Demystifying Application Performance Management Libraries for Android

Yutian Tang[1], Xian Zhan[1], Hao Zhou[1], Xiapu Luo[1,*], Zhou Xu[2], Yajin Zhou[3] and Qiben Yan[4]

[1]Department of Computing, The Hong Kong Polytechnic University
[2]Department of Computer Science and Technology, Wuhan University
[3]Department of Computer Science and Technology, Zhejiang University
[4]Computer Science and Engineering , Michigan State University
[*]Corresponding Author: csxluo@comp.polyu.edu.hk

***Abstract*—Since the performance issues of apps can influence users' experience, developers leverage application performance management (APM) tools to locate the potential performance bottleneck of their apps. Unfortunately, most developers do not understand how APMs monitor their apps during the runtime and whether these APMs have any limitations. In this paper, we demystify APMs by inspecting 25 widely-used APMs that target on Android apps. We first report how these APMs implement 8 key functions as well as their limitations. Then, we conduct a large-scale empirical study on 500,000 Android apps from Google Play to explore the usage of APMs. This study has some interesting observations about existing APMs for Android, including 1) some APMs still use deprecated permissions and approaches so that they may not always work properly; 2) some app developers use APMs to collect users' privacy information.**

***Index Terms*—Application Performance Management Library, Android, Empirical Study**

I. Introduction

Android, a major mobile operating system, has shown its dominance in many mobile computing landscapes, such as tablets, smartphones, as well as vehicle systems. Since the performance issues of apps can influence users' experience, more and more developers adopt application performance management (APM) tools to monitor and detect potential problems in their apps [1], [2].

APMs have been used in many applications, such as cloud applications, web applications, and mobile apps. They can help developers monitor and detect potential performance issues in applications [3], [4]. Unfortunately, developers may lack deep understanding of the functionalities of APMs [1], [4], [5] as most of them are close-sourced. Moreover, APMs may be used intentionally or unintentionally to collect users' private information. For example, most APMs allow developers to log customized information in apps, which is called as leaving a breadcrumb. Demystifying the design of APMs and exploring the usage practices of APMs can benefit all stakeholders, including APM vendors, app developers, and app users. For APM vendors, the limitations we found in existing APM libraries can help them improve their products. For app developers, the implementation approaches of common functions as well as their limitations we reported can help them have a better understanding of APM libraries. For app users, the privacy leaks via APMs we discovered from apps can raise their awareness of this issue.

Motivation. Existing studies on APMs mainly explain how to use the data collected by APMs to diagnose or locate the problems in a program [3]–[6]. For instance, Ahmed et al. [3] discussed whether APMs can detect the performance regressions for web applications (e.g., excessive memory usage, high CPU utilization, and inefficient database queries) through an empirical study on three commercial APMs and one open-source APM. Trubiani et al. [5] proposed an approach to detect performance anti-patterns in load testing with Kieker APM. Heger et al. reported [4] the workflow of an APM. Unfortunately, they neither conduct a systematic analysis on the functionalities of APMs nor reveal the implementation details of APMs. As a result, developers may only have a vague idea about these APMs. To fill the gap, we demystify the major functionalities of 25 Android-oriented APMs and discuss their limitations.

Contribution. The contributions of this paper are as follows:

- To the best of our knowledge, this is the *first* systematic study on 25 popular APMs for Android. We demystify 8 major functionalities common in APMs and discuss their limitations.
- We conduct a large-scale empirical study on 500,000 Android apps from Google Play to explore how APMs are used in apps. We find that 23,397 apps can collect sensitive data from users through APMs;

We release our experimental dataset, results, and other supporting materials [7].

II. Demystifying APMs

In this section, we describe how the key features common in APMs are implemented as well as their limitations.

A. APMs under Examination

Criteria for Selecting Android APMs. We define the following criteria for selecting candidate APMs:

- The APM can be used to monitor Android apps;
- The APM must support key functions [8], including capturing crashes, network, diagnosis, capturing Android not respond (ANR) errors, performing Time-on-Page (ToP) analysis,

logging/tracking, viewing memory usage, CPU utilization and time consumed.

We manually inspect 53 APMs based on our selecting criteria and select 25 APMs that satisfy our conditions. The entire list of selected APMs can be found in [7].

