The following publication Y. Tang et al., "A Systematical Study on Application Performance Management Libraries for Apps," in IEEE Transactions on Software Engineering, vol. 48, no. 8, pp. 3044-3065, 1 Aug. 2022 is available at https://doi.org/10.1109/TSE.2021.3077654

A Systematical Study on Application Performance Management Libraries for Apps

Yutian Tang, Haoyu Wang, Xian Zhan, Xiapu Luo, Yajin Zhou,
Hao Zhou, Qiben Yan, Yulei Sui, Jacky Keung

Abstract—Being able to automatically detect the performance issues in apps can significantly improve apps' quality as well as having a positive influence on user satisfaction. Application Performance Management (APM) libraries are used to locate the apps' performance bottleneck, monitor their behaviors at runtime, and identify potential security risks. Although app developers have been exploiting application performance management (APM) tools to capture these potential performance issues, most of them do not fully understand the internals of these APM tools and the effect on their apps. To fill this gap, in this paper, we conduct the first systematic study on APMs for apps by scrutinizing 25 widely-used APMs for Android apps and develop a framework named APMHunter for exploring the usage of APMs in Android apps. Using APMHunter, we conduct a large-scale empirical study on 500,000 Android apps to explore the usage patterns of APMs and discover the potential misuses of APMs. We obtain two major findings: 1) some APMs still employ deprecated permissions and approaches, which makes APMs fail to perform as expected; 2) inappropriate use of APMs can cause privacy leaks. Thus, our study suggests that both APM vendors and developers should design and use APMs scrupulously.

Index Terms—Empirical study, Android, Application performance management,

1 Introduction

Android is dominating the market for smartphone operating systems today. There are around 3 million apps in the Google Play store according to a report from AppBrain [1]. With high smartphone penetration, mobile apps have become indispensable to billions of users. The runtime performance of an app can significantly affect its user experience. Thus, more and more Android developers tend to employ application performance management (APM) tools to cope with performance bottlenecks [2], [3].

APMS can be used in desktop and mobile applications, such as cloud applications, web applications, and mobile apps. APMs assist developers in locating the potential performance limitations in applications [4], [5]. However, developers may not have profound understandings of the APMs' functionalities [2], [5], [6] as most of them are commercial (closed source) products.

Besides, inappropriate use of APMs can cause security issues for apps. For example, the regular logging frameworks (e.g., android.uti.log) allow developers to write custom logs. In most APMs, they also provide such functionality to developers. Since the information recorded in logs is determined by developers, developers may collect sensitive data at runtime. Demystifying the design of APMs and exploring the usage practices of APMs can benefit all stakeholders, including APM vendors, developers, and app users. To assist developers in understanding the functionalities of APMs, in this paper, we conduct a systematical study on APMs for Android apps. Besides providing insightful understandings of APMs, we reveal seven design defects in existing APMs (see Sec. 3) which can be used to improve existing APMs.

Motivation. Existing studies on APMs mainly explain how to use the data collected by APMs to diagnose or locate the problems (e.g., bugs) in a program [4], [5], [7]. For instance, Ahmed et al. [4] discussed whether APMs can detect the performance regressions (e.g., excessive memory usage, high CPU utilization, and inefficient database queries) for web applications. They conducted an empirical study on three commercial APMs and an open-source APM to examine whether these APMs can help developers diagnose the performance regressions. Trubiani et al. [6] leveraged the Kieker APM to detect performance anti-patterns in load testing. Streitz et al. [8] presented how SAP company employs APMs for performance prediction. Heger et al. reported [5] the activities and key concepts in an APM from the data perspective, such as data collection, data processing, data interpretation.

Existing research does not reveal the implementation details of APMs. As a result, developers use APMs as black-box tools. They only have a general but vague idea about these APMs instead of an insightful understanding. Apart from discussing the implementations of APMs, we also explore potential security issues, including whether there are defects in APMs, whether APMs actively collect data from users, whether APMs can be exploited by attackers. To fill this gap, we conduct a thorough study on 25 Android-oriented APMs and demystify their functionalities. Our research can enlighten developers about APMs with the points they usually ignore.

Major Extension. This paper is an extension of [9], which

Y. Tang is with ShanghaiTech University, China
H. Wang is with Beijing University of Posts and Telecommunications, China
X. Zhan, H. Zhou, and X. Luo are with the Hong Kong Polytechnic University, Hong Kong SAR, China.
Y. Zhou is with Zhejiang University and Engineering Laboratory of Mobile Security of Zhejiang Province, China
Q. Yan is with Michigan State University, USA
Y. Sui is with University of Technology Sydney, Australia
J. Keung is with the City University of Hong Kong, Hong Kong SAR, China
Y. Tang (csytang@ieee.org) and X. Luo (csxluo@comp.polyu.edu.hk) are the corresponding authors;

was published in the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019). The major extensions include:

• We provide additional background knowledge on two instrumentation approaches, which include manual instrumentation and auto-instrumentation. We compare them and present the differences and their suitability;

• We conduct additional analysis on commercial APMs (see Sec. 3). To be exact, we present the limitations and drawbacks of existing APMs (see Sec. 3.9). Meanwhile, we discuss whether existing APMs are qualified for monitoring apps' runtime performance (see Sec. 3.10);

• We define and propose a new framework named `APMHunter`, which can automatically detect the APM usages in the apps (see. Sec. 4). `APMHunter` contains three modules: *APM API identification* (i.e., for detecting the usage of APMs), *static analysis* (i.e., app analysis framework), and *usage identification* (i.e., locating and reasoning the usage of APMs in apps). It is worth mentioning that our method is obfuscation resilient so that it can handle both obfuscated and none-obfuscated apps. Furthermore, we also design a set of experiments to evaluate its accuracy (see Sec. 4.5);

• We conduct an additional empirical study on apps (see Sec. 5). Compared with the conference paper, we extend the two research questions (RQ1, RQ4), including (1) we rank APMs based on their popularity. We find that developers overwhelmingly choose commercial APMs rather than open-source APMs. We present 4 possible concerns for developers to use commercial APMs (see RQ1); and (2) comparing with the conference version, we extend our tool to support detecting privacy leaks from user inputs. We conduct additional experiments on detecting privacy leaks from user inputs to APMs (see RQ4). We also answer three new research questions (RQ2, RQ3, RQ5), including (1) as most APMs provide built-in logging functions for developers, we analyze the intention and consequences of using logging frameworks and APMs respectively (see RQ2); (2) we investigate the consequences of using multiple APMs in one app (see RQ3); and (3) we evaluate the performance overhead introduced by using an APM in an app. The corresponding results can assist developers in deciding whether to use APMs in their apps (see RQ5); and

• We add a new section (see Sec. 6) to provide guidance for stakeholders. For APM vendors, we offer four suggestions for them to develop APMs, which cover the vulnerabilities and limitations we found in the existing commercial APMs. For app developers, we present two suggestions for using APMs in their apps.

Contribution. In summary, our key contributions includes:

• To the best of our knowledge, this is the *first* work that conducts a comprehensive study to demystify the functionalities of Android-oriented APMs. We select 9 major functions in APMs and introduce the implementation details of these APMs, i.e., how these functions are implemented. We reveal 7 design defects in these APMs (see Sec. 3);

• We develop a prototype named APMHunter to automatically identify APMs used in an app, record APM usages, and report privacy leaks from the app to APMs. Moreover, APMHunter can process both obfuscated apps and non-obfuscated apps (see Sec. 4); and

• We conduct a large-scale empirical study on 500,000 Android apps fetched from Google Play to explore how APMs are used in apps, what the side-effects are introduced by APMs, and whether user privacy is leaked to APMs. We find that 23,397 apps will collect sensitive data from users through APMs (see Sec.5);

Skeleton. The rest of this paper is organized as follows: we present the background and usage of APMs in Sec.2, and elaborate the functionalities of these APMs in Sec. 3. To explore the usage of APMs in an app, we develop a tool named `APMHunter`, whose design and implementation are described in Sec. 4. Sec. 5 presents the details of our empirical study. Furthermore, we discuss the lessons learned and several keys issues of APMs in Sec. 6. After presenting the related work in Sec. 7, we conclude the paper in Sec. 8.

Data Availability. The experimental data and our tool APMHunter are available at: https://sites.google.com/view/systematical-apm-study.

2 Preliminary

This section introduces the motivation for using APMs in apps and how to embed an APM into an app.

2.1 The Need of APMs

Developers debug their apps locally before APMs become popular. They cannot know how apps work in different environments, as it can be hard for developers to test an app on all types of devices from all devices vendors (e.g., Blackberry, Google Pixel, Huawei, SAMSUNG). Moreover, through local debugging, developers cannot collect the runtime performance of their apps once the apps are released. To enable tracking runtime performance of apps, APMs play the key role. By integrating APMs into apps, developers can collect the runtime performance of apps even the apps have been released and distributed.

2.2 How to Use APMs in Apps?

To use an APM, a developer usually takes the following three steps:

• First, the developer registers an account for the APM. The account is used to login to the APM console to view the performance of the monitored app. Then, the developer needs to register an app with the APM vendor to obtain a unique ID for tracking the app;

• Second, the developer downloads the APM Standard Development Kit (SDK) and integrates it into the apps.

• Last, the developer publishes his app via app stores (e.g., Google Play, Amazon) and collects runtime performance of the app through the APM. The data is then displayed on the APM's console.

2.3 Manual Integration vs. Automatic Integration

Existing APMs support two ways of integration: manual integration and automatic instrumentation. The differences between the manual integration and automatic integration are listed as follows (also see Fig. 1).

• *Manual integration* requires developers to manually integrate APM SDKs into apps. As illustrated in Fig. 1, developers add an APM SDK as a dependency in the source

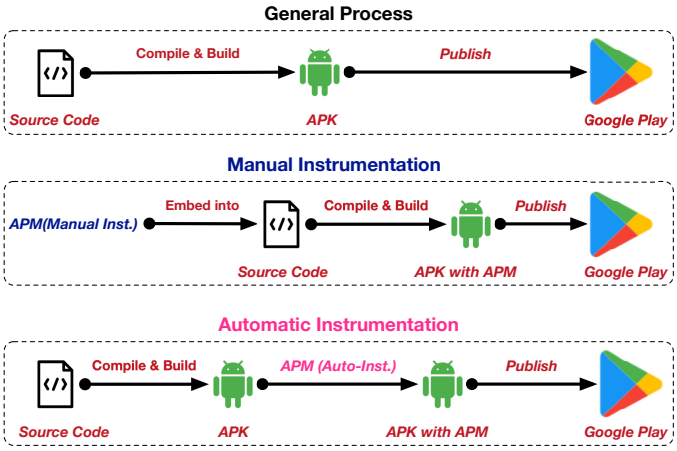


Fig. 1: Workflow for Integrating APMs

code. Once the source code is built, the APM is embedded in the APK (bytecode of the app). The procedure is the same as using a third-party library in an app. Most APMs (e.g., UMeng [10], Flurry [11]) adopt this strategy.

• *Automatic integration* works on app binaries rather than source code. APM vendors provide scripts for automatically injecting APMs (e.g., CA Mobile [12]) into app binaries (APKs). The manual integration is suitable for apps whose source code is available, whereas the automatic integration usually aims at binaries.

There is no evidence on which approach is better. But it is worth mentioning that attackers can repackage victim apps [13] by leveraging APMs which support automatic integration. For example, it can be hard for attackers to obtain the source code of the Facebook app but the app is available on Google Play. Thus, the attackers can instrument the Facebook app through APMs that support automatic integration, and then publish the repackaged app on some third-party app markets to collect users' data. We compare the usage of these two strategies and find that developers overwhelmingly choose APMs with manual integration. The corresponding data and detailed explanation are presented in Sec. 5.1.

