SCIENCE CHINA
Information Sciences

• LETTER •

May 2018, Vol. 61 050107:1–050107:3
https://doi.org/10.1007/s11432-017-9354-x

Special Focus on Internetware: Research Progress and Current Trend

LogPruner: detect, analyze and prune logging calls in Android apps

Xin ZHOU, Kaidong WU, Huaqian CAI, Shuai LOU, Ying ZHANG & Gang HUANG*

School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China

Received 4 December 2017/Accepted 14 January 2018/Published online 18 April 2018

Citation Zhou X, Wu K D, Cai H Q, et al. LogPruner: detect, analyze and prune logging calls in Android apps. Sci China Inf Sci, 2018, 61(5): 050107, https://doi.org/10.1007/s11432-017-9354-x

Dear editor,
Android is a popular mobile operating system that is accounted for more than 87% of all smartphone sales in the second quarter of 2017. The number of available apps hosted in Google play store has reached to 3.5 M at the end of 2017 and is still counting[1]. In developing apps, developers usually use log messages for debugging and diagnosing their apps in order to fix bugs or locate the performance bottlenecks [1, 2]. At release stage, the logging messages should be muted for security consideration and higher performance. However, our previous empirical study revealed that developers prefer to deactivate the logging calls instead of removing them [3]. By applying reverse engineering, malicious attackers can reactivate logging calls and steal sensitive personal information from log messages [4, 5]. Although we have proposed a tool to prune the logging calls as well as its dependent instructions based on a user predefined logging class configuration [3], it is difficult for developers to locate the logging class when apps getting more complicated, especially in third-party libraries. We further complement the work by providing a logging class detection algorithm that intends to ease developers' work from scratch.

Given a released app in .apk format, LogPruner analyzes and optimizes the byte-code. It generates the optimized app, in which logging calls as well as their dependent instructions are removed. The detailed process of LogPruner is shown in Figure 1(a). We detail the processes as follows.

(1) Detecting logging classes. For app developers, they can easily specify the logging classes in their own apps. As for third-party logging classes, the developers can use features to achieve coarse screening, then manually identify the logging classes. We further propose LogDetector that intends to detect logging class automatically.

(2) Removing logging calls. We define the method calls whose owner is logging class as logging calls. LogPruner removes logging calls from the dex-format file by bytecode transformation. After the removal of logging calls, some objects that created only for the logging calls will be useless.

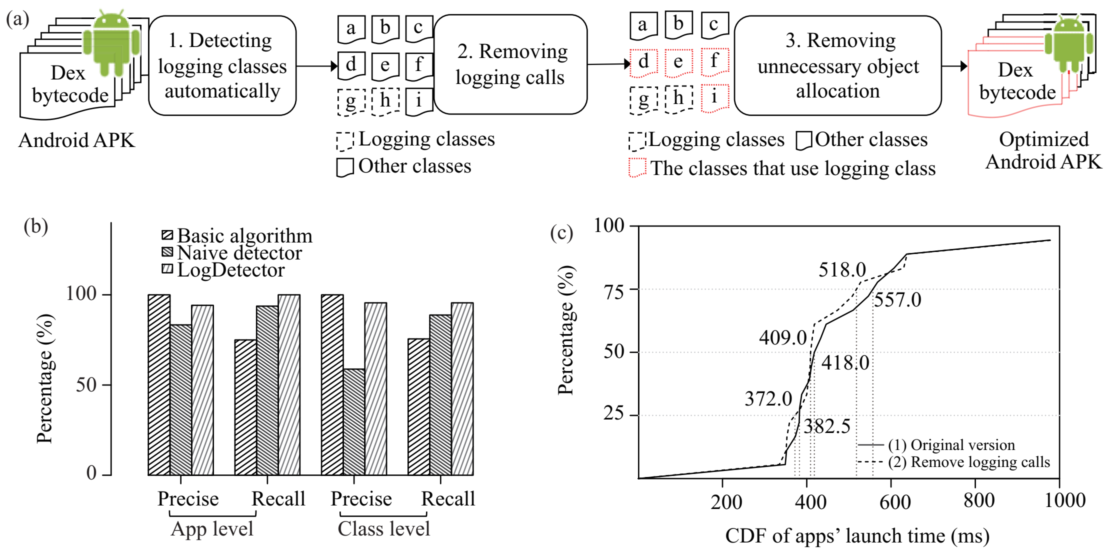
(3) Removing unnecessary instructions. Finally, LogPruner uses a variant version of live variable analysis to remove all the unnecessary instructions that construct the unnecessary object.

We examine the source code of logging libraries in Android arsenal. After inspecting more than 30 third-party logging libraries, we summarize the features of logging classes as follows.

(1) Compact methods. The methods in logging class usually have a small number of instructions because the computational logic is relatively simple.

(2) Low coupling and high cohesion. Meaning the logging class is the number of dependent

* Corresponding author (email: hg@pku.edu.cn)
1) https://www.statista.com/.



(d)

	Name	Description
Internal	ClassName	Name of the logging class
	UseInsideOnceCount	How many methods just use the other method once
	UseInsideTwiceCount	How many methods just use the other method twice
	InstructionCount	Instruction count per method
External	UseLogDirectly	How many methods use log directly
	UseLogIndirectly	How many methods use log indirectly
	UsedClassCount	How many classes uses the class (in package)
	UsedMethodCount	How many methods uses the class (in package)

Figure 1 (Color online) (a) Detailed processes of LogPruner; (b) performance of LogDetector; (c) performance of LogPruner on apps' launch time; (d) Log features exploited in LogDetector.

classes for logging class is relatively small.

The detailed features we filter to detect the logging classes are presented in Figure 1(d).

Logging detection is a classification problem essentially. That is, the detection algorithm classifies the logging classes by their features into two categories: logging classes and normal classes. Because the detection results will be used in pruning the logging calls later, the algorithm should have high precision.

Although many statistical learning algorithms can be used to differentiate the type of classes, all of them need a big training data-set to ensure accuracy. Hence, algorithms should achieve a trade-off between learning instances and accuracy. A larger amount of learning instances can improve the detection accuracy but take more time to exploit the available instances. We have studied the logging classes used in 78 open source projects and proposed a preliminary algorithm, LogDetector. The detailed processes are as follows.

(1) Checking class names. Some developers use third-party logging classes in their apps. Third-party jars are set in the build path of the project and logging classes are imported when required. Sometimes these classes can be detected by their name. For example, if com.umeng.fb.util.Log appears in the apk, we can label it as "AppLogger" directly. LogDetector maintains a list of famous third-party logging classes and prints the class into the result if found. Besides, the classes like "*log" or "*logger", which have "log" or "logger" in their package name may be logging classes. We will further check them in following two steps.

(2) Checking external features. Given a class, we first check its UseLogDirectly and UseLogIndirectly. Consider a normal class may have high UseLogIndirectly by calling methods in logging classes, we just check the class that has lower UseLogDirectly or higher UseLogIndirectly in the third step. UsedClassCount and UsedMethodCount will be checked, too. As a logging class, there will be many classes and methods use it, unless its methods are not called at all. So its

UsedClassCount and UsedMethodCount must be non-zero.

(3) Checking internal features. "AppLogger" created by developers is usually simple. It means lower UseInsideOnceCount and UseInsideTwiceCount. Except for methods like writing logs into file, logging methods are not complex as well. So classes which have lower InstructionCount may be a logging class.

To evaluate the performance of LogDetector, we test our detection algorithm on the set of 20 open-source apps. The evaluation is conducted on LogDetector and naive detector, their performances are compared with the basic algorithm. Naive detector just checks classes that have "log" or "logger" in their name or package name. The algorithm will select classes which have high UseLogDirectly or UseLogIndirectly of these to label as "AppLogger". The basic algorithm just prints all third-party logging classes. The results are shown in Figure 1(b). We calculate precision and recall at two levels: apk level and class level. At apk level, the precision of LogDetector is higher than that of the naive detector. At the class level, LogDetector has higher precision and recall than that of the naive detector. At any levels, LogDetector has higher recall and precision than the basic algorithm, because LogDetector leverages features to find app logging classes. Obviously, LogDetector has higher precision and recall by considering new features.

To evaluate the correctness as well as the performance of LogPruner, we further test LogPruner on an Android smartphone and measure its performance at launch time.

These apps run on an Android smart phone with a 1.2 GHz Qualcomm Snapdragon 410 CPU and 2 GB of RAM. We randomly operate each app for half an hour. Although the function of the apps can not revisit completely, our operation can cover most function of the apps. Results show that all the 20 optimized apps performed as before in functionality. Thus, we verify the correctness of our approach from both theoretical analysis and experiments.

Besides, we test the original version of these apps for comparison. CDF of their launch time is shown in Figure 1(c). After removing logging calls, most apps launch more quickly. The median of launch time of optimized apps is 409 ms, while it is 418 ms on original apps. On average, the launch time reduced 4% after optimization.

Conclusion. We complete LogPruner with a logging class detection algorithm, LogDetector. Based on the features we collect from 78 open source projects, LogDetector is designed to automatically detect logging classes. The detected logging classes are further analyzed and optimized by LogPruner to eliminate the logging hazards in Android apps. Experiments not only validate the correctness of LogPruner but also show it can outperform the original version on launch time by 4% across a range of popular off-the-shelf apps.

Acknowledgements This work was supported by National Key Research and Development Program (Grant No. 2016YFB1000105), and National Natural Science Foundation of China (Grant No. 61725201).

References

1 Marty R. Cloud application logging for forensics. In: Proceedings of the 2011 ACM Symposium on Applied Computing, TaiChung, 2011. 178–184

2 Yuan D, Zheng J, Park S, et al. Improving software diagnosability via log enhancement. In: Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, Newport Beach, 2011

3 Cai H Q, Zhou X, Lou S, et al. LogPruner: a tool for pruning logging call in android apps. In: Proceedings of the 9th Asia-Pacific Symposium on Internetware (Internetware'17), Shanghai, 2017

4 Hoglund G, McGraw G. Exploiting Software: How to Break Code. London: Pearson Higher Education, 2004

5 Wei F G, Roy S, Ou X M, et al. Amandroid: a precise and general inter-component data flow analysis framework for security vetting of android apps. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS'14), New York, 2014. 1329–1341