B. Key Functions in APM

Capturing Crashes. When a crash occurs, an APM captures the uncaught `exception` and records the execution trace, which can be used by developers to learn how the exception is triggered. If an exception is not captured by any `try-catch-finally` blocks, it is reported as an uncaught exception, which causes the crash of an app. All selected APMs capture such crashes by registering an uncaught exception handler (using `Thread.setDefaultUncaughtExceptionHandler`), a customized instance of `Thread.UncaughtExceptionHandler`, to the main thread. When an uncaught exception occurs, the uncaught exception handler can capture the exception.

A limitation in this approach is that using setDefaultUncaughtExceptionHandler can update the `UncaughtExceptionHandler` with the Android framework. If an app uses two APMs, only the one last initialized can capture uncaughted exceptions as it overrides the default UncaughtExceptionHandler. As a result, only one APM can capture the crash.

Network Diagnosis. APMs can be used to diagnose the network bottleneck and monitor network performance. In general, there are two options: socket based solution and aspect oriented programming (AOP) [9] based solution.

Socket Connection Monitoring. APMs can track the network requests by monitoring the socket in use. Specifically, it can be realized by implementing the `SocketImplFactory` interface, and then setting the customized `SocketImplFactory` as the default. The information about the IP address and port of the target server can be obtained through reflection.

Using AOP for Interception

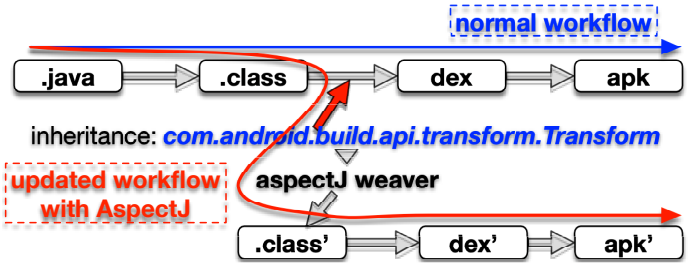

Fig. 1: Workflow of AspectJ in APM

```
@Pointcut("execution(* transfer(..))")// the pointcut expression
private void anyOldTransfer() {}// the pointcut signature
```

Fig. 2: The Structure of A Pointcut in AOP

Another method for monitoring and measuring URI requests is using AOP. For example, AspectJ is used to implement

```
1 @Pointcut("call(org.apache.http.HttpResponse
  org.apache.http.client.HttpClient.execute(org.apache.h
  ttp.HttpHost, org.apache.http.HttpRequest)) &&
  (target(httpClient) && (args(target, request) &&
  baseCondition()))")
2 void httpClientExecute(HttpUriRequest
request) {
3       …
4 }
```

Fig. 3: Http Requests Interception based on AOP

AOP in APMs [10]. The workflow of AspectJ in an APM is shown in Fig. 1. When a developer builds an app with `Gradle`, the APM can hook the transformation process from `classes` to a `dex` file by inheriting the `Transform` class [11]. Then, the AspectJ weaver [10] injects the customized code into the original `classes`.

AspectJ allows developers to use `Pointcut` to implement code injection dynamically. As shown in Fig. 2, a pointcut declaration contains two elements: a method signature comprising the name of the method and method parameters, and a pointcut expression determines the method executions to track. As the example shown in Fig. 3, the pointcut designator `call` is used to match all method executions whose method signatures are defined in the pointcut expression.

The pointcut designator `target` can intercept all join points (execution of methods). Consequently, with all these pointcut designators, APMs can capture network requests at runtime.

The limitation in the APMs with AOP-based interception is that it relies on the class `Transform`, which only supports the transformation from `classes` to `dex` file with Gradle build. Therefore, it can only be used in apps built with Gradle. If an app is not built with Gradle, AOP-based interception cannot work as expected.

Analyzing Android Not Response (ANR) Error. Application Not Responding (ANR) error is another typical error that frequently occurs in apps. When a user interface (UI) thread of an app is blocked for a long time, the ANR error will be triggered. APMs leverage the following approaches to capture the ANR errors:

Solution 1. APMs can implement a watchdog to detect ANRs. The watchdog is a `thread`, which can check the status of main `thread` in a periodic way. If the main `thread` has been frozen for more than a threshold, the watchdog will report an ANR error.