3 Application Performance Management

3.1 Selection Criteria

Since Android has a prominent position in the entire mobile eco-system, in this paper, we focus on APMs targeting on Android apps. Since previous studies on APMs do not offer a list of Android-oriented APMs [2], [4], [6], [7], we select the target APMs through the following steps: (1) As APMs are considered as third-party libraries, we first crawl the candidate third-party libraries belonging to the category of development tools from AppBrain. Specifically, we crawl the metadata of these libraries, which include descriptions, the market share, tags (e.g., crash reporting, open source), and the official site [14]; (2) To select the target APMs from these candidates, we carefully define the following criteria by referencing [15]:

• The APM must be able to monitor Android apps;

• The APM can support at least 3 common functionalities. The common functionalities include capturing crashes, correlating server performance, logging, tracking user behaviors, and capturing ANR.

As a result, we obtain 25 APMs that meet the criteria as shown in Table 1, including 20 commercial APMs and 5 open-source APMs. The statistics from AppBrain [14] show that the selected APMs hold over a 90% market share [14] of the apps that use APMs. Note that, unlike other APMs, Google Firebase is a collection of libraries, including ad analysis, A/B testing, remote configuration, cloud messaging, and performance diagnosis. In this paper, we only focus on Firebase Analytics, because it allows developers to track users' behavior and capture crashes at runtime. Table 1 lists the functions supported by these APMs and their integration methods. In the coming sections, we introduce how these functions are implemented by reverse-engineering all these APMs.

3.2 Capturing Crash in Java Code

When a crash happens in Java code, an APM captures the uncaught `exception` and records the execution trace for the crash, which provides more information about how the exception is triggered. If an exception is not captured by any `try-catch-finally` block, it is then treated as an uncaught exception, which causes the app to crash. The APMs that can capture such crashes follow the same procedure: they register an uncaught exception handler (using `Thread.setDefaultUncaughtExceptionHandler`), a customized instance of `Thread.UncaughtExceptionHandler`, to the current thread. When an uncaught exception occurs, the uncaught exception handler implemented by APMs captures the exception.

> The main limitation is that using setDefaultUncaughtExceptionHandler can update the `UncaughtExceptionHandler` for the Android framework. If an app uses two APMs, only the last initialized APM can capture uncaught exceptions, because only one `UncaughtExceptionHandler` can be defined as the default handler (more details are introduced in §5).

3.3 Capturing Crash in Native Code

Crashes can also happen in native code (C/C++) of an Android app. APMs handle native crashes with the following steps: installing a signal handler, extracting the stack traces, and building the symbol files.

• Installing a signal handler. When crashes occur in native code, an error signal is generated [38], [39]. APMs can capture this error signal by using the method `sigaction` to register a signal handler.

• Extracting the stack traces. After receiving the signals, the APM copies the crashed process to a daemon process, which shares an address space with the crashed process. This allows the APM to trace the crashed process.

• Building the symbol files. APMs locate the start point of the crash to recover the crash point from the program. After that, APMs generate human-readable stack traces and symbol files. The symbol files hold a variety of data, such as

TABLE 1: APM Libraries Studied

Lib	Monitoring								Free−Pay	Integration
	Crash(Java-Native)	Network	ANR	CPU	Mem.	ToP	Log	Event Track.		
Tingyun [16]	✓− ✗	✓	✓	✓	✓	✓	✓	✓	✓−✓	source code
BaiduAPM [17]	✓−✓	✓	✓	✓	✓	✗	✓	✓	✓−✓	source code
UMeng [10]	✓−✓	✗	✗	✗	✗	✗	✓	✓	✓−✓	source code
Mobile Tencent [18]	✓−✓	✓	✗	✓	✓	✗	✓	✗	✓−✗	source code
OpenInstall [19]	✗−✗	✓	✗	✗	✗	✗	✓	✓	✗−✓	source code
New Relic [20]	✓−✗	✓	✗	✓	✓	✗	✓	✓	✗−✓	source code
App Dynamics [21]	✓− ✗	✓	✗	✗	✗	✗	✓	✓	✗−✓	source code
OneAPM [22]	✓− ✓	✓	✓	✓	✓	✗	✗	✗	✗−✓	source code
GrowingIO [23]	✗−✗	✓	✗	✗	✗	✓	✗	✓	✗−✓	source code
Google Firebase [24]	✓−✓	✓	✗	✗	✗	✗	✓	✓	✓−✓	source code
Dynatrace [25]	✓− ✗	✓	✗	✓	✓	✗	✓	✓	✗−✓	source code
Site24×7 [26]	✗− ✗	✓	✗	✓	✓	✗	✗	✓	✗−✓	source code
AppPulse Mobile [27]	✓− ✗	✓	✗	✗	✓	✗	✗	✓	✗−✓	bytecode
CA Mobile [12]	✓− ✗	✓	✓	✓	✓	✗	✗	✓	✗−✓	bytecode
Apteligent [28]	✓− ✓	✓	✗	✓	✓	✗	✓	✓	✗−✓	source code
Flurry [11]	✓−✓	✗	✗	✗	✗	✗	✓	✓	✗−✓	source code
AppsFlyer [29]	✗− ✗	✗	✗	✗	✗	✗	✓	✓	✗−✓	source code
Yandex Metrica [30]	✓− ✓	✓	✗	✗	✗	✗	✓	✓	✗− ✓	source code
Adjust [31]	✓− ✗	✗	✗	✗	✗	✗	✓	✓	✗− ✓	source code
Ironsource [32]	✓−✗	✓	✗	✗	✗	✗	✓	✓	✗− ✓	source code
Countly [33]	✓− ✗	✓	✗	✗	✗	✗	✓	✓	open source	source code
Sentry [34]	✓− ✗	✗	✓	✓	✓	✓	✓	✓	open source	source code
AndroidGodEye [35]	✓− ✗	✓	✓	✓	✓	✗	✓	✗	open source	source code
BlackCancary [36]	✗− ✗	✗	✓	✓	✓	✓	✗	✗	open source	source code
ArgusAPM [37]	✓− ✗	✓	✓	✓	✓	✓	✓	✓	open source	source code

Network: network diagnosis; ANR: Android Not Responding; CPU: CPU utilization; Mem.: Memory usage; ToP: Time on page; Log: customized logging.

function, module, call frame information, which helps with the debugging process.

Interested readers are referred to Google's Breakpad [40] framework for details. Breakpad is a mature and open-source library for debugging and analyzing crashes for C/C++ program. It is used in most commercial APMs (e.g., UMeng, Tingyun, Sentry).

3.4 Network Diagnosis

APMs can also be leveraged for diagnosing the network bottleneck and monitoring network performance. In general, there are two widely adopted solutions: socket based solution and aspect-oriented programming (AOP) [41] 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 Java Reflection [42].

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

Fig. 2: The Structure of A Pointcut in AOP

Using AOP for Interception Another strategy for monitoring and measuring URI requests sent by apps is using AOP. In APMs, the AspectJ is used for implementing AOP [43]. The workflow of AspectJ in an APM is shown in Fig. 3. When a developer builds an app with `gradle` [44], the APM can hook the transformation process from `classes` to a `dex` file by inheriting the `Transform` class [45]. Then, the AspectJ weaver [43] injects the customized code into the original `classes` and then builds the customized `dex` file.

AspectJ allows developers to leverage `Pointcut` to implement code injection for runtime monitoring. As shown in Fig. 2, a pointcut contains two elements: a signature that comprises a name and parameters, and a pointcut expression that determines the method executions to track. Fig. 4 shows an example, where 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 limit matching to join points (execution of methods). Consequently, with all these pointcut designators, APMs can capture network requests at runtime.

