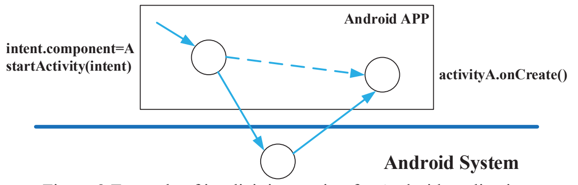

Figure 2 Example of implicit invocation for Android application

This paper uses API signature and privilege mapping APIPermissionMap file [16] provided by Android malware analysis tool androguard [15] to identify privileged statements in Android applications, and save these statements in tgtAPISet.

Algorithm1 Inter-process Data Flow Analysis Algorithms-Arrival Definition

```
Input: $method, inData$
Output:
        $flowInUnitDataMap(unit \to flowInDataSet)$, $returnData$
1 Function inter-procedure-data-flow:
2  $cfgNodes \leftarrow method.cfgNodes()$;
3  for n in cfgNodes do
4    $OUT[n] = \emptyset$;
5  end
6  $f \leftarrow cfgNodes.getFirstNode()$;
7  $IN[f] \leftarrow IN[f] \cup inData$;
8  $changed \leftarrow cfgNodes$;
9  while $changed \neq \emptyset$ do
10   $choose\ a\ node\ n\ in\ changed$;
11   $changed = changed - n$;
12   for all nodes p in predecessors(n) do
13     $IN[n] \leftarrow IN[n] \cup OUT[p]$;
14   end
15   $OldOUT \leftarrow OUT[n]$;
16   $OUT[n] \leftarrow transfer\_function$
         $(IN, n, flowInUnitDataMap)$;
17   if $OldOUT \neq OUT[n]$ then
18     for all nodes s in successors(n) do
19       $changed \leftarrow changed \cup s$;
20     end
21   end
22 end
23 $l \leftarrow cfgNodes.getReurnNode()$;
24 if l.returnLocal in IN[l] then
25   $returnData \leftarrow l.returnLocal.data$;
26 end
27 End Function
```

Starting from the starting point of external intent data flow (the method of obtaining external intent objects, such as activity.getIntent () method), this paper uses context-sensitive and flow-sensitive inter-process data flow analysis to find all statements related to external intent. The implementation of inter-process data flow analysis algorithm is mainly composed of algorithm 1 and algorithm 2, which mainly uses ***arrival definition*** data flow analysis technology and DFS algorithm.

Algorithm 2 Inter-process Data Flow Analysis——Transfer Function

```
Input: $IN, n, flowInUnitDataMap$
Output: $OUT[n]$
1 Function transfer_function:
2   $KILL[n] \leftarrow \emptyset$;
3   $GEN[n] \leftarrow \emptyset$;
4   $useLocals \leftarrow n.getUsedLocals()$;
5   $defLocal \leftarrow n.getDefLocal()$;
6   if $useLocals \cap IN[n] \neq \emptyset$ then
7     $GEN[n] = GEN[n] \cup defLocal$;
8     $flowInUnitDataMap.put(n.unit, intentData)$;
9   else
10    $KILL[n] = KILL[n] \cup defLocal$
11  end
12  if $defLocal \neq null$ then
13    if $m=getMethodCall(n) \neq null$ then
14      if $Pair(m,arg)$ not in hasProcessedMethodSummarySet then
15        $returnData \leftarrow inter\text{-}procedure\text{-}data\text{-}flow(m, arg.data).returnData$;
16        if $returnData \neq null$ then
17          $GEN[n] \leftarrow GEN[n] \cup defLocal$
18        end
19        $hasProcessedMethodSummarySet.add(Pair(m, arg))$;
20      end
21    end
22  else
23    if $(m=getMethodCall(n)) \neq null$ then
24      if $m$ not in hasProcessedMethodSummarySet then
25        $returnData \leftarrow inter\text{-}procedure\text{-}data\text{-}flow(m, arg.data).returnData$;
26        $hasProcessedMethodSummarySet.add(Pair(m, arg))$;
27      end
28    end
29  end
30  $OUT[n] = GEN[n] \cup (IN[n] - KILL[n])$;
31 End Function
```

Each method corresponds to a control flow graph (CFG), and the statements in each method correspond to a node in the CFG. Each node n has set of IN and OUT, which represent the set of variables related to intent data before node n and the set of variables related to intent data after node n executes. After each node is actually executed, the set of variables associated with intent data changes, which can be calculated by the following formula: $OUT[n] = GEN[n] \cup (IN[n] - KILL[n])$. Here $GEN[n]$ is the set of variables associated with intent data added after the node is executed, $KILL[n]$ is the set of variables that are reassigned after executing this node and are not related to intent data. The formula is implemented by the transfer_function function (line 16 of algorithm 1, details of implementation are shown in algorithm 2). IN[n] of each node is the union of OUT[n] sets of all its predecessor nodes (algorithm 12 to 14 rows). Simulate the execution of each statement (that is, the transfer_function function) until all nodes of $OUT[n]$ do not change, and eventually all statements associated with external intent data will be obtained.