Solution 2. As it is known that Android is a message-driven system, system events are scheduled and appended to the message queue by the main thread. The main thread is also named as the `Looper` thread, which is responsible for looping message queue and handling messages in the message queue continuously. When the `Looper` is blocked (ANR error), Android outputs the ANR error into a certain trace file (`data/anr/traces.txt`). APMs can capture ANR errors by overriding the logger with the following API: `Looper.getMainLooper().setMessageLogging(Printer`

printer), because once the ANR occurs Android records the ANR error with default Printer and writes to the trace file. More precisely, APMs override the Printer to capture the ANR errors.

Compared with Solution 2, Solution 1 has two defects: (1) the watchdog thread has to keep checking the main thread to capture the ANR error; and (2) it is not easy to set the threshold. A small value can introduce performance overhead as it frequently checks the main thread. Not to mention that a large threshold can make the watchdog fail to report ANR timely.

Time-on-Page (ToP) Analysis. The time-on-page analysis aims at computing the time spent during UI display transitions. In Android, once Choreographer component receives an event (i.e., vertical synchronization) from the display system, it schedules the rendering work for the next frame [11]. The callback method doFrame is automatically invoked by Android when Android starts rendering the next frame. APMs apply Choreographer.FrameCallback.doFrame() to monitor UI display transitions. The time spent during UI display transitions can be recorded by APMs.

The limitation of this ToP analysis is that Choreographer is introduced since API 16. Therefore, it cannot be used for apps using old Android framework APIs.

Logging and Tracking. Developers can employ the logging functions provided by APMs to collect runtime information for debugging or understanding users' execution traces during the runtime. The information recorded with the built-in logging function in APM will be sent back to the server. Similarly, developers can also exploit APMs to track an event. Developers often use such a API to collect users' behaviors, such as preferences and execution paths.

Memory usage. Collecting memory usage is useful to diagnose the potential memory leakage in an app. In general, there are three approaches for collecting memory usage at runtime: (1) using the Android API ActivityManager.MemoryInfo; (2) reading the system file /proc/meminfo; and (3) invoking the Android API ActivityManager.getProcessMemoryInfo.

The method (1) and (3) can provide memory usage of the app. Whereas method (2) allows the inspection of the memory usage of all running processes.

CPU Utilization. To capture the CPU utilization, APMs obtain the CPU usage by inspecting system files. These system files include /proc/cpuinfo, /proc/<pid>/stat, /proc/stat, and /sys/devices/system/cpu/cpu0.

Time Consuming. To compute the time consumed by a code fragment, APMs mainly use two approaches: currentTimeMillis and TimeUnit.MILLSECOND. Both functions are defined in Java SDK.

We notice a compatibility defect in the existing APMs. That is, the file /proc/stat cannot be visited since Android 8.0 (API 26). APMs cannot collect CPU usage of all active processes with this approach.

III. Empirical Study

We guide our empirical study by answering the following research questions (**RQs**).

RQ1. How prevalent are APMs in Android apps?

Motivation. We setup this RQ to reveal the usage situation of APMs that are adopted by Android apps.

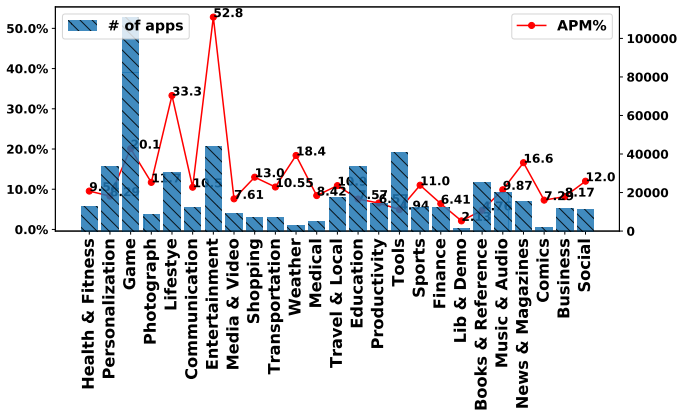

Fig. 4: APM Usage by App Category

Results. We randomly select 500,000 apps from Google Play. The size of apps ranges from 100KB to 1.2GB. These apps come from 25 categories. Note that we merge all subcategories of Game into one. We find that there are 55,722 apps (11%) that use APMs. The details of these apps' categories can be found in Fig. 4. From the diagram, we can see that developers usually use APMs in entertainment and lifestyle apps. By contrast, developers seldom choose APMs if apps belongs to Lib & Demo and Book & Reference.