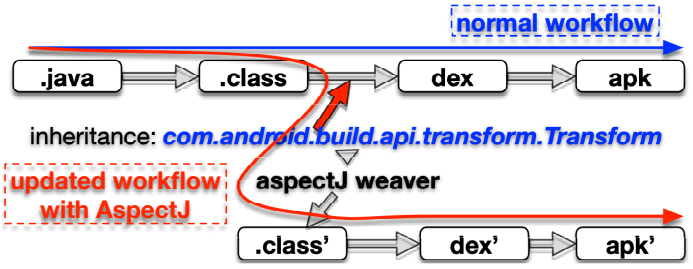


Fig. 3: Workflow of AspectJ in APM

```
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. 4: Http Requests Interception based on AOP

The AOP-based approach relies on `Transform`, which only supports the transformation from `class` to `dex` file with Gradle build. It means that the AOP-based approach is only suitable for apps built with Gradle [44].

3.5 Analyzing Android Not Response (ANR) Error

Application Not Responding (ANR) error is another typical performance issue in apps. When a user interface (UI) thread of an app is blocked for a long time, the ANR error is triggered. APMs use the following approaches to track ANR errors:

Solution 1. APMs can implement a watchdog to detect ANR errors. The watchdog itself is a `thread`. It checks the status of main `thread` in a periodic way. If the main `thread` is frozen for over `n` seconds (`n` is a predefined by APM vendors), the watchdog reports the ANR error.

Solution 2. As Android is a message-driven system, system events are scheduled and appended to the message queue. The main thread (a.k.a `Looper` thread) is responsible for looping the 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`). By rewriting the `Looper.getMainLooper().setMessageLogging(Printer printer)` API, APMs can capture the ANR. This is because once the ANR occurs, Android records the ANR error using the default `Printer` and writes to the trace file (`data/anr/traces.txt`). As APMs override the `Printer`, APMs can capture the ANR error.

Compared with Solution 2, Solution 1 has two limitations: (1) the watchdog thread has to keep checking the main thread to capture the ANR error; and (2) it is difficult to set a proper timeout value for the watchdog thread. A small timeout value can cause performance overhead as it frequently checks the main thread. However, a large timeout can make the watchdog fail to report ANR errors promptly.

3.6 Time-on-page Analysis

The time-on-page (ToP) analysis aims at monitoring the time spent on a page (e.g., an `Activity`). To compute the time-on-page, APM must be able to capture UI display transitions. When a UI display changes, a new page is loaded. It suggests a display transition.

APMs apply `Choreographer.FrameCallback.doFrame()` to monitor UI display transitions. In Android, the Choreographer component receives timing pulses from the display, and then it schedules the rendering work for the next display frame [45]. The callback method `doFrame` is automatically invoked by Android when Android starts rendering the next display frame.

The limitation of the ToP analysis is that the Choreographer API is introduced since Android 4.1 (API 16). Thus, it cannot be used for devices with an API level lower than 16.

3.7 Logging and Tracking

Developers can employ the logging functions provided by APMs to collect users' execution traces during the runtime. The information recorded with the built-in logging function in APM is sent back to the server. Developers can also leverage APMs to track any concerned event. For example, in New Relic APM, developers can use `recordCustomEvent(type,name,attributes)` to record an event at runtime. In practice, developers can use such APIs to collect user behaviors, such as preferences and execution paths.

3.8 Other supporting functions

Memory usage. Memory usage data can be used to diagnose potential memory leaks in the app. In general, there are three approaches for collecting memory usage data: (1) using the Android API `ActivityManager.MemoryInfo`; (2) accessing the system file `/proc/meminfo`; and (3) utilizing the Android API `ActivityManager.getProcessMemoryInfo`.

The method (1) and (3) can provide memory usage information of the target app. The method (2) provides the memory usage of all processes running in Android. Then, the memory usage information for the target app is then filtered by APMs.

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`. Since Android 8.0 (API 26), the file `/proc/stat` cannot be visited without the root permission.

Time Consuming. To compute the time consumed by a code fragment, APMs mainly take advantage of two APIs: `currentTimeMillis` and `TimeUnit.MILLSECOND`. Both are defined in Java SDK.

There is a compatibility defect in the existing APMs that 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. However, developers can still collect the CPU usage from other system files (e.g, /proc/cpuinfo), which leads to the leak of private data (e.g., CPU family, CPU model) through APM (also reference [46]).

3.9 Limitations and Drawbacks of APMs

Requesting unnecessary or dangerous permissions. Some APMs request permissions that are proven to be deprecated or unnecessary. These permissions include READ_LOGS, READ_PHONE_STATE, GET_TASK, BLUE_TOOTH, SYSETM_ALERT_WINDOW, and SYSETM_OVERLAY_WINDOW. Specifically, the permission READ_LOGS and GET_TASK are deprecated. Some permissions should not be granted, such as device state and Bluetooth state. This information usually should not be leaked to developers.A number of research studies point out that some attacks are strongly related to these permissions [47]–[50].

Accessing sensitive data and files. Some APMs collect information from `logcat`. Even though the permission is deprecated since Android 4.1, it still works for legacy Android system versions prior to Android 4.0. In addition, the `logcat` contains the information from all running processes. Therefore, data from other processes is exposed to APMs and app developers, causing the leakage of users' privacy information.

TABLE 2: APMs' Capability on Handling Performance Anti-patterns

Anti-pattern	APMs Support	Anti-pattern	APMs Support
(1) GUI lagging	1,2,8,14,22,23,24,25	(2) Energy leak	Nil
(3) Memory bloat	1,2,4,6,8,11,12,13,14,15,22,23,24,25	(4) Cyclic/Frequent invo.	1,2,4,6,8,11,12,14,15,22,23,24,25
(5) Expensive callee	1,2,4,6,8,11,12,14,15,22,23,24,25	(6) Loading time	1,2,4-15,18,20,21,23,25
(7) Query local DB	Nil	(8) UI overdraw [51]	Nil

1:Tingyun; 2:BaiduAPM; 3:UMeng; 4: Mobile Tencent; 5: OpenInstall; 6: New Relic; 7: App Dynamics; 8: OneAPM; 9 GrowingIO; 10: Google Firebase; 11: Dynatrace; 12: Site24 × 7; 13: AppPulse; 14: CA Mobile; 15: Apteligent; 16: Flurry; 17: AppsFlyer; 18 Yandex Metrica; 19 Adjust; 20: Ironsource; 21 Countly; 22: Sentry; 23: AndroidGodEye; 24: BlackCancary; 25: ArgusAPM

We do not recommend APM vendors to access the `/proc/stat` file as it cannot be visited without the root permission since Android 8.0. APM vendors have other options (e.g., visit the file `/proc/cpuinfo`) for accessing the CPU information as presented in Sec. 3.8. Although these approaches are not officially blocked by Android, using them can lead to privacy leaks. [46]. For example, by accessing the file `/proc/cpuinfo`, we can obtain the device's CPU information, such as CPU family, CPU vendor id, and CPU model.

3.10 Address Performance Anti-patterns in Android Apps.

Next, we explore whether APMs can assist developers in debugging and locating anti-patterns in apps. Anti-patterns are considered as bad programming practices in a program [52]. Inspecting whether APMs can resolve these anti-patterns helps APM vendors find room to improve their APMs.

We carefully select 8 performance anti-patterns of mobile apps from several empirical studies [51], [53]–[55]. These anti-patterns are defined and confirmed by app developers and can be considered as the key issues concerned by developers in terms of app performance. We manually evaluate whether existing APMs can address them.

The list of anti-patterns is presented in Tab. 2. Specifically, (1) *GUI lagging* prevents user events from being handled in a timely way. This also triggers ANR errors; (2) *energy leak* represents unexpected excessive consumption of battery power of an app; (3) *memory bloat* refers to the bug that can incur unnecessarily high memory consumption; (4) *cyclic invocation* represents a frequently executed method in an invocation cycle; *frequent invocation* refers to a method being frequently executed; (5) *expensive callee*: a method is slow in executing its callees' code; (6) *loading time* is the time for loading a resource or a UI display; (7) *query a DB* stands for computing time spent for querying an item from a local database; and (8) *UI overdraw* [51] represents catching the case that an app draws the same pixel more than once within a single frame.

We follow the following steps to determine whether an APM can detect a performance antipattern.

• STEP 1 (Learn from APM documentation): We first learn the performance anti-patterns from existing studies [2], [4], [6], [7] to understand their consequences. Then, we check whether the APM documentations clearly claim that they can detect these anti-patterns or they can detect the consequences caused by these performance anti-patterns. For example, the "GUI lagging" anti-pattern can result in ANR errors. Thus, APMs supporting ANR errors detection can detect this anti-pattern.

However, only using STEP 1 may introduce biases because sometimes an APM's documentation may not be consistent with its implementation. We perform additional checks in STEP 2 and 3.

• STEP 2 (Build samples): We build a sample app, which does not contain any performance anti-pattern. We term it "*base* app". For each anti-pattern, we implement and add a module containing the anti-pattern to the *base* app. We name the new app as $anti_i$, where i is the index of the anti-pattern (see Table 2). For example, anti_1 represents the *base* app with the GUI lagging;

• STEP 3 (Verification): To evaluate whether an APM can detect a performance anti-pattern i, we embed the APM to the *base* app and the $anti_i$ app respectively. Then, we analyze whether the data collected by APMs suggests a performance anti-pattern. For example, if an APM finds that the $anti_3$ app consumes more memory than the *base* app, it suggests a "memory bloat" anti-pattern.

The results are summarized in Table 2. We can see that several common performance anti-patterns cannot be handled, including energy leak, time-consuming for database query, and UI overdraw. For "energy leak", the APMs under investigation cannot detect it as they don't collect battery usage data. Thus, we cannot rely on APMs to detect the energy leak antipattern. For "query a local DB", the existing APMs do not offer any DB related functions, such as the time needed for querying a database. Same for "UI overflow", if a pixel is drawn more than once, the state-of-art APMs do not provide any functions for recording it.

> Even though some anti-patterns can be well resolved by using APMs, there are still several common performance anti-patterns that cannot be addressed, including energy leak, time-consuming for database query, and UI overdraw. Therefore, there is room for APM vendors to improve their products.

3.11 Evolution of APMs

Android itself frequently introduces new features and updates the existing system during its lifetime. With the updates of the Android system, new features are introduced and some APIs become deprecated. Therefore, it is worth investigating: (1) How do APMs evolve during their lifecycles? and (2) Do APMs evolve to respond to the changes on Android?

We collect all the available versions for the 25 APMs we studied. As APM SDKs are presented as `.jar` files,

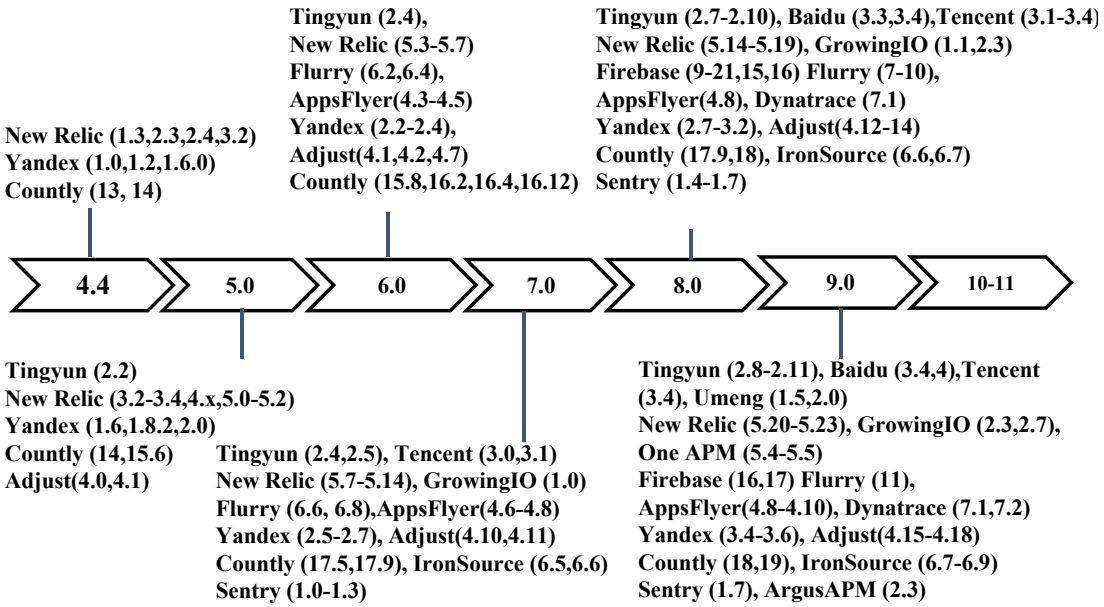


Fig. 5: APM Evolution Timeline (map to Android version)

we leverage `PKGDiff` [56] to locate changes cross different versions. Then, we manually analyze the changes in the successive two versions and summarize them. To reason the changes and the intentions behind the changes, we analyze the changelogs and documentation of these APMs to ensure correctness.

By inspecting all the 25 APMs, we find there are 18 APMs[1] whose historical versions are publicly available. Fig. 5 shows the mapping relations between APM versions and Android versions. We can see that a single version of APM can spread across different Android versions.

After manually inspecting the changes upon APMs during their lifecycles, we learn that the changes fall into the following cases:

1. Fixing the compatibility issues: There are two sub-patterns for such compatibility-related changes:

Pattern 1. Fixing the compatibility issues with Android systems. When Android updates, the APMs may not be compatible with the latest version, which results in crashes at runtime. Thus, vendors update their APMs to fix such issues.

Pattern 2. Fixing the compatibility issues with other third-party libraries. Similar to other apps [57], APMs may use some third-party libraries. If they are not compatible with these libraries, APM vendors update their APMs to be compatible with the major third-party libraries.

2. Additional features for additional application scenarios. APM vendors update their products by supporting more scenarios. For example, APMs (e.g., Adjust, Flurry) add additional tracking APIs for developers to monitor the in-app purchase action.

3. Kotlin support and Native code support. More and more apps are developed in the Kotlin language rather than Java. Thus, the APMs are evolved to support Kotlin-based apps.

1. Other 7 APMs are OpenInstall, App Dynamcis, Site 24*7, AppPulse, CA, and Apteligent

4. Code optimization and bug fixing: Another common practice is code optimization and bug fixing. As APMs are embedded into apps, APM vendors always attempt to limit the sizes of APMs and the resources required by APMs.

Besides, we also investigate whether the changes introduced by a new Android version break the functionalities of APMs and how APMs respond to these changes.

- Version $4.4 \rightarrow 5.0$: Since Android 5.0, the permission `GET_TASKS` becomes deprecated. However, some APMs still request such permission from users.
- Version $7.0 \rightarrow 8.0$: Since Android 8.0, the file `/proc/stat` cannot be accessed without the root permission. However, APMs relying on this file to collect CPU usage do not make any adjustments.
- Version $9.0 \rightarrow 10-11$: Since Android 10.0, the sample rate allowed by API `getProcessMemoryInfo` is significantly limited. If the API is called faster than the limit, the same data as the previous call is returned. It suggests that APMs should adjust the invocation rate of the `getProcessMemoryInfo` API if used to collect memory usage. However, no APM makes the corresponding adjustments.

In summary, some version updates break APMs' functionalities, but APMs fail to make the corresponding changes to cope with such updates. As a result, these updates make some functions in APMs fail to work properly.

> By inspecting the evolution process of APMs, we summarize 5 key intuitions for APM vendors to update their APMs, including fixing comparability issues, hunting bugs, supporting Kotline, native code, supporting HTTP/2, optimizing code, and involving new features. Since some OS updates can break APMs' functionalities, APM vendors should carefully cope with the OS updates.

4 METHODOLOGY

To understand how APMs are used in real apps, we develop `APMHunter`, an automated tool, to detect the us-

ages of APMs in apps. The overview of APMHunter is shown in Fig. 6. APMHunter contains three major components: `APM API Identification`, `Static Analysis`, and `Usage Identification`.

• The `APM API identification` module aims at detecting whether any APM is used in an app;
• The `static analysis` module is a general framework to analyze a given app that uses APMs; and
• The `usage identification` module records the usage patterns of APMs and checks misuse of APMs (i.e., collect sensitive data with APMs).

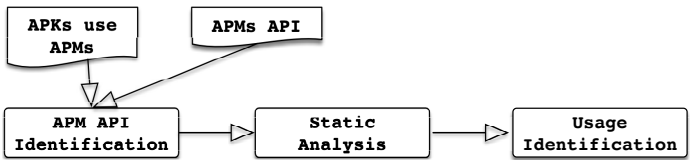


Fig. 6: The Overview of APMHunter

4.1 APM API Identification

Motivation. This component aims at identifying APMs used in apps, which helps us spot apps that leverage APMs. Since code obfuscation techniques transform code into an obscure format [58], [59], they are widely used by commercial apps. To analyze obfuscated apps, we design the following obfuscation-resilient approach. If an app is packed, we will first use existing tools [60]–[62] to unpack it.

Intuition. As reported in [63], [64], there are usually at least two invariants that are preserved during obfuscations.

• The first invariant is the class inheritance relation. Although the names of classes, methods, parameters, and variables can be obfuscated, class inheritance relations remain stable even with obfuscation. For example, if class A is a superclass of class B, after obfuscation, A is still the superclass of B; and

• The second invariant is the caller-callee relation. Inside a program, the invocation from a caller to its callee does not change. For instance, a method `downloadVideo` calls two methods `Util.isNetworkAvailable` and `StringBuilder.append`, the caller-callee relations from `downloadVideo` to `Util.isNetworkAvailable` and `StringBuilder.append` donot change during obfuscation.

Methodology. We perform the following steps to identify APM usages in apps:

• **Step 1.** We first build the signatures of all methods in all 25 APMs. It results in a list of signatures for the APIs in these APMs. The way to generate the signature for a given method is presented in Alg. 1;
• **Step 2.** For a given app, we build the signatures of all methods in the app;
• **Step 3.** Last, we leverage the signature lists of APMs (results in **Step 1**) as the search queries for detecting the usage of APMs in an app.

Our signature construction for a single method is presented in Alg. 1. As mentioned in **Step 1** and **Step 2**, Alg. 1 is used to generate signatures for methods in APMs and apps. Note that, the terms "class" and "type" are used interchangeably as a type in Java is represented by a class.

ALGORITHM 1: GETFUNCSIG

Input : f_0– target function; $\mathcal{C}_s$ – system classes; $\mathcal{F}_s$ – system functions;