The transfer_function mainly analyzes whether intent data related variables are used in node n. If the intent data-related variables are used, the assigned variables in the node are considered intent-related (lines 6 to 7 of algorithm 2). If the node contains a method call, it enters the method to call algorithm 1 again for analysis (line 15, line 25 of algorithm 2). For a method that has different data flows in different call contexts, at the call point a copy of the original function is created to consider different types of data flow input. Because only intent-related data streams are considered in this paper, the input types of parameters of the methods need to determined and the input types of intent data flows are finite (Intent, Action, Data, Category, Extras), the number of replicates created by the methods is limited. Therefore, the clone-based context-sensitive inter-process data flow analysis can ensure the accuracy of data flow analysis without causing significant performance overhead.

If the return value of this method is related to intent data, *defLocal* is added to $GEN[n]$ (line 17 of algorithm 2). Each node and intent data for each incoming node are put in *flowInUnitDataMap* (line 8 of algorithm 2). Intent data record their data types, including intent objects, intent action attributes, intent category attributes, intent extras attributes and so on. Querying flowInUnitDataMap to find all control statements if and intent data that flows into control statements, and they are stored in *ifControlDataMap* (ifUnit→intentData).

Through the above-mentioned, the set of statements protected by privileges tgtAPISet and the set of control statements related to intent data ifControlDataMap can be obtained. By iterating the set of tgtAPISet, the Log statements which print the corresponding permissions of tgtAPI and tgtAPI and the information of the APP where they are located are generated, and Log statements are instrumented before the tgtAPI. Iterating ifControlDataMap, the Log statements are inserted before "if" to print key variables and the attributes of intentData data which flowing into "if". If the data attribute is Extra, a Log statement that prints the key variable is inserted before the intentData source statement get*Extra (key). After the instrumentation is completed, the signature APP is repackaged and the instrumented APP is obtained. Therefore, when the instrumented APP runs, we can get the running logs related to intentData data.

It should be noted that the reinforcement technology [17] and the anti-re-packaging technology are becoming more and more popular, which results in the application of static analysis can not get the real application code, and the application of re-packaging can not run properly. However, the tool in this paper is for developers, who can use it before the application is released (before using consolidation and repackaging technology). Therefore, our tool is still valid.

B. Dynamic Analysis

Algorithms 3 is the test case generation algorithm.

Algorithm 3 Test Case Generation Method Based on Dynamic Feedback

```
Input: $detected\text{-}app$
Output: $capabilityLeakSet$
1 Function Main:
2   $ECSet \leftarrow getEC$
       ($detected\text{-}app.AndroidManifestXml$);
3   $capabilityLeakSet \leftarrow \emptyset$;
4   for exported-component in ECSet do
5     $actionSet, dataSet,$
          $categorySet, extraSet \leftarrow \emptyset$;
6     while true do
7       if $isFirstTest$ then
8         $initial\text{-}intent = newIntent$
              ($exported\text{-}component$);
9         $logFile \leftarrow testApp$
              ($detected\text{-}app, initial\text{-}intent$);
10      else
11        $selectCategorySet$
              $.add(categorySet)$;
12        $selectExtraSet \leftarrow$
              $combineWithDiffKey$
              $AndType(extraSet)$;
13        for a in actionSet do
14          for d in dataSet do
15            for c in selectCategorySet do
16              for e in selectExtraSet do
17                $intent = newIntent$
                      (a,d,c,e,exported-component);
18                if hasNotTested(intent) then
19                  $logFile \leftarrow testApp$
                        ($detected\text{-}app, intent$);
20                end
21              end
22            end
23          end
24        end
25      end
26      $oneTestCLSet$,intent-test-info
            $= analyseLog(logFile)$;
27      $capabilityLeakSet \leftarrow$
            $capabilityLeakSet \cup oneTestCLSet$;
28      if intent-test-info $\neq \emptyset$ then
29        actionSet,dataSet,categorySet,extraSet
30        .addAll(intent-test-info,
              mutation(intent-test-info));
31      else
32        break;
33      end
34    end
35  end
36 End Function
```

We initially test APP with intent objects without any data (7-9 lines of algorithm 3), and then analyze the generated test log. If intent-test-info is empty, that is to say, the actionSet, categorySet, dataSet and extraSet of intent-test-info are empty, the test is stopped, and the next exposed component testing continues. Otherwise, new information is added to the intent attribute set (line 29 of algorithm 3). At the same time, for the new extra attribute, we mutate it to generate the extra attribute that may satisfy the control statement (line 30 of algorithm 3).

For example, the new obtained extra attribute information is as follows.

key: "fromPush", type: int, value: 0.

Because we don't know the judgment condition of the control statement, two other extra attributes which may satisfy the condition of the control statement are generated.

1) key: "fromPush", type: int, value: 1

2) key: "fromPush", type: int, value: -1

Then the test cases are regrouped (11 to 24 lines of algorithm 3). The Category attribute in intent object is Set < String>. In this paper, all possible category values are taken as the Category attribute of intent object (line 11 of algorithm 3). The intent extra attribute is Set < key, type→value>. This paper divides the set extraSet attributes of all possible extra values into different sets according to key and type, and combines one value from these different sets into an intent extra attribute at a time (line 12 of algorithm 3). From line 13 to line 24, arithmetic 3 generates a test case to test APP, and records the test cases that have been tested to ensure that the test cases are not repeated. Arithmetic 3 continuously generates new test case tests based on intent-test-info of the test log until intent-test-info is empty.

III. Experimental Analysis and Evaluation

We selected the most popular applications of Wandoujia in 2017. There are 810 selected applications, including 18 categories and of the 45 most popular applications in each category. We removed the application of reinforcement and Soot analysis failure [18], and finally 607 applications were selected.

This paper chooses IntentFuzzer as the contrast of the dynamic test of capability leakage. Because the author could not be contacted, IntentFuzzer is implemented according to its paper. The four attributes of IntentFuzzer intent test case are constructed as follows.

(1) IntentFuzzer's intent action construction includes three aspects: one is to expose the action value in intent-filter of components, the next is to find strings prefixed by the application package name from all strings of APP, and the other is the standard action defined by all Android systems. IntentFuzzer uses the above action set as a candidate set of action attributes for test cases.

(2) IntentFuzzer predefines some URIs of common data types. When testing APP, if the predefined URI matches the intent-filter of exposed components, the URI is used to construct the data attributes of intent test cases.

(3) IntentFuzzer achieves key and type of extra attribute in dynamic testing by modifying the source code of Android system, and generates value randomly. In this way, the extra attribute of intent test case is constructed.

(4) IntentFuzzer does not consider the category attribute, and the Category attribute of intent test case is always empty.

A. Experimental Results

As shown in Table 1, a total of 6,070 in 16 kinds of capability leakages were found. The first column of Table 1 is the type of capability leakage, the second column is the number of APPs with this type of capability leakage, and the third column is the number of capability leakage points (location of capability leakage, i.e. tgtAPI location) in all APPs. There are serious capability leakages, such as DISABLE_KEYGUARD privilege ability leakage which is the main privilege to achieve the lock screen function, and KILL_BACKGROUND_PROCESSES privilege ability leakage which is the privilege to achieve the killing of background processes. There are also vulnerabilities with less harmful capability leakages, such as BROADCAST_STICKY capability leakage which will lead to application broadcasting not working properly, ACCESS_FINE_LOCATION capability leakage which may lead to application power consumption problems, and BLUETOOTH capability leakage which will lead to arbitrary turn on and off mobile Bluetooth.

Table I Experimental results

Permission	AppUseCount	AllCount
DISABLE_KEYGUARD	6	7
CHANGE_WIFI_MULTICAST_STATE	2	2
RECEIVE_BOOT_COMPLETED	1	2
SET_WALLPAPER_HINTS	3	3
BROADCAST_STICKY	169	261
ACCESS_FINE_LOCATION	140	454
KILL_BACKGROUND_PROCESSES	4	5
ACCESS_COARSE_LOCATION	126	303
CHANGE_WIFI_STATE	3	4
GET_TASKS	261	626
ACCESS_NETWORK_STATE	405	3201
WAKE_LOCK	99	187
ACCESS_WIFI_STATE	294	928
MODIFY_AUDIO_SETTINGS	4	4
BLUETOOTH	7	10
READ_PHONE_STATE	48	73

The following formulas are used as indicators of false negative rate of evaluation tools. Let the test APP set be AppSet and the size be n. For APP A_i, we assume that its cability leakage set is $CLSet_i$ and the size is s_i。 We utilize tool t to test APP A_i. The set of capability leakage points for detecting P_j is PS_{ij}^t 。 For the ability leakage P_j of APP A_i, the detection advantage ratio of tool t1 to tool t2 is:

$$G_{t_1t_2}(A_i, P_j) = \frac{((PS_{ij}^{t_1} - PS_{ij}^{t_1} \cap PS_{ij}^{t_2}) - (PS_{ij}^{t_2} - PS_{ij}^{t_1} \cap PS_{ij}^{t_2}))}{PS_{ij}^{t_1} + PS_{ij}^{t_2} - PS_{ij}^{t_1} \cap PS_{ij}^{t_2}}$$

It is the proportion of G to H, here G is the difference between the P_j permission leakage results detected by tool t1 and tool t2, H is the total result of the two tools detecting APP A_i P_j permission leakage results. Therefore, the average detection advantage ratio of tool t1 to tool t2 is:

$$G_{t_1t_2} = \frac{\sum_{i=0}^{n} \sum_{j=0}^{s_i} G_{t_1t_2}(A_i, P_j)}{n * s_i}$$

The results of 607 APPs detected by this tool and IntentFuzzer are calculated according to the above formulas to get Table 2.

Table II Comparing with IntentFuzzer results

Min($G_{t_1t_2}(A_i, P_j)$)	Max($G_{t_1t_2}(A_i, P_j)$)	$G_{t_1t_2}$
0%	100%	19.38%

Among them, t1 is the tool of this paper and t2 is IntentFuzzer. According to Table 2, the average test results of this tool are 19.38% better than those of IntentFuzzer. For a single APP, the ability to detect permission capability leakage is up to 100%. That is to say, the tool can detects that the permission has the capability leakage, while IntentFuzzer does not detect it. Or the results of IntentFuzzer are included in the results of this tool. The worst case of this tool is the same result as that of IntentFuzzer. Therefore, it can be seen that the Android inter-application capability leakage detection tool proposed in this paper is completely superior to IntentFuzzer.

B. Time Efficiency

Table 3 is the time consumption of data flow analysis, instrumentation and dynamic testing during the analysis of 607 APPs.

Table III Running time

	Min	Max	Average
data flow analysis	0.06s	161.70s (2.70min)	25.40s
instrumentation	0.15s	92.32s (1.54min)	20.21s
dynamic testing	7.46s	2567.09s (42.78min)	507.94s (8.47min)

Among them, the shortest time of data flow analysis is 0.06s, the longest time is 2.70 min, and the average time of data flow analysis is 25.40 s per APP. The shortest time of instrumentation is 0.15s, the longest time is 1.54min, and the average time of pile insertion is 20.21s. The shortest dynamic testing time was 7.46 seconds, the longest time was 42.78 minutes, and the average dynamic testing time per APP was 8.47 minutes. Therefore, the average detection time per APP is about 9 minutes, which meets the actual time efficiency requirements. For some individual APP dynamic testing time is very large, reaching 42.78 minutes. But this time is still acceptable. Different exposed components of APP deal with external intent differently, which leads to different information obtained by dynamic testing of each APP. Therefore, the number of test cases generated by dynamic testing is different, and the number of exposed components of different APPs is different, so the time of dynamic testing of different APPs may vary greatly.

C. Examples of Exploiting Capability leakage Vulnerabilities

Application A is a very popular lock screen application, which has been downloaded more than 10 million times. The tool in this paper detects that it has DISABLE_KEYGUARD capability leakage, so we guess that it has illegal access vulnerabilities. We use the test cases generated by this tool to trigger the DISABLE_KEYGUARD privilege leak and find that keyboard locks can be crossed without passwords. Attack Demo Video can be found on Youku [19].

Application B is a cleanup software, which has KILL_BACKGROUND_PROCESSES capability leakage. It can be used to kill background applications. The attack demonstration video can be found on Youku [20].

IV. Conclusion and Prospect

This paper presents an automatic detection tool based on dynamic feedback for capability leakage vulnerabilities between Android applications. Through context-sensitive and flow-sensitive inter-process data flow analysis, the location and key variables of pile insertion can be found, and the instrumented APP can be automatically generated.

The tool continuously uses the results of dynamic testing of instrumented APP, adjusts the test case to test APP, and can greatly improve the code coverage of the test. Compared with the existing capability leak vulnerability detection work, the tool in this paper has a lower missing rate, and the test results are better than 19.38%. At the same time, this tool has detected 16 kinds of 6070 capability leakage vulnerabilities of 607 APPs in Wandoujia, including some serious capability leakage vulnerabilities.

The capability leakage vulnerability evaluation is helpful for developers to quickly repair serious capability leakage vulnerabilities. Therefore, we may take the evaluation of capability leakage vulnerability as one of our future research work.

Acknowledgment

This work is supported partly by the National Key R&D Program of China 2018YFB0803400, 2018YFB2100300 and National Natural Science Foundation of China (NSFC) under grant 61772487.

References

[1] FELT A P, WANG H J, MOSHCHUK A, et al. Permission re-delegation: Attacks and defenses[C/OL]//20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011,Proceedings.2011.

[2] YAN J, DENG X, WANG P, et al. Characterizing and identifying misexposed activities in android applications[C/OL]//Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018.2018:691-701

[3] CHIN E, FELT A P, GREENWOOD K, et al. Analyzing inter-application communication in android[C/OL]//Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys 2011), Bethesda, MD, USA, June 28 - July 01, 2011. 2011:239-252.

[4] LI L, BARTEL A, KLEIN J, et al. Automatically exploiting potential component leaks in android applications[C/OL]//13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2014, Beijing, China, September 24-26, 2014. 2014: 388-397.

[5] HE Y, LI Q. Detecting and defending against inter-app permission leaks in android apps[C/OL]//35th IEEE International Performance Computing and Communications Conference,IPCCC 2016, Las Vegas, NV, USA, December 9-11, 2016. 2016: 1-7.

[6] XIE J, FU X, DU X, et al. Autopatchdroid: A framework for patching inter-app vulnerabilities in android application[C/OL]//IEEE International Conference on Communications, ICC 2017,Paris, France, May 21-25, 2017. 2017: 1-6.

[7] LIU F, CAI H, WANG G, et al. Mr-droid: A scalable and prioritized analysis of inter-app communication risks[C/OL]//2017 IEEE Security and Privacy Workshops, SP Workshops 2017,San Jose, CA, USA, May 25, 2017. 2017: 189-198.

[8] YANG K, ZHUGE J, WANG Y, et al. Intentfuzzer: detecting capability leaks of android applications[C/OL]//9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS ’14, Kyoto, Japan - June 03 - 06, 2014. 2014:531-536.

[9] DEMISSIE B F, GHIO D, CECCATO M, et al. Identifying android inter app communication vulnerabilities using static and dynamic analysis[C/OL]//Proceedings of the International Conference on Mobile Software Engineering and Systems, MOBILESoft ’16, Austin, Texas,USA, May 14-22, 2016. 2016: 255-266.

[10] Soot[EB/OL]. https://sable.github.io/soot/.

[11] WIKI. Java bytecode[EB/OL]. 2019-03-20. https://en.wikipedia.org/wiki/Java_bytecode.

[12] VALLEE-RAI R, HENDREN L J. Jimple: Simplifying java bytecode for analyses and transformations[Z]. 1998.

[13] ARZT S, RASTHOFER S, FRITZ C, et al. Flowdroid: precise context, flow, field, objectsensitive and lifecycle-aware taint analysis for android apps[C/OL]//ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’14, Edinburgh, United Kingdom - June 09 - 11, 2014. 2014: 259-269.

[14] LI L, BARTEL A, BISSYANDÉ T F, et al. Iccta: Detecting inter-component privacy leaks in android apps[C/OL]//37th IEEE/ACM International Conference on Software Engineering,ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1. 2015: 280-291.

[15] DESNOS A, et al. Androguard: Reverse engineering, malware and goodware analysis of android applications[J]. URL code. google. com/p/androguard, 2013:153.

[16] ANDROGUARD. Android permission api mappings[EB/OL]. 2019-03-20. https://github.com/androguard/androguard/blob/master/androguard/core/api_specific_resources/api_permission_mappings/permissions_25.json.

[17] Dexguard[EB/OL]. https://www.guardsquare.com/en/products/dexguard

[18] BARTEL A, KLEIN J, TRAON Y L, et al. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot[C/OL]//Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, SOAP 2012, Beijing, China, June 14,2012. 2012: 27-38.

[19] https://v.youku.com/v_show/id_XNDExOTg1ODA3Mg

[20] https://v.youku.com/v_show/id_XNDExOTg2MDAxMg