RQ2. Will apps collect sensitive data using APMs?

Motivation. As APMs allow developers to log and collect values of variables at runtime, for this RQ, we aim at checking whether sensitive data will leak through APMs.

Methodology. We leverage FlowDroid [12] to detect the privacy leaks from sensitive data (a.k.a, source) to statements sending the data outside the application or device (a.k.a, sink). We use the sensitive APIs defined in existing works [12], [13] as source (e.g., getLatitude(), getSimSerialNumber()). We select the APMs' logging APIs as sinks. We manually verify that these APIs can send the data to APM servers. To be exact, for an APM API, we design a demo and log messages with the API. If messages logged by this API can be received by the APM server, we consider API as a sink. We consider a path from a source to a sink as a leak.

Results. As a result, we find 23,397 apps out of 55,722 apps (42%) collect sensitive data from users with APMs. In total, 99,566 leaks are found in all these 23,397 apps. The top-ranked sources for these leaks are: TelephonyManager::getDeviceId() (13943 leaks), android.location.LocationManager::getLastKnownLocation() (13906 leaks), org.apache.http.HttpEntity::getEntity (5030 leaks), and android.location.Location::getLatitude() (2852 leaks).

RQ3. What are the limitations in existing APM?

Motivation. In this RQ, we aim at discussing the limitations in APMs, which can assist APM vendors to improve their products.

Methodology. We care about two types of defects in APM: using deprecated permissions and accessing sensitive data. First, we manually collect permissions required by these APMs. Then, we reverse engineer all APMs to inspect insecure APIs.

Result. Some APMs request permissions that are proven to be deprecated or unnecessary. These permissions include READ_LOGS, READ_PHONE_STATE, GET_TASK, BLUE_TOOTH, SYSTEM_ALERT_WINDOW, and SYSTEM_OVERLAY_WINDOW. Specifically, the permission READ_LOGS and GET_TASK are deprecated.

IV. Related Work

Application Performance Monitoring. Trubiani et al. [5] discussed how to use the information collected by APM to diagnose the problem in applications. Ahmed et al. [3] studied the effectiveness of APMs for measuring the runtime performance of web applications. Yao et al. [1] discussed how to instrument logs in order to have better monitoring performance. Willnecker et al. [6] proposed an approach to model the performance of JavaEE applications. Different from these studies, we focus on exploring the functionality of APMs and discovering usage patterns of APM rather than discussing the way to use the data collected by APMs.

Network Measurement. Since the Android framework provides convenient interfaces for users to intercept and forward network packets, many apps are designed to measure the mobile network performance [14]. Li et al. [13] adopted the network round-trip time (nRTT) as the metric to appraise the accuracy of network measurement apps. Xue et al. [15] conduct a systematic study of three types of factors, including implementation patterns of measurement apps, Android architecture, and network protocols, to learn how these factors bias the measurement results of these apps.

Measurement and Monitoring for Apps. Several approaches have been proposed to diagnose performance bottlenecks in apps. AppInsight [16] instruments mobile apps by interposing event handlers to collect information on critical paths that are triggered by user transactions. Lee et al. [17] proposed a user interaction-based mobile application profiling system which can analyze fine-grained information, including user interaction, system behavior, and power consumption, to perform Android app tuning. AndroidPerf [18], a cross-layer profiling system, leverages cross-layer dynamic taint analysis as well as instrumentation to obtain both the execution information and the performance information about Android apps. DiagDroid [19] adopts a dynamic instrumentation approach to capture the data related to UI interactions and diagnose UI performance of apps.