```
1  fSigBuf ← ∅ // fSigBuf is a temporary String buffer
2  ℒ ← ∅ // ℒ is a temporary variable
3  t_0 ← GETHOSTCLASSTYPE (f_0)
4  fSigBuf ← fSigBuf ∪ TYPEENCODING (t_0, t_0, C_s)
   /* Get and encoding parameter                          */
5  for t_p ← GETPARAMETERTYPE (f_0) do
6  |  fSigBuf ← fSigBuf ∪ TYPEENCODING (t_0, t_p, C_s)
7  end
   /* Get and encoding return                             */
8  t_r ← GETRETURNTYPE (f_0)
9  fSigBuf ← fSigBuf ∪ TYPEENCODING (t_0, t_r, C_s)
   /* Get and encoding all callees in the method          */
10 for each callee f_i ∈ GETCALLEE (f_0) do
11 |  if f_i ∈ F_s then
12 |  |  ℒ ← ℒ∪ name (f_i, argType(f_i), retType(f_i))
13 |  end
14 |  else
15 |  |  ℒ ← ℒ∪ GETFUNCSIG (f_i, C_s, F_s)
16 |  end
17 end
18 fSigBuf ← fSigBuf ∪ SORTNEXCLUDERE(ℒ)
      // SORTNEXCLUDERE: sort and remove redundant signatures
19 return fSigBuf
```

We use "class" to represent both classes and interfaces rather than distinguishing them.

• **Input**: Our algorithm requires three inputs: f represents the method for generating signature; $\mathcal{C}_s$ gives the all system classes; and $\mathcal{F}_s$ denotes all system functions;
• **Line 1-4**: We first get the type (t_0) of f_0's host class (i.e., GETHOSTCLASSTYPE in Line 3). Next, we invoke the TYPEENCODING method (see Alg.2) to encode the type t_0;
• **Line 5-7**: Then, we encode the parameters that declared in the method f_0 with the TYPEENCODING method;
• **Line 8-9**: We embed the `return` information into f_0's signature;
• **Line 10-17**: If method f_0 invokes any methods, we iteratively collect the signatures of these callees. Then, we add these signatures to the temporary variable $\mathcal{L}$. However, if a callee is a system function (i.e., Android system API), we directly add the method signature to the temporary variable $\mathcal{L}$; and
• **Line 18-19**: Then, we sort and remove redundant signatures in $\mathcal{L}$ and append the results to fSigBuf. We sort and remove redundant signatures in $\mathcal{L}$ as: (1) we only consider the method invoked in f_0 regardless of their order; and (2) if a method is invoked multiple times, we only count once. As our task in this module is to identify certain methods (i.e. APM APIs) that are invoked in apps, we only need to consider each signature once.

The function TYPEENCODING (Alg. 2) is designed for encoding any type defined in the app.

• **Input**: The TYPEENCODING's input consists of the host class c_h, which represents the current class context of the c_t (i.e., we currently encode the class c_t which is used in class c_h), target class c_t, which represents the type under encoding; and $\mathcal{C}_s$, which contains all classes defined in the app;
• **Line 1-5**: If the type (i.e. c_t) to be encoded is a system type (i.e. Android system API), we return the name of the type;
• **Line 6-13**: If the type (i.e. c_t) to be encoded is defined in the app, we encode its superclass c_p (if exists), and append

ALGORITHM 2: TYPEENCODING

Input : c_h– class; c_t – target class; $\mathcal{C}_s$ – classes defined in app;

```
1  tBuf ← ∅ // tBuf is a temporary String buffer
2  ℒ ← ∅ // ℒ is a temporary String buffer
3  if c_t ∈ C_s then
4  |  tBuf ← tBuf ∪ name(c_t)
5  end
6  else
7  |  c_p ← GETSUPERCLASS (c_t)
8  |  tBuf ← tBuf ∪ TYPEENCODING (c_h, c_p, C_s)
9  |  for c_i ← GETINTERFACES (c_t) do
10 |  |  ℒ ← ℒ∪ TYPEENCODING (c_h, c_i, C_s)
11 |  end
12 |  tBuf ← tBuf ∪ SORT(ℒ)
13 end
14 return tBuf
```

the encoding of c_p to the temporary result tBuf. If c_t inherits any interface, we also encode all interfaces, and append the encoding results to tBuf. We sort and remove redundant signatures in $\mathcal{L}$ as we only consider the interfaces inherited by c_t regardless of their order; and

• **Line 14**: The temporary variable tBuf is returned as the coding of the type c_t.

When building method signatures with Alg. 1, we replace non-system identifiers/names with symbol 'x' in signatures to reduce the side-effects introduced by obfuscation. For example, class name `com.networkbench` is replaced by `X.X`.

After encoding the methods in an app and an APM with Alg. 1, we search the initialization method of an APM inside the app. For example, the UMeng APM can be initialized by invoking the `UMConfigure.init()` API. If method m in app invokes the `UMConfigure.init()`, method `UMConfigure.init()`'s signature must appear in m's signature as `UMConfigure.init()` is a callee of m (see Alg. 1 (Line 10-17)). As a result, we know that the APM is used in the app.

4.2 Static Analysis

Motivation. After confirming an app contains an APM, the next step is to characterize the APM usages in the app. We intend to collect the following information: (1) the position where APMs are initialized; (2) the leak of users' private data thought the APM; and (3) the APM APIs used in the app (See §4.3). To achieve this goal, we conduct a static analysis on the app to build its inter-procedural control-flow graph (ICFG).

Methodology. To build the ICFG and conduct the data flow analysis on an app, we perform the following steps: (1) locating entry points for the app; (2) performing static analysis; and (3) exploring inter-component communications in the app.

4.2.1 Entry Points

To build the ICFG for an app, we first resolve its entry points. Unlike desktop Java applications, Android apps do not have explicit entry points (e.g., `main` method). The entry points of an app are derived from two aspects [65]: (1) lifecycle methods in Android components (i.e., `Activity`); and (2) user interface (UI) events handler callbacks. Specifically, we leverage the state-of-art tool EdgeMiner [66] to explore these two types of entry points.

4.2.2 Build the ICFG and Perform Data-flow Analysis

The next step is to build the ICFG for an given app. The ICFG contains control flow information of the given app. Besides, we perform the data flow analysis on the app, and append the data flows to the ICFG. This process is implemented based on the FlowDroid [67], which is a widely used Android app analysis framework based on Soot (Java optimization framework) [68].

4.2.3 Inter-component Communication (ICC)

Inside an app, data can be sent cross different components through `Intents`. Since `Intents` in an app introduce additional data flows to the ICFG, we append data flows introduced by `Intents` to our ICFG. An `Intent` can be *explicit* or *implicit*. In an *explicit* `Intent`, the target is given by an explicit class name. `APMHunter` obtains the class name carried by the Intent and then links the target class with the Intent. An *implicit* `Intent` only specifies the functionality that it wants to invoke instead of the class name of the target component. To infer the target of an *implicit* Intent, we adopts the `IC3` [69] tool. `IC3` transforms the ICC problem into a Multi-Valued Composite (MVC) constant propagation problem (i.e., finding all possible values of objects concerned at a certain program point).

IC3 resolves the MVC constant propagating problem with the COnstant propAgation Language (COAL) and then employs a COAL solver to solve the problem. By building a Program Dependence Graph (PDG) and performing an MVC data flow analysis, IC3 can infer the arguments in an implicit Intent, and then find the target component for the Intent. We first run IC3 to collect the inter-component communications between components. The results can be bound with FlowDroid by leveraging the API `IC3ResultLoader`. Once this setup is accomplished, the data flows introduced by ICC can be appended to our ICFG.

4.2.4 String Analysis

Motivation. As introduced in §3, developers can use APMs to collect runtime values with the built-in logging functions. Hence, we conduct the string analysis to explore the data collected by developers. Such data can reflect developers' intention of using APMs and determine the correlation between APMs and apps (e.g., category, functionality).

Methodology. To capture the values collected in the app, we leverage Violist [70], a String analysis framework for Android, to perform String analysis. Specifically, Violist separates the *representation* and *interpretation* of string operations. To compute the value of a string, Violist defines an Intermediate Representations (IRs) to capture string operation. After computing the string value in the IR format, the result (in IR format) is translated to a string. Finally, the String analysis function (from Violist) is integrated into our APMHunter framework.

4.3 Usage Identification

Motivation. The key goal of the *usage identification module* is to understand how APMs are used by developers in terms of

the APIs used by developers and their context information (e.g., where the APIs are instrumented).

Methodology. APMHunter records the following usages of an APM for a given app:

(I) The position of an APM's initialization. When an APM is initialized by an app, it starts monitoring the performance of the app. Therefore, locating an APM's initialization can provide insights on developers' usage patterns of an APM and help us detect potential APM misuse.

(II) The privacy leaks through APMs. We detect the privacy leaks from two aspects: (1) privacy leaks from user inputs. For example, if there is a field for a password, the data of the field can be leaked to the APM in the app. Consequently, developers receive sensitive data from users; and (2) privacy leaks from permission-related APIs. For example, the location data can be obtained with method `getLastknowLocation()` [45], which can be leaked to developers via APMs. Such leakage is defined as permission-related privacy leak as the method `getLastknowLocation()` is associated with the ACCESS_COARSE_LOCATION or ACCESS_FINE_LOCATION permission.

Next, we illustrate the process of these two types of privacy leakage detection.

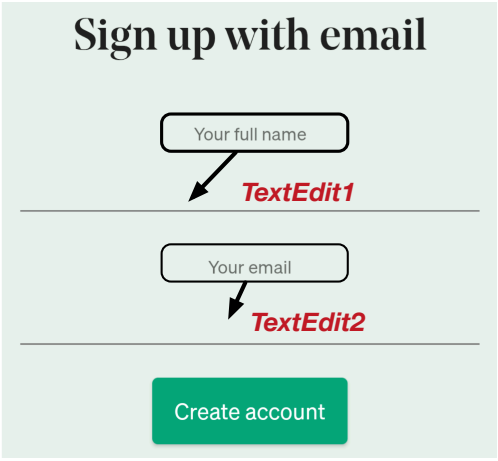


Fig. 7: Adjacent Label sample from Medium App

• Users' account information and profiles. It reveals users' personal information in an app, which includes username, first name, last name, password, email account, birthday and phone number;
• Location information. It represents users' home addresses. Different from the system derived location (i.e., latitude and longitude), here we focus on the delivery addresses/home addresses, which are input by users; and
• Users' financial information. The financial information is mainly related to users' payment information, such as credit card numbers, expiration date, and security code.

To detect the sensitive information, we follow the idea in UIPicker [71] to perform the GUI analysis [72]. First, for a given app, we parse the resources file (e.g., layout files, strings) in the app. Next, we identify the textual semantics of UI element with natural language processing and fetch corresponding private data. Specifically, a Support Vector Machine (SVM) classifier is built with the supervised machine learning. It takes a UI element's context in the whole layout into consideration to determine whether an element is privacy-sensitive. If a UI element is associated with the private data, we track the access of the UI element starting from the *findView* APIs, including `findViewById()`, `findViewWithTag()`, `findViewsWithText()`, by adding these to `SourceAndSink` file as source. The `SourceAndSink` is served as the input for FlowDroid module in APMHunter to detect privacy leaks. The evaluation of sensitive UI element identification is presented in Sec. 4.5.

Privacy leaks from permission-required data. We use the static analysis to identify the permission-required sensitive data leaked to APMs. PScout [48], a widely used API-Permission mapping set, is leveraged to collect system APIs whose executions require certain permissions to be granted. For example, method `getLastKnowLocation()` requires the ACCESS_COARSE_LOCATION or ACCESS_FINE_LOCATION permissions to be granted. Then, `APMHunter` builds the `SourcesAndSinks` file for finding privacy leak paths using FlowDroid [67].

Next, we use an example to illustrate why settings can detect the privacy leak. If an app leverages `getLastKnowLocation()` to obtain the users' location, then the location data is passed to an APM. As a result, users' privacy is leaked. Such privacy leakage is represented by a valid path from `getLastKnowLocation()` to an APM logging API, which can be detected by our tool `APMHunter`.

(III) Characterizing the used APM APIs and their context information. To infer the developers' intentions of using APMs, we record the APM APIs used in apps. For example, the API `setRevenue` in Adjust APM is used to track the revenue for an app. By recording the usage of this API, we learn that the developers aim at tracking their revenues.

Specifically, we record the following information for understanding the APM usages:
• The methods and classes/interfaces that use APM APIs. For methods, we check whether these methods are lifecycle methods or inheritance methods of lifecycle methods in Android [45]. For classes/interfaces, we check whether these classes represent app components or app instances;
• We record the code segments under monitoring. For instance, sometimes, textcolorreddevelopers leverage the tracking APIs in APMs to understand how app users interact with their apps, and which parts in apps draw users' interests; and
• We also infer the variables collected by developers via APMs using string analysis. By inspecting variables collected by developers, we can infer developers' intentions.

4.4 Implementation

APMHuner is built atop FlowDroid [67]. In APMHuner, we link the entry point methods in an app with the `dummyMain`. The `dummyMain` method, a faked main method provided by FlowDroid, allows developers to traverse the ICFG through the `dummyMain` rather than starting from all entry points. FlowDroid also provides an interface (see class `Ic3ResultLoader`) for loading the results produced by IC3. The string analysis module (from Violist) and additional features for exploring APM usages are integrated into our tool APMHunter. More implementation details of APMHunter can be found on our project page.

4.5 Evaluation on APMHunter

4.5.1 Accuracy on APM detection

We evaluate the accuracy of APMHunter by using 500 randomly-selected apps from Google Play. To determine whether an app is obfuscated or not, we disassemble each app with Apktool [73], and then manually check whether its package name is obfuscated or not. As shown in Table 3, there are 123 obfuscated apps and 377 non-obfuscated apps. Note that, since identifying the obfuscated apps is non-trivial process and difficult. For obfuscated apps here, we just consider the apps with package name obfuscation.

TABLE 3: The Benchmark of Selected Apps

	With APM	Without APM	Total
Obfuscated Apps	70	53	123
Non-obfuscated Apps	122	255	377
Total	192	308	500

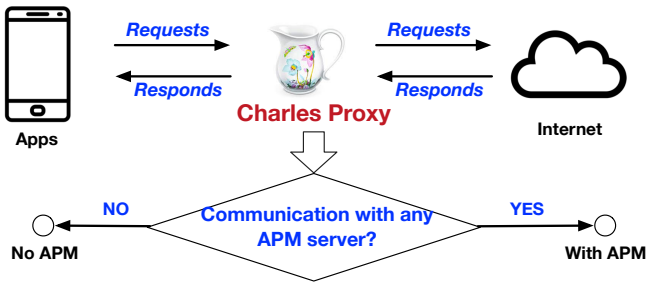


Fig. 8: Workflow for Building the Benchmark

Fig. 8 shows the workflow for building the benchmark, which is used for evaluating the performance of APMHunter. First, we set up a CharlesProxy [74] and connect the target device to the CharlesProxy. CharlesProxy is a cross-platform HTTP debugging proxy server application, which can capture the communication between target device and remote servers. Once an app sends requests to a remote server or receives responses from a remote server, the CharlesProxy captures these packets. If an app uses a specific APM, the APM sends the collected data to the APM's remote server. The captured packets sent from an app to a specific APM server suggest the APM used by the app. For example, if an app contains the Apteligent APM, we can capture the packets sent to `appload.ingest.crittercism.com` via CharlesProxy. Note that Apteligent was named as crittercism. In this way, we can determine whether an app uses an APM even if the app is obfuscated. In practice, each app will be run automatically for one hour with our python script that is built upon the Android Monkey framework [75]. We then analyze all packets captured by CharlesProxy to determine whether an app uses certain APMs. As APMs periodically upload data collected to remote servers, the execution coverage of the target app does not affect the result.

By inspecting the packets sent from apps, we find that 192 apps use APMs, and 308 apps do not employ any APM as shown in Table 3. We apply APMHunter to these sample apps in order to evaluate its performance. The results are shown in Table 4. In Table 4, the columns represent the statistical results by referencing the benchmark, the rows represent the statistical results given by the APMHunter. For example, 60 (row 1, column 1) represents that there are 60 obfuscated apps that use APMs, and APMHunter correctly identifies them. 10 (row 3, column 1) represents that there are 10 obfuscated apps that use APMs, but APMHunter fails to find them. The precision of APMHunter is 97.7% (168/172) and the recall of APMHunter is 87.5% (168/192).

TABLE 4: The Performance of APMHunter on APM Identification

APMHunter	With APM (Benchmark)	Without APM (Benchmark)	Total
OA-With	60	4	64
NO-With	108	0	108
OA-Without	10	49	59
NO-Without	14	255	269
Total	192	308	500

OA-With: obfuscated apps that use APMs;
NO-With: None-obfuscated that apps use APMs;
OA-Without: obfuscated apps that do not use any APM;
NO-Without: None-obfuscated app that do not use any APMs;

After manually inspecting 28 apps that APMHunter fails to make the correct decisions, we have the following observations. The reason why APMHunter cannot reach 100% precision is due to the use of other libraries (e.g., Google GMS). When an app uses a third-party library, especially Google GMS or Google Ads, the Firebase APM is invoked by the third-party library. However, the original app does not use the Firebase APM. For non-obfuscated apps, APMHunter can distinguish that the APM is invoked by a third-party library. However, APMHunter cannot determine whether the APM is invoked in the third-party library or the host app if the app is obfuscated, and thus it may make incorrect decisions.

The encoding algorithm in the APM identification module is version insensitive. That is, the method signature of a method in an APM's two different versions can be different. Our encoding algorithm is built upon two invariants: class inheritance relations and caller-callee relations, which can be changed during evolution. If an app adopts an out-of-date APM, APMHunter may not identify the APM in use. This is the obstacle in achieving a higher recall. The reason why we do not support all versions of APMs is that some vendors only provide the latest SDK versions and the previous versions are no available.

TABLE 5: The Performance of APMHunter on Detecting Sensitive UI Elements

	Sensitive UI element	Insensitive UI element
Sensitive UI (APMHunter)	383	16
Insensitive UI (APMHunter)	24	149

4.5.2 Accuracy of identifying sensitive UI elements

We use the 500 randomly selected apps to evaluate whether sensitive UI elements can be captured by our tool. We install these apps on a device and inspect the UI elements in apps. Among all samples, there are 129 apps whose UIs are not in English. By manually validating the rest 371 apps, we find 407 sensitive UI elements. Even though not all 371 apps contain APMs, we still consider them in this experiment, because the target of this experiment is assessing whether our tool can correctly identify sensitive UI elements from apps. The performance of our tool on detecting sensitive

UI elements is displayed in Tab. 5. The precision is 95.9% (383/399) and the recall is 94.1% (383/407).

The false positive rate is 4.1%. We manually inspect the 16 elements that APMHunter fails to make the correct decision and find the reason: APMHunter cannot capture the semantic context of the app, which misleads the tool to make the incorrect decision. For example, in a travel app, the "phone number" field refers to the phone number of a hotel rather than a user's phone number. Moreover, in some game apps, the "name" fields can refer to virtual names or other meaningless strings. For example, there is a name combiner app, which asks users to input two names and then combines them into one. In this app, the name field can be any string rather than a real user name.

The false negative rate is 5.9%. We find that 24 UI-sensitive elements cannot be identified by our tool and find the reasons as follow: (1) some very low-frequency texts or abbreviations cannot be inferred by the privacy-related analysis, such as DLNO (i.e., Drive License Number), SSN (Social Security Number, which is used in US); and (2) some apps use icons to provide semantic information for the required fields. For example, a mail icon indicates that uses have to input their emails in the EditText field.

4.5.3 Accuracy of privacy leak analysis

As presented in Sec. 4.5.1, 192 apps (out of 500 sample apps) use APMs. Among them, there are 141 whose UIs are in English. Therefore, our analysis is based on these 141 apps.

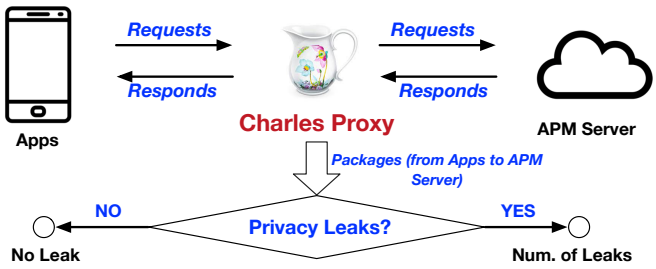


Fig. 9: Workflow for evaluating the accuracy of APMHunter in terms of privacy leaks detection.

Fig. 9 shows the workflow for evaluating the accuracy of APMHunter in terms of privacy leaks detection. We use CharlesProxy to collect packets from apps to APM servers. By inspecting the packets collected, we can determine whether users' private data is leaked to APM servers. Specifically, we first build a virtual profile for testing, which contains the following fields: email, name (both first and last name), sex, age, race, phone number, ID, home address, credit card number, credit card CVV number, location. For a given app, we connect the Pixel phone to ChralesProxy and install the app. Next, we run the app and input the data from the virtual profile when required. For example, if the app needs us to register an account, we use the data in the virtual profile to build the account. As aforementioned, the CharlesProxy can listen to the communication between apps and APM servers. We collect the packets sent from apps to their APM servers. If sensitive data (i.e., data from virtual profile) is found in the packets captured by CharlesProxy, we consider it to be a privacy leak.

By verifying each app, we find 91 leaks from 16 apps. APMHunter correctly reports 77 leaks out of 91 leaks. The recall of APMHunter is 84.6% in terms of detecting privacy leaks. We find the following reasons lead to this: (1) third-party UI framework: some apps leverage third-party UI frameworks (e.g., Butter Knife) rather than the default UI frameworks. As APMHunter does not support third-party UI frameworks at this stage, some sensitive data leaked in this way cannot be detected with APMHunter; (2) some apps use a medium (e.g., a file, SharedPreference) to transfer data, which cannot be captured. For example, sensitive data is first written to a file and then the data is retrieved from the file. The precision of APMHunter is 96.2% as we find that 3 leaks reported by APMHunter are not real leaks. This is because FlowDroid, upon which APMHunter is built. Thus, it cannot handle array indices precisely.

5 Empirical Study

We guide our empirical study by answering five research questions (RQs), which are organized by three aspects:
- **1**: The popularity of APMs in the wild (RQ 1);
- **2**: How APMs are used in practice. Specifically, we are interest in: comparing logging frameworks with APMs (RQ 2); discussing consequences of using multiple APMs (RQ 3); and learning the privacy issues raised by APMs (RQ 4); and
- **3**: What is the performance overhead of using APMs (RQ 5).

5.1 Popularity of APMs in the Wild

★ RQ 1. How prevalent are APMs in Android apps?
Motivation. We are interested in understanding whether APMs have been widely adopted in Android apps by answering the following sub-questions:
- (**1-A**): Are APMs widely adopted by Android apps?
- (**1-B**): How APMs are ranked according to popularity?

Methodology. (**1-A**) We implement our obfuscation-resilient approach for APM detection (see §4) in APMHunter and use it to characterize the APM usage in a large set of apps. (**1-B**) Then, we calculate the popularity of APMs based on the results from the previous step.
Subject Apps. We randomly crawl 500,000 Android apps from Google Play, which cover 25 main app categories. Note that, in Google Play, the game apps are grouped in different sub-categories. In our taxonomy, we consider all the sub-categories of game as one (i.e., Game). The sizes of these apps range from 100KB to 1.2 GB. The distribution of apps across the 25 categories is shown in Fig. 10.
Results. (**1-A**) From the 500,000 apps, we find that 55,722 apps (11.1%) use APMs. The categories of these apps are summarized in Fig. 10. We observe that APMs are most popular in the apps from the following five categories: *Entertainment* (52.8%), *Lifestyle* (33.3%), *Game* (20.1%), *Weather* (18.4%) and *News & Magazines* (16.6%). On the contrary, APMs are infrequently used in the apps from the following categories: *Library & Demos* (2.13%), *Book & Reference* (4.76%), *Finance* (6.41%), *Productivity* (6.57%), and *Comics* (7.29%).

As developers mainly use APMs to monitor the performance of their apps, we find some clues about the different adoption rates:

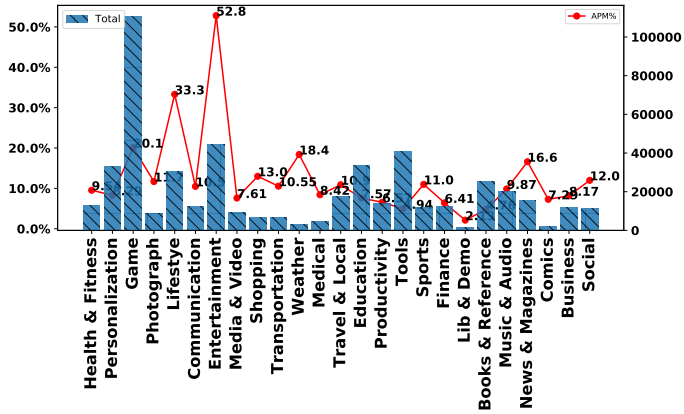


Fig. 10: APM Usage across App Categories.

• **User Experience**: For Entertainment and Game apps, the runtime performance of these apps can strongly influence user experience. Poor user experience decreases satisfaction, loyalty, and credibility. They can leverage APMs to detect performance bottlenecks and fix bugs promptly. Thus, leveraging APMs can assist developers in improving user experience.

• **Performance**: For Weather, Lifestyle, News apps, they request data (e.g, weather data, news data) from remote servers and display them to end-users. Thus, monitoring and ensuring the network performance would be the major concern for developers to adopt APMs.

For the most time, apps from Library, Demos, and Book categories run locally. The runtime performance may not be as important as apps from other categories. As a result, the adoption rate for using APMs is low.

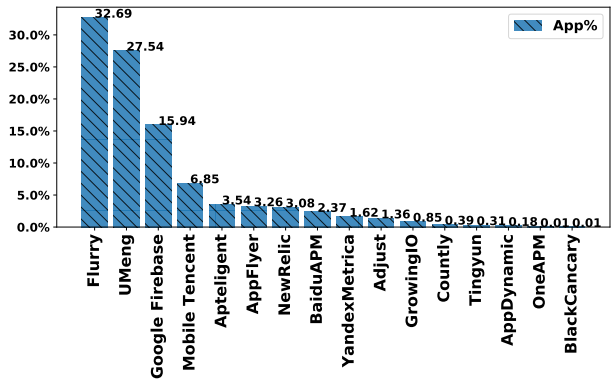


Fig. 11: Share of APMs in terms of Popularity

(**1-B**) For all the 25 APMs we studied based on 224,039 app samples, we calculate the frequency of each APM. The share of each APM is displayed in Fig. 11. The top popular APMs are: Flurry APM (32.69%), UMeng (27.54%), Google Firebase (15.94%), Mobile Tencent (6.85%), Apteligent (3.54%), and AppFlyer (3.26%). Furthermore, some APMs are nearly not adopted (<0.01%) in the sample apps, including AndroidGodEye, AppPulse, ArguAPM, CA Mobile, Dynatrace, Ironsource, OpenInstall, Sentry, Site 24*7.

• **Commercial APMs. vs. Open Source APMs.** Besides, we compare the use of commercial APMs and open-source APMs. Surprisingly, we find that developers overwhelmingly choose commercial APMs comparing to open source APMs, i.e., 99.6% of the apps are with commercial APMs, whereas only 0.4% of all apps choose open source APMs. Based on the comparison between open-source and commercial APMs, we can conclude the following reasons for choosing commercial APMs:

• **Functionality.** We find that in most cases commercial APMs contain all functions in open-source APMs. It means open-source APMs do not have any unique part to make them competitive;

• **Usability.** Most open-source APMs do not provide any dashboard for collecting and analyzing data. Developers have to deploy or even build the dashboard on their own, which can be time-consuming. However, for commercial APMs, a dashboard is provided where all collected data can be displayed;

• **Costs.** Most commercial APMs allow developers to deploy one app for free. Developers are charged if they have more than one app published. However, this can be easily bypassed by creating multiple accounts with different email accounts; and

• **Technical Support.** Besides, technical support is another factor that can affect the choice of APMs. As open-source APMs lack such support, developers have to maintain the APMs (e.g., bug fixing) on their own.

Therefore, we can conclude that compared to the service from commercial APMs, current open-source APMs are not strong enough to motivate app developers to choose them.

• **Manual instrumentation vs. Automatic instrumentation** Two APMs (App Pulse and CA Mobile) support automatic instrumentation. Other APMs only support manual instrumentation. Based on the 500,000 apps, we find that 17 apps (<0.01%) are instrumented with automatic instrumentation. Developers overwhelmingly choose APMs that support manual instrumentation. The reason can be twofold:

(1) Customization: Two APMs (App Pulse and CA Mobile) that support automatic instrumentation do not allows developers to make any customization, such as recording a value of a variable. In contrast, other APMs allow developers to collect runtime variable with logging functions; and

(2) Easy of use: After trying all the APMs under examination, we find that it is not hard to configure and use an APM that supports manual instrumentation. In terms of easy of use, two types of APMs is similar.

Popularity. Among the 500,000 Android apps, 55,722 (11%) apps use the APMs studied in this paper. In particular, APMs are most popular in apps from the categories, including Entertainment (52.8%), Lifestyle (33.3%), Game (20.1%), and Weather (18.4%).
Among the 25 APMs, the top 6 popular APMs are: Flurry (32.7%), UMeng (27.54%), Google Firebase (15.95%), Tencent (6.85%), Apteligent (3.54%), and AppFlyer (3.26%).
Surprisingly, we find that comparing to open source APMs, developers overwhelmingly (99.6%) choose commercial APMs, considering the functionality, usability, costs, and even technical support.

5.2 APMs in Practice

★ RQ 2. Do developers still use logging frameworks (e.g., android.util.Log) even they have logging functions from APMs?

Motivation. APMs provide APIs which allow developers to collect runtime data. Meanwhile, the logging functions can also be achieved by Android's built-in logging function (i.e. `android.util.Log`) and other logging frameworks. Therefore, we aim at understanding the intention for using both APMs' logging APIs and general logging frameworks (e.g, `android.util.Log`).

Subject Logging Frameworks. We select four most widely-used logging frameworks in Java and Android for comparison [1], including `android.util.Log`, `org.slf4j`, `SLF4J-android`, and `java.util.logging.Logger`.

Methodology. Similar to APM detection, we used the detection approach described in §4 to identify the logging frameworks used in apps. We randomly selected 100 apps that use both logging functions from APMs and logging frameworks. Note that we only consider the APMs that provide logging functions in this task. It is because we want to learn from the apps that leverage both logging frameworks and logging functions from APMs.

Similar to APM detection, we employ the detection approach described in §4 to identify the logging frameworks used in apps. We also record constants in logs and the type of logs (i.e., debug, warning, information). The constants in a log can present the semantic content of the log. With the type of a log, we can infer the developers' intention. Although the runtime data is no longer available for developers when an app is published, we can still extract the constants in logs and the type of logs from apps with static analysis.

Results. We find that 224,039 out of 500,000 apps (44.8%) apps adopt one or more logging frameworks. 22,739 apps out of 55,722 APM-integrated apps (40.8%) use both APM logging functions and general logging frameworks.

The use of logging frameworks in practice. We collect logs and record log contents from 100 sample apps. For the contents collected in these logs, they can be categorized into four groups including *string constant*, *integer constants*, *type of logs*, and *variables*. More specifically, the *string constant* and *integer constants* represent the constant values used in logs. We find that developers leverage *string constant* to record actions. For example, the message "user rated the app on appirater" is used to inform developers that users have already rated the app. The *integer constants* are used to record the line numbers in source code for debugging. The *variables* collected can assist developers in debugging their apps.

By analyzing the log content collected from 100 sample apps, we find that developers leverage the logging frameworks for debugging purposes, such as recording variables for debugging, and recording source code line numbers.

The use of APMs in practice. Different from using logging frameworks, developers use APM mainly for monitoring apps' performance and understanding user behaviors. To be specific, based on our observation of 100 sample apps, we learn that developers mainly use APMs for the following purposes:

• Monitoring the runtime performance (i.e., memory, runtime bugs, CPU usage, network performance) of their apps;
• Detecting network problems: Developers use APMs to diagnose potential network connection problems;
• User profiling: The goal of user profiling is to understand and categorize app users [76]. Specifically, we observe that developers mainly care about the followings: (1) understanding users' geographical distribution by inspecting their IP locations; (2) knowing users' unique hardware device IDs; (3) obtaining the device module information (e.g, Google, SAMSUNG, LG); (4) obtaining the preferences stored. Understanding users' profiles can be a double-edged sword. On the one hand, the data collected can be leveraged to improve the app, but on the other hand, it can result in privacy leakages. For example, obtaining preferences from users may violate privacy protection rules, such as GDPR [77], [78].
• Understanding the execution paths: Another key usage of APM is to learn how users interact with an app. A common practice is that developers instrument logs via APMs at all lifecycle methods. When a user enters or leaves a page, the APM can collect such information for developers.

By collecting this information, developers can benefit from: (1) knowing which pages are popular among app users and which pages are seldom visited, (2) learning the time spent on each page assists developers in improving their apps, and (3) knowing the execution path can help developers infer users' behaviors and preferences.
• Using trace statements to monitor certain code segments: some code segments in an app require several resources (e.g., CPU and memory). Thus, monitoring the code performance at runtime is a key usage of APMs. By inspecting real-world apps, we find that developers place traces mainly for two types of code segments: time-consuming code and frequently-visited code. For the former, the execution of a code segment can be time-consuming subject to the environment, device, and other factors. For example, developers often monitor the downloading tasks (e.g., downloading a video/image from a server). The latter refers to the code segments that are executed more than once. Thus, developers place trace statements around them to monitor the performance and then optimize the code when possible.

Use both logs and APM logging functions. We find developers use logging frameworks even APMs provide such functions. The reason is two-fold:
• **Only using logging frameworks:** We find that developers cannot collect log data once their apps are released if they only using general logging frameworks. If developers intend to collect data from end users with logging frameworks, additional efforts are required. They have to set up a server and send the data collected with logs to the server. However, this can be easily achieved with APMs.
• **Only using APMs:** If only APMs are used in the apps, developers cannot obtain timely responses for local tasks, especially for local debugging. The data collected by APMs cannot be directly shown to developers when developers use some logging functions from APMs. It is because that most APMs do not upload the data to servers frequently.

To wrap up, the logging frameworks cannot collect runtime log data once the apps are released whereas the logging feature in APMs cannot provide timely feedback for local

tasks, especially for local debugging. Consequently, there is a need for using both logging frameworks and the logging features from APMs. It also suggests that APM vendors may let their APMs support local debugging.

Logs vs. APMs. 22,739 app out of 55,722 APM-integrated apps (40.8%) also adopt logging frameworks. APMs cannot fully replace logs and vice versa. The reason is that APMs are not suitable for local debugging whereas most log frameworks cannot collect runtime data once apps are deployed. Therefore, the intentions of using APMs and logging may be different.

★ RQ 3. What are the consequences of using multiple APMs in one app?

Motivation. Based on our observations in RQ1, we find that app developers may use multiple APMs in one app. Specifically, 10,531 apps (out of 55,722 apps,18.9%) use more than one APMs in apps. Therefore, we analyze how multiple APMs work at runtime. Specifically, our exploration is performed from two aspects: (**3-a**) why do developers leverage multiple APMs in one app? and (**3-b**) do multiple APMs introduce any side effect to apps?

Methodology To answer **3-a**, We manually inspect 100 apps that use multiple APMs in a single app. We first reverse engineer each app and then manually inspect the usage of the APMs in it. We collect and compare the following information for different APMs in a single app:

• The APIs leveraged by different APMs;

• The data collected by these APMs can be used to infer developers' intention;

To answer **3-b**, we evaluate the performance when multiple APMs are integrated into one app. Here, we select 4 APMs, including two open-source APMs (ArgusAPM [37] and AndroidGodEye [35]) and two commercial APMs (Baidu and UMeng), for this task.

Baseline experiments.

• In the first baseline experiment, we evaluate whether these APMs can work properly at runtime. If one APM cannot work properly, it can affect our evaluation.

We evaluate all four APMs with the demo app respectively in the baseline experiment. We confirm that all four APMs can monitor the performance of apps at runtime. That is, they correctly offer the functionalities as they claimed, such as capturing crashes, collecting network performance, detecting ANR errors, collecting CPU and memory usages. The results can be found in Table 6.

• In the second baseline experiment, we evaluate whether a performance issue can be detected with one APM but missed when two APMs are used together.

In the first baseline experiment, we already test and confirm that all four APMs can work properly in terms of performance monitoring. Thus, we need to test whether the performance issues can still be detected under different APM pairs. Hence, we employ six APM pairs in this experiment, including <ArgusAPM, AndroidGodEye>, <ArgusAPM, BaiduAPM>, <ArgusAPM, UMengAPM>, <AndroidGodEye, BaiduAPM>, <AndroidGodEye, UMengAPM>, <BaiduAPM, UMengAPM>. After testing with different APM pairs, we find that if a performance issue can be detected with one APM, it can also be detected when two APMs are used together.

TABLE 6: Baseline Experiment Results.

	Java-Native Crash	Network	ANR	CPU	Memory
ArgusAPM	✓-✗	✓	✓	✓	✓
AndroidGodEye	✓-✗	✓	✓	✓	✓
Baidu APM	✓-✓	✓	✓	✓	✓
UMeng APM	✓-✓	✓	✓	✓	✓

Multiple APMs in one app. For two open-source APMs, we first manually instrument these two APMs by adding additional log statements to indicate the APM used. When a function in a certain APM is invoked, we are informed by the logs instrumented. Next, we integrate these APMs (i.e., open-source APM) with a self-built demo app. The demo app uses 5 components to present 5 problems, including Java-side crash, network communication error, ANR, large CPU usage, large memory consumption. However, these two open-source APMs do not cover the native crash capture function, we use two commercial APMs to capture the native crashes. Specifically, we build another demo app (demo-app-2), which embeds both Baidu APM and UMeng APM. At runtime, we trigger the native bug in the demo-app-2 to determine which APM (Baidu APM or UMeng) can capture the signal.

Results.

TABLE 7: Multiple APM Performance

	Crash	Network	ANR	CPU	Memory
First Init. APM	✗(J)/✓(N)	✓	✓	✓	✓
Last Init. APM	✓(J)/✓(N)	✓	✓	✓	✓

First Init. APM: the APM that is first initialized in the app;

3-a. By checking the apps with multiple APMs, we find the following intuitions that developers use multiple APMs:

• **Bad programming practice.** First and foremost, it comes from the developers' poor programming practices. Developers leverage different APMs to monitor the performance of different modules in a single app. The intuition is twofold: (1) modules are developed by different groups in a company. They use different APMs for their modules. When the entire app is built by merging modules from different groups, multiple APMs are integrated into one app; and (2) developers target at distinguishing the performance results from different parts of an app. By instrumenting different APMs for different modules, it is easy to locate the source of performance bottlenecks. However, we do not suggest such practice as it can introduce side effects (see Table 7) and performance overhead (see Fig. 12, 13, 14).

• **Combine advantages in different APMs.** Each APM has its unique features and advantages. For example, the `Adjust` APM provides additional functions for developers to monitor the ads in an app. The `UMeng` APM provides additional options for developers to monitor users' behavior under the game context. Thus, some app developers leverage different APMs with various concerns. By demystifying the APMs, we find that most functions in APMs are overlapped (see Table 1). Thus, using multiple APMs contributes less to the monitoring capability.

3-b. To understand the side-effects introduced by the execution order of APMs, we conduct two-round testing for comparison. For the first round, we initialize ArgusAPM and then initialize AndroidGodEye in the demo app. For the second round, we test in the opposite order. The result can

be found in Table 7. In general, for most functions, including network diagnosis, ANR, CPU utilization, and memory usages, both of them can capture the information. However, as for the crash from the Java side, only the later initialized one can capture the crash. For native crash, we test with Baidu APM and UMeng, we find the native crash can be captured by both APMs. This is because when capturing the Java-side crash, APMs implement an `UncaughtExceptionHandler` to the main thread, and only the later Handler is considered as `UncaughtExceptionHandler` because a thread can only have one default `UncaughtExceptionHandler`. That is the reason why only the later initialized APM can capture the Java-side crash. However, when considering the native crash, each APM only implements a listener to capture the crash signal. That is the reason why both APMs can capture the native-side crash.

Consequences of using multiple APMs. By manually inspecting 100 apps with multiple APMs, we find that there is no sound evidence to embed multiple APMs in one app. In this study, we find that the key functions in each APM are the same even though the implementations can be different. Leveraging multiple APMs cannot improve the APMs' monitoring capability. Even worse, we find that using multiple APMs in one app can lead to side effects.

★ RQ 4. Will app developers collect sensitive data using APMs?

Motivation. As APMs allow developers to collect values of variables with build-in logging functions at runtime, we aim at checking whether developers exploit APMs for stealing sensitive data from users.

Methodology. We perform our experiments on the 55,722 apps with the APMs with the approach presented in Sec. 4.3.

Results. As a result, we find 23,403 apps out of 55,722 apps (42%) collect sensitive data from users with APMs. In total, 99,583 leaks are explored in all these 23,403 apps. The top-ranked sources for these leaks are: `TelephonyManager::getDeviceId()` (13,943 leaks), `android.location.LocationManager::getLastKnownLocation()` (13,906 leaks), `org.apache.http.HttpEntity::getEntity` (5,030 leaks), and `android.location.Location::getLatitude()` (2,852 leaks). Next, we manually analyze data collected by developers. Then, we category them into the following groups based on their intentions. For example, `android.location.LocationManager::getLastKnownLocation()` and `android.location.Location::getLatitude()` are used to obtain the locations of app users, therefore, we categorize them into the same group based on the same intention. As a result, we find the following key intentions for developers to collect user data:

- Hardware device information: developers care about what kind of devices that app users adopt. Specifically, it includes (1) the unique hardware ID of a device; and (2) the module of a device (e.g., Google, SAMSUNG, LG).
- Users' location information: developers care about the geographic distribution of their app users;
- Uses' preferences: developers also care about users' preference settings in the apps. Note that, this information can be private as some private data can be embedded in the preferences;

Privacy leakage detection and protection. Users can leverage our APMHunter tool to detect whether the apps collect their sensitive data via APM. Unfortunately, most app end users (e.g., elder people, children, non-IT people) do not have any CS background. It can be difficult for them to use our tool to detect malicious behaviors even if the tool is available. A more practical solution is fixing the problem from other stakeholders. We suggest two possible solutions:

Solution 1: Google Play can analyze the apps submitted by developers to identify potential privacy leaks through APMs.

Solution 2: APM vendors can limit the operations/APIs in APMs and provide non-sensitive data to developers. For example, for location data, the APMs can provide non-sensitive data (e.g., the city, the state, the country) rather than specific GPS coordinates (e.g., latitude position, longitude position).

Privacy. It is a common practice (42%) that apps collect private data through APMs. Developers care more about the following private information: (1) hardware device ID; (2) user location information; and (3) users' preferences. For APM vendors and app distributors (i.e. Google Play), they should carefully check the APM-based malicious behaviors.

5.3 Performance Overhead

★ RQ 5. What is the performance overhead of using APMs?

Motivation. For this RQ, we intend to discuss the additional computing resources introduced by using APMs.

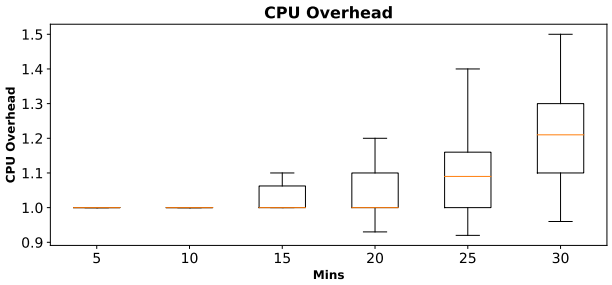


Fig. 12: CPU Overhead

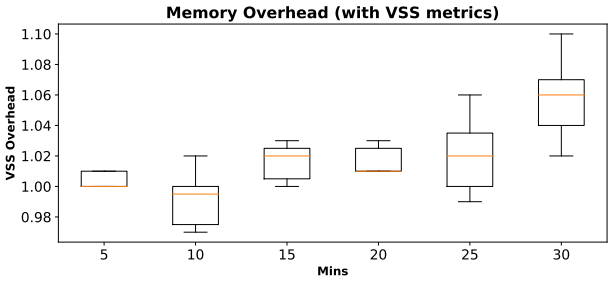


Fig. 13: Memory Overhead (with VSS metrics)

Environment. We randomly select 50 apps that each of them uses only one APM under study. These apps are installed on our test machine, which is a Pixel phone. The Android version of the Pixel phone is Android P (API 27).

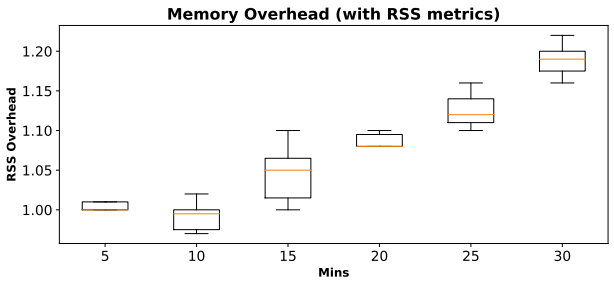


Fig. 14: Memory Overhead (with RSS metrics)

Methodology. We develop a tool to repackage an APK by removing all APM API invocations from the app. As a result, for each APK, we obtain a repackaged version that does not contain any APM usage. To evaluate the performance overhead introduced by APMs, we compare these two versions (with APMs and without APMs) using Sapienz [79], which is a widely used fuzzing analysis tool for Android apps. We test each app using Sapienz for half an hour. During the testing, the memory consumption of an app is calculated with `top` utility tool in adb (`adb shell top`) and we further resort to `adb shell dumpsys cpuinfo` to get the CPU usage information. We set the sampling interval to 10 seconds.

Results. Fig. 12 shows CPU usage overhead with the box plot. The x-axis represents the total minutes that the app has been executed. The y-axis represents the overheads. For example, node (5,1.01) represents that after 5 minutes, the CPU usage of the app with APM is 1.01 times higher than the app without APM. The average values of CPU overhead can be 1 to 1.2 times higher than that of apps without APMs. The median values of CPU overhead range from 1 to 1.2. As for memory consumption, two common metrics are adopted in Unix-like systems (e.g., Android). The Resident Set Size (RSS) shows how much memory is currently used by the process (i.e., the app in our context). While the Virtual Set Size (VSS) shows how much memory is allocated to the process. Fig. 13 and 14 show the memory overhead with VSS and RSS metrics respectively. Specifically, as for VSS usage, the costs for apps with APMs can be 1 to 1.058 times higher than that of apps without APMs. The median values of VSS range from 1 to 1.06. As for RSS usage, the costs for apps with APMs can be 1 to 1.189 times higher than that of apps without APMs. The median values of RSS range from 1 to 1.192.

In summary, even though using an APM in an app requires additional computational resources, the overhead introduced is not high.

Performance Overhead. APMs introduces overheads in terms of memory and CPU usage. By conducting an experiment on 50 apps, we find that the overhead introduced by APMs is not high.

6 Lessons Learned

In this section, we summarize our findings in this study and provide suggestions and tips for stakeholders.

6.1 Suggestions for APM Vendors

(1) Avoid requesting dangerous or deprecated permissions. As summarized in Sec. 3.9, several APMs request deprecated or even dangerous permissions, such as READ_LOGS, READ_PHONE_STATE, from app users. These permissions are officially marked as dangerous and deprecated by Android [45].

Using deprecated permissions can cause some functions in APMs to be no longer valid/supported in the latest Android versions, which can lead to potential compatibility issues. Even worse, using deprecated permissions in apps can result in critical security issues. For example, the work [47] shows that the attacks can be launched once the permission `SYSTEM_ALERT_WINDOW` is authorized.

(2) Avoid accessing sensitive files. As presented in Sec.3.9, some files (e.g.,`proc/stat`) in the Android system store the sensitive data. APMs must be prohibited from accessing these files. Inappropriate usage of APMs can cause privacy leaks. As introduced in Sec. 3, some optional solutions can be adopted by vendors to prevent the use of these sensitive files. For example, they can access the file `/proc/cpuinfo` rather than the file `proc/stat` for CPU usage.

(3) Introduce additional features for handing known performance anti-patterns. As presented in Sec.3.10, we summarize 8 common performance anti-patterns from existing studies [51], [53]–[55] to evaluate whether existing APMs are suitable for resolving them. Unfortunately, existing APMs cannot diagnose all anti-patterns. Sometimes, additional human efforts are required to explore performance bottlenecks. Therefore, we suggest APM vendors extend the functionalities of APMs to support all common performance anti-patterns aforementioned.

(4) Respecting to the changes on the Android System. We find that APM vendors update their APMs without considering the changes on Android (see. Sec. 3.11). However, the changes in the Android system can influence the performances of APMs. For example, APMs can access file `/proc/stat` for collecting CPU information. However, since Android 8.0 (API 26), such access is prohibited. Therefore, we suggest that APM vendors respect the changes in the Android system when updating their products.

(5) Privacy management. When an app is integrated with an APM, the users' runtime data is collected by the APM. Users cannot terminate such collection. From users' perspectives, they should be able to determine whether they would like to share their data. Thus, APM vendors should provide such options.

(6) Preventing app developers from building malicious apps with APMs. Even though it is not a common practice to leverage APMs in malicious apps, we still find that some malicious apps use APMs to collect users' passwords, addresses, and so forth. Thus APM vendors must carefully monitor the data collected through APMs.

6.2 Suggestions for App Developers

(1) Avoid collecting private data from users through APMs. As presented in Sec.5.2, 42% of apps leverage APMs to collect private data from users. The data collected includes device ID, location information, device modules, and so forth. Malicious apps can collect users' private data

through their custom code. The existing tools/studies [80]–[83] detect malicious apps by inspecting their code or execution traces. However, our study finds that malicious apps can collect private data through APMs.

Even though data from users can assist developers in improving their apps, developers must carefully inspect the data collected through APMs. For example, developers can collect coarse-granularity location data instead of fine-granularity location data. One guidance that developers can refer to is the General Data Protection Regulation of EU (EU GDPR) [77]. Moreover, developers should clarify the data collected in their apps' privacy policies [84]–[87].

(2) Using one APM rather than more. As presented in Sec.5.2, using more than one APM, for most times, cannot contribute to app monitoring and information collection. However, more APMs may introduce additional side effects to apps, such as additional costs in CPU and memory. Besides, some properties of apps are collected more than once (e.g., CPU, memory, and ANRs). But the crash can only be capture by the latest initialized APM. Even though each APM has its unique features and advantages, we do not suggest developers for using more than one APM in an app. For example, some APMs can help developers detect whether the ads in apps are displayed at runtime. Some APMs allow developers to record how users are interacting with their game with ease. Based on our investigation, it is not hard to manually implement these features. For example, to detect whether the ads in apps are displayed at runtime, developers can leverage the logging functions in the APM. Therefore, developers can implement the feature needed on their own rather than importing another APM. We recommend developers to remove redundant APMs from their apps.

7 Related Work

7.1 Application Performance Monitoring

Trubiani et al.'s work [6] discussed how to use the data collected by APM to diagnose the network bottleneck in applications. Ahmed et al.'s work [4] studied the effectiveness of APMs for measuring the runtime performance of web applications. Yao et al.'s work [2] discussed the way to improve the performance of system monitoring by instrumenting logs. Willnecker et al. [7] proposed an approach to model the performance of JavaEE applications with APMs. Different from these studies, we focus on exploring the functionalities of Android-oriented APMs and discovering the usage of APMs in real-world apps instead of the ways to use the data collected by APMs.

7.2 Measurement and Monitoring for Apps

Network Measurement. Since the Android framework provides convenient interfaces for users to intercept and forward network packets, many Android apps [88] target for measuring mobile network performance. To scrutinize the measurement accuracy of these apps, many studies have been proposed. Li et al. [70] adopted the network round-trip time (nRTT) as the metric to appraise the accuracy of network measurement apps, and they found that nRTTs measured by these apps are heavily inflated. Xue et al. [89] conducted a systematic study on three types of factors, including implementation patterns of measurement apps, Android architecture, and network protocols, to learn how these factors influence the measurement results of these apps. Li et al. [90] pointed out that the delay inflation, which influences the accuracy of network measurement apps, can be introduced from (1) the long path of sub-function invocations in Android runtime; and (2) the sleeping functions in the drivers between the kernel and physical layer.

App Monitoring. To diagnose performance bottlenecks in apps, several approaches have been proposed to conduct efficient app monitoring. AppInsight [91] instrumented mobile apps by interposing event handlers to collect information on critical paths that are triggered by user transactions. Lee et al. [92] proposed a user interaction-based mobile application profiling system, which analyzes fine-grained information, including user interaction, system behavior, and power consumption, to perform Android app tuning. AndroidPerf [93], a cross-layer profiling system, leverages cross-layer dynamic taint analysis and instrumentation to obtain both the execution information and the performance information about Android apps. DiagDroid [94] adopted a dynamic instrumentation approach, which is based on abstractions of various categories of UI-triggered asynchronous executions, to capture the data related to UI interactions and diagnose UI performance of apps. Technically, all these works concentrate on app monitoring by instrumenting the subject apps rather than leveraging APMs.

7.3 Instrumentation

Instrumentation is another frequently used technique in program understanding, debugging and testing [3], [67], [79], [95], [96]. Similar to APMs, developers sometimes use logs to collect information and debug apps. Karami et al. [3] proposed an approach that uses instrumentation to analyze and model the behavior of an app. Specifically, it focused on file I/O and network connection issues by inspecting API calls. Similarly, the tool ARTist assists developers in understanding program behaviors by extracting APIs and arguments in the program [95]. The goal of ARTist is to help developers understand the program and reveal the malicious behaviors in the program. Other studies adopted instrumentation techniques to test Android apps for diagnosing bugs and monitoring performance [67], [79], [96].

8 Conclusion

Although more and more apps adopt APMs, developers use them as black-box tools. In this paper, we demystify the functionalities of APMs in detail. We discuss 9 commons functions in APMs, and we reveal 7 design defects in APMs. We also evaluate the performance side-effects introduced by using APMs. To evaluate and study how APMs are used in practice, we developed a prototype named APMHunter. We evaluate the performance of APMHunter on 500 apps from Google Play. