2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)

Automatic Generation of Capability Leaks' Exploits for Android Applications

Mingsong Zhou, Fanping Zeng, Yu Zhang, Chengcheng Lv, Zhao Chen, Guozhu Chen
University of Science and Technology of China
Hefei, Anhui, China
billzeng@ustc.edu.cn

***Abstract*—The capability leak of Android applications is one kind of serious vulnerability. It causes other apps to leverage its functions to achieve their illegal goals. In this paper, we propose a tool which can automatically generate capability leaks' exploits of Android applications with path-sensitive symbolic execution-based static analysis and test. It can aid in reducing false positives of vulnerability analysis and help engineers find bugs. We utilize control flow graph (CFG) reduction and call graph (CG) search optimization to optimize symbolic execution, which make our tool applicable for practical apps. By applying our tool to 439 popular applications of the Wandoujia (a famous app market in China) in 2017, we found 2239 capability leaks of 16 kinds of permissions. And the average analysis time was 4 minutes per app. A demo video can be found at the website https://youtu.be/dXFMNZWxEc0.**

***Index Terms*—capability leak, Android, inter-component communication, symbolic execution**

I. Introduction

Capability Leak, also known as Permission Re-Delegation [1], occurs when a vulnerable application performs a privileged action on behalf of a malicious application without permission. Inter-component communication between Android applications is common. A lot of apps provide some special functions for other apps by exported components. However, many developers do not fully understand the confused use rules in Android application components. They either expose the components unintentionally [2], or expose them intentionally but fail to check the component caller's permissions. It causes several security problems. For example, a capability leak MASTER_CLEAR is found in Samsung Epic 4Gs phone image [3]. It is easy to delete all user data by constructing an intent. Therefore, research of capability leaks of Android applications is important and significant.

In this paper, we elaborate capability leak of Android applications as follows: if there is an app B, without permission p, can invoke A's code protected by permission p directly (without user's UI operation) from A's exported components, we say that app A has a permission p capability leak. In our paper, we take into account all APIs protected by permissions even if external intent data do not flow in these APIs. Because some APIs do not need any parameters and APIs without external input data can also cause immense destruction.

Our main contributions are as follows:

1) We propose a tool which can automatically generate capability leaks' exploits of Android applications.
2) We utilize CFG reduction and CG search optimization to optimize symbolic execution, which make our tool can apply to practical apps.
3) We analyzed 439 popular apps of various categories. And we found 2239 capability leaks of 16 permissions, including some very serious capability leaks.

II. System Overview

Figure 1 depicts an overview of our work, which is mainly divided into four parts. In the first part, we extract app's call graph , control flow graphs for each method and find all Android permission-protected APIs (i.e. *tgtAPI*). Then we reduce our CG by removing methods that are not in paths between *exported-component*s' methods (i.e. *startPoint*) and *tgtAPI*. In the second part, we find all paths between *startPoint* and *tgtAPI*, which represent all possible capability leak paths. We utilize CFG reduction and CG search optimization to optimize the process of finding paths. Then we extract the intent constraints of each path and convert these intent constraints into SMT2 language in the third part. Using the Microsoft Z3 constraint solver [4] to solve, we generate intent test cases based on the results of Z3. In the fourth part, *test-app* utilizes the intent test cases to launch the *instrumented app*. Then our tool reads the test log and generate the *detection report* of the *detected app*. The *detection report* includes capability leaks of *detected app* and exploits of these capability leaks. In the following sections, we will introduce each part in detail.

A. Extract CG and each method's CFG

To obtain a call graph suitable for analysis of Android apps, the call graph must take into account implicit calls of Android app. Android implicit calls include component lifecycle methods, callback methods, inter-component communication methods. Our tool is based on soot [5] and we use identical methods as described in previous works [6] [7], where the call graph is continuously updated with identified callback registrations until a fixed point is reached.

To know what permissions the API's invocation needs, we use the *APIPermissionMap* provided by the Androguard [8], which stores the map between Android permissions and the

978-1-7281-0888-9/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSTW.2019.00068

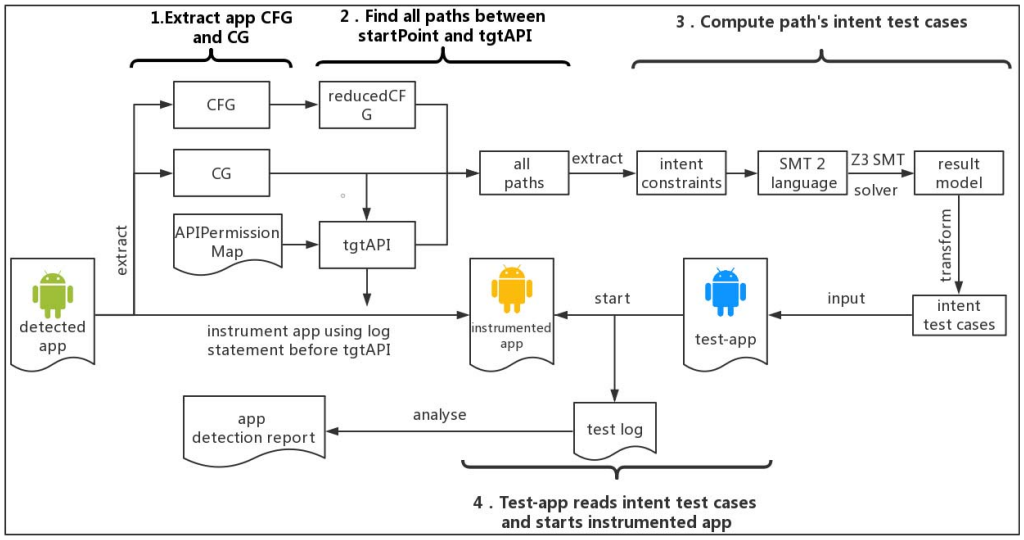

Fig. 1. System overview

Android API. We utilize it to find all Android permission-protected APIs (*tgtAPIs*). Then we utilize backward traversal from *tgtAPI* and forward traversal from *startPoint* to remove nodes (methods) not in the paths between *startPoint* and *tgtAPI*, which can substantially reduce the search scope when find all paths between *startPoint* and *tgtAPI*.

B. Get Intent Conditions of Paths between StartPoint and TgtAPI

CG and CFG are directed cyclic graph. In our paper, the paths that we search between *startPoint* and *tgtAPI* are Eulerian path. To reduce the scale of problem, we optimize search process and reduce each method's CFG by data-flow analysis.

1) CG Search Optimization: Suppose there is a path $ABCDE$, where A, B, C, D and E are methods. A is *startPoint* and E is *tgtAPI*. At First, we use the way of section II-B2 to reduce CFG of each method. Then we compute all intent constraints of all paths between two methods (i.e. A and B, B and C, C and D, D and E). And we utilize Z3 to solve all intent conditions of all paths between methods (named *intentConditionSet*). This process will be introduced in section II-C in detail. *intentConditionSet* will be saved and use repeatedly. It will save a lot of time, because an edge (method invocation) will be in multiple paths. Finally, we get intent conditions of this path (i.e. $ABCDE$) by using intersection operation for these *intentConditionSet*s. When we compute intent conditions of a path, we merge identical intent conditions and remove conflict intent conditions. This reduces the number of intent conditions that need to be combined when we search a path and it avoids duplicate and useless intent test cases. We also use function summary which stores all possible intent conditions of paths from current method to *tgtAPI*. When the method is analyzed again, the result is taken directly and do not need further analysis.

```
// before reduction
    public void doTask1(String pName) {
        int pid=643;
        if (x > 5)
        {
            if (intent.getAction.equals("kill"))
            {   //branch1
                String key="pid";
                pid=intent.getIntExtra(key);
                ...
            } else {...}//branch 2
        }
        else
            {
                if (y < 6) {...}//branch 3
                else {...}//branch 4
                if (z > 7) {...}//branch 5
                else {...}//branch 6
            }
        killProcess(pName,pid);// tgtAPI
    }
// after reduction
    public void doTask2(String pName) {
        int pid = 643;
        if(x>5)
        {
            if (intent.getAction.equals("kill"))
            {   //branch 1
                String key="pid";
                pid=intent.getIntExtra(key);
            }else {}//branch 2
        }
        else {}//branch 3
        killProcess(pName,pid);//tgtAPI
    }
```

Listing 1. Reduce CFG

2) CFG Reduction: In our paper, we only focus on statements related to external intent (intent from another application), because the unique input of inter-component communication is intent. We utilize a light-weight inter-procedural data-flow analysis from *sources* [9] (only use *sources* related to external intent, for example: getIntent(), onReceive(Intent)) to remove the statements that are not

related to the external intent. Our inter-procedural data-flow is mainly based on the reaching definition technique [10], which focuses on statically determining which definitions may reach a given point in the code. But we do not remove intent-irrelevant condition statements if their branch statements contain intent-relevant statements. For example, as shown in Listing 1, the statement $if(x > 5)$ can not be removed because the true branch of $if(x > 5)$ has intent-relevant statements. But we can remove $if(y < 6)$ and $if(z > 7)$. We also reserve statements that intent-relevant statements depend. For example, the statement $String\ \ key\ =\ "pid"$ will not be removed in listing 1. Since most statements are not related to intent, it is possible to substantially reduce the CFG. Thus, our method is efficient. As shown in Listing 1, there are 6 branches in total 6 paths in the *doTask1* method. But none of the 5 branches have any statements related to intent data, so we can simplify *doTask1* method into *doTask2* method and *doTask2* method only has 3 branches in total 3 paths.

C. Compute Path's Intent Constraints

Z3 is a state-of-the-art theorem prover from Microsoft Research. It can be used to check the satisfiability of logical formulas over one or more theories. We utilize Z3 to solve intent conditions. At first, we get all statements of a path. Then we will process statements separately to generate intent constraints in format of SMT2 language. These statements mainly include $intent.getAction$,$intent.hasCategory$, $intent.get*Extra$, $equals$, if, variable definition and other operation statements about value of intent attribute. For example, if statement is $if(str.equals("success"))$, the SMT2 language is $(assert\ \ (=\ \ str\ \ "success"))$. Z3 will return the value of str is "$success$". Our tool can get correct string and primitive values in most cases when compute path's intent constraints.

D. Test-app Test Instrumented App

The intent test cases that we generate satisfy the intent conditions of paths, but there are some other conditions of paths that we can not control. So we need use dynamic way to test whether paths are reachable. We insert log statements before *tgtAPI*s in *detected app* and repackage *detected app* as a new app (named *instrumented app*). And the log statement mainly records these information: test case number, *exported-component*'s name, package name, *tgtAPI* and etc. Then we develop *test-app* which do not have any permissions and it utilizes test cases to test *instrumented app*. If the *tgtAPI* can be triggered, the log will be generated, indicating that the app occurs a capability leak. Then our tool reads the test log to generate a *detection report* for *detected app*. The exploits of capability leaks in *detection report* can trigger the corresponding capability leak, and they can help developers analyse bugs.

III. Evaluation

To assess our work, we study the following research questions:

RQ 1: What is the accuracy of path's intent conditions we generate?

RQ 2: Can our tool be applied to practical apps and find capability leaks ? Can exploits help developers find bugs?

RQ 3: Our tool uses symbolic execution, what is runtime efficiency of our work?

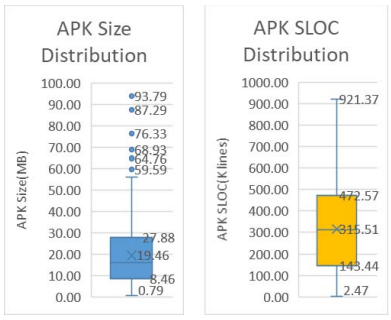

Fig. 2. APK Size and SLOC Distribution

The following experiment results are collected on Ubuntu 18.04 with a 3.6GHz Intel Core i7 CPU and 32GB RAM. Our dataset consists of 18 categories of applications from Wandoujia in 2017. We select 45 most popular apps for each category and in total 810 apps, then remove hardened apps [11] and apps that soot can not analyze [12]. Finally, we get 439 apps. These apps size and SLOC(source lines of code) are shown in figure 2. What we need to explain is that the use of harden technology is becoming more and more popular, which causes that we can not get the real source code of app. Fortunately, our tool is proposed for developers, and developers can use our tool to detect the app before it is hardened, so our tool is still useful. To answer **RQ1**, we divide 439 apps into five categories (0-9M,9M-18M,18M-27M,27M-36M,36M-) according to app size. We select 5 apps randomly from each category in total 25 apps to evaluate our tool.

A. RQ1: Accuracy of Path's Intent Conditions

We run our tool for these 25 apps and record the statements that we can not handle (named *unhandledSet*). At the same time, we manually check each statement we can handle and record the statements that we can not get correct value (named *incorrectSet*). For example, a string value from network. *allStatementsSet* is a set of all intent-relevant statements. We use the following correctness metric to access the accuracy of intent conditions that we generate:

$$\frac{size(allStatementsSet)-size(unhandledSet)-size(incorrectSet)}{size(allstatementsSet)}$$

The accuracy results are shown in figure 3. The accuracy of path's intent conditions is high and no app has a rate lower than 90%. This indicates that for the overwhelming majority of cases, our tool can generate correct intent conditions.

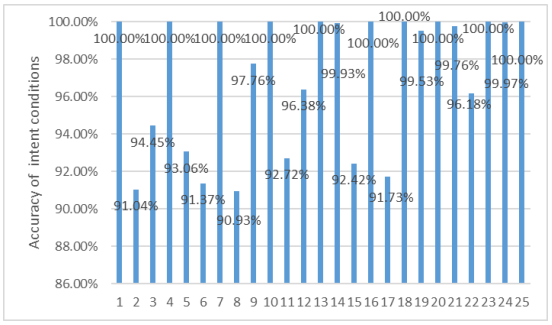

Fig. 3. Accuracy of Intent Conditions

B. RQ2: Can our tool apply to practical apps?

1) Experiment Results: The analysis result of 439 apps is shown in Table I, For each capability leak, we counted the number of apps have the capability leak (i.e. App Count column) and the number of the capability leak's points in all apps (i.e. All Count column). There are 2239 capability leaks of 16 kinds of permissions, including some serious capability leaks, such as DISABLE_KEYGUARD, KILL_BACKGROUND_PROCESSES, MODIFY_AUDIO_SETTINGS and so on. Therefore, our tool can detect capability leaks efficiently.

TABLE I
CAPABILITY LEAKS LIST

Permission	App Count	All Count
DISABLE_KEYGUARD	8	9
CHANGE_WIFI_MULTICAST_STATE	1	1
SET_WALLPAPER_HINTS	4	4
BROADCAST_STICKY	84	84
ACCESS_FINE_LOCATION	106	180
ACCESS_COARSE_LOCATION	94	157
CHANGE_WIFI_STATE	3	4
ACCESS_NETWORK_STATE	323	1071
GET_TASKS	216	272
WAKE_LOCK	56	81
ACCESS_WIFI_STATE	227	318
MODIFY_AUDIO_SETTINGS	4	4
SET_WALLPAPER	1	1
BLUETOOTH	7	10
READ_PHONE_STATE	30	35
KILL_BACKGROUND_PROCESSES	7	8

2) Exploitation Analysis: App A is a popular lock screen app and has been downloaded more than 10 million times. We found that it has a DISABLE_KEYGUARD capability leak. We guess that there is an illegal login vulnerability. Then We use exploits generated by our tool to attack app and they help us pass the lock sceen without a password. The attack demo is on the youtube [1]. We have informed the app's developers. App B is a clean app, whose function is phone clean. And we found that it has KILL_BACKGROUND_PROCESSES capability leak. It may be used by other apps to kill processes. The attack demo is on the youtube [2]. Except for capability leaks, we found that a lot of apps crashed when we used our exploits to launch them, which is a kind of local denial of service attack. So these apps must be more robust, it may be leveraged by other apps for vicious competition. Therefore, our exploits are valid and help users find bugs.

C. RQ3: Runtime Efficiency

Table II presents static analysis part and dynamic test part's average, minimum, and maximum execution time of 439 apps. As shown in table II, The total average time for each app analysis is less than 4 minutes. The maximum time for static analysis app is 5168.494s, which is about 1.43h. 1.43h is a reasonable analysis time for generating highly precise intent test cases and few apps' static analysis is more than 16 min in our statistics. Therefore, our optimization for symbolic execution of inter-component capability leaks detection is efficient, and our tool meets the requirement of actual use.

TABLE II
EXECUTION-TIME

Period	Execution Time		
	Average	Minimum	Maximum
Static	185.228s(3min)	0.078s	5168.494s(1.43h)
Dynamic	52.984s	7.414s	889.919s(14.82min)

IV. Related Work

There are many static analysis works for detecting security problems of inter-component communication (for example: [13], [14]). But they all cannot determine whether the vulnerability really exists and developers have to spend much time in vulnerability analysis. Fang Liu et al. [15] proposed the MR-Droid to find inter-component communication vulnerabilities among practical apps, which uses the map-reduce system to detect communication vulnerabilities among large-scale apps. The results of the tool are limited by the dataset and it does not take into account malicious apps. Its result can not indicate the detected app is security. And [16] also has this problem.

V. Conclusion

We propose an effective tool which can automatically generate capability leaks' exploits of Android applications with symbolic execution and test. It can aid in reducing false positives of vulnerability analysis and help developers find bugs. Our tool can apply to practical apps because of our optimized symbolic execution. We analyzed 439 apps in Wandoujia and found 2239 capability leaks of 16 kinds of permission.

Acknowledgment

This work is mainly inspired by LetterBomb [17]. This work is supported partly by the National Key RD Program of China 2018YFB0803400 and National Natural Science Foundation of China (NSFC) under grant 61772487.

[1]https://youtu.be/rWdSiWUy2bc

[2]https://youtu.be/YE84G4yko0A

REFERENCES

[1] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*, 2011. [Online]. Available: http://static.usenix.org/events/sec11/tech/full_papers/Felt.pdf

[2] J. Yan, X. Deng, P. Wang, T. Wu, J. Yan, and J. Zhang, "Characterizing and identifying misexposed activities in android applications," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, 2018, pp. 691–701. [Online]. Available: https://doi.org/10.1145/3238147.3238164

[3] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones," in *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, 2012. [Online]. Available: https://www.ndss-symposium.org/ndss2012/systematic-detection-capability-leaks-stock-android-smartphones

[4] "Z3 wiki," https://github.com/Z3Prover/z3/wiki.

[5] "Soot," https://sable.github.io/soot/.

[6] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel, "Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, 2014, pp. 259–269. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594299

[7] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. D. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, 2015, pp. 280–291. [Online]. Available: https://doi.org/10.1109/ICSE.2015.48

[8] A. Desnos *et al.*, "Androguard: Reverse engineering, malware and goodware analysis of android applications," *URL code. google. com/p/androguard*, p. 153, 2013.

[9] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014. [Online]. Available: https://www.ndss-symposium.org/ndss2014/machine-learning-approach-classifying-and-categorizing-android-sources-and-sinks

[10] "Reaching definition wiki," https://en.wikipedia.org/wiki/Reaching_definition.

[11] "Dexguard," https://www.guardsquare.com/en/products/dexguard.

[12] A. Bartel, J. Klein, Y. L. Traon, and M. Monperrus, "Dexpler: converting android dalvik bytecode to jimple for static analysis with soot," in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, SOAP 2012, Beijing, China, June 14, 2012*, 2012, pp. 27–38. [Online]. Available: https://doi.org/10.1145/2259051.2259056

[13] Y. He and Q. Li, "Detecting and defending against inter-app permission leaks in android apps," in *35th IEEE International Performance Computing and Communications Conference, IPCCC 2016, Las Vegas, NV, USA, December 9-11, 2016*, 2016, pp. 1–7. [Online]. Available: https://doi.org/10.1109/PCCC.2016.7820624

[14] E. Chin, A. P. Felt, K. Greenwood, and D. A. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys 2011), Bethesda, MD, USA, June 28 - July 01, 2011*, 2011, pp. 239–252. [Online]. Available: http://doi.acm.org/10.1145/1999995.2000018

[15] F. Liu, H. Cai, G. Wang, D. Yao, K. O. Elish, and B. G. Ryder, "Mr-droid: A scalable and prioritized analysis of inter-app communication risks," in *2017 IEEE Security and Privacy Workshops, SP Workshops 2017, San Jose, CA, USA, May 25, 2017*, 2017, pp. 189–198. [Online]. Available: https://doi.org/10.1109/SPW.2017.12

[16] X. Zhong, F. Zeng, Z. Cheng, N. Xie, X. Qin, and S. Guo, "Privilege escalation detecting in android applications," in *3rd International Conference on Big Data Computing and Communications, BIGCOM 2017, Chengdu, China, August 10-11, 2017*, 2017, pp. 39–44. [Online]. Available: https://doi.org/10.1109/BIGCOM.2017.21

[17] J. Garcia, M. Hammad, N. Ghorbani, and S. Malek, "Automatic generation of inter-component communication exploits for android applications," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, 2017, pp. 661–671. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106286