V. Conclusion

Since developers still use APMs as black boxes, we conduct the firs study to demystify APMs for Android. We reveal the implementations of 8 key functions commons in APMs and their limitations. Moreover, we conduct a large-scale empirical study on 500,000 Android apps with interesting observations (e.g., limitations and issues in existing APMs).

VI. Acknowledgement

We thank the anonymous reviewers for their helpful comments. This work is supported in part by the Hong Kong General Research Fund (No. 152223/17E, 152239/18E), the Fundamental Research Funds for the Central Universities (No. NSFC 61872438), and the NSF grant (No. CNS-1566388).

References

[1] K. Yao, G. B. de Pádua, W. Shang, S. Sporea, A. Toma, and S. Sajedi, "Log4perf: Suggesting Logging Locations for Web-based Systems' Performance Monitoring," in *Proc. ICPE*, 2018, pp. 127–138.

[2] M. Karami, M. Elsabagh, P. Najafiborazjani, and A. Stavrou, "Behavioral Analysis of Android Applications Using Automated Instrumentation," in *Proc. SERE*, 2013, pp. 182–187.

[3] T. M. Ahmed, C.-P. Bezemer, T.-H. Chen, A. E. Hassan, and W. Shang, "Studying the effectiveness of application performance management (apm) tools for detecting performance regressions for web applications: An experience report," in *Proc. MSR*, 2016, pp. 1–12.

[4] C. Heger, A. van Hoorn, M. Mann, and D. Okanović, "Application Performance Management: State of the art and challenges for the Future," in *Proc. ICPE*, 2017, pp. 429–432.

[5] C. Trubiani, A. Bran, A. van Hoorn, A. Avritzer, and H. Knoche, "Exploiting Load Testing and Profiling for Performance Antipattern Detection," *Information and Software Technology*, vol. 95, pp. 329 – 345, 2018.

[6] F. Willnecker, A. Brunnert, W. Gottesheim, and H. Krcmar, "Using Dynatrace Monitoring Data for Generating Performance Models of Java EE Applications," in *Proc. ICPE*, 2015, pp. 103–104.

[7] OnlineArtifact, "Demystifying apm for android." [Online]. Available: https://sites.google.com/view/apm-empiricalstudy/

[8] Techbencon, "Performance engineering survey: Findings from 400 dev, test, and it ops professionals." [Online]. Available: https://techbeacon.com/app-dev-testing/performance-engineering-survey-findings-400-dev-test-it-ops-professionals

[9] V. O. Safonov, *Using aspect-oriented programming for trustworthy software development*. John Wiley & Sons, 2008, vol. 5.

[10] Eclipse, "Aspectj." [Online]. Available: https://www.eclipse.org/aspectj/doc/released/adk15notebook/index.html

[11] Google, "Android documentation." [Online]. Available: https://developer.android.com

[12] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *SIGPLAN Not.*, vol. 49, no. 6, pp. 259–269, 2014.

[13] L. Li, A. Bartel, T. F. BissyandÃl', J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting Inter-Component Privacy Leaks in Android Apps," in *Proc. ICSE*, vol. 1, 2015, pp. 280–291.

[14] D. Wu, R. K. C. Chang, W. Li, E. K. T. Cheng, and D. Gao, "Mopeye: Opportunistic monitoring of per-app mobile network performance," in *in Proc. USENIX ATC*, 2017, pp. 445–457.

[15] L. Xue, X. Ma, X. Luo, L. Yu, S. Wang, and T. Chen, "Is what you measure what you expect? factors affecting smartphone-based mobile network measurement," in *Proc. INFOCOM*, 2017, pp. 1–9.

[16] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh, "Appinsight: Mobile app performance monitoring in the wild," in *Proc. OSDI*, 2012, pp. 107–120.

[17] S. Lee, C. Yoon, and H. Cha, "User interaction-based profiling system for android application tuning," in *Proc. UbiComp*, 2014, pp. 289–299.

[18] L. Xue, C. Qian, and X. Luo, "Androidperf: A cross-layer profiling system for android applications," in *Proc. IWQoS*, 2015, pp. 115–124.

[19] Y. Kang, Y. Zhou, H. Xu, and M. R. Lyu, "Diagdroid: Android performance diagnosis via anatomizing asynchronous executions," in *Proc. FSE*, 2016, pp. 410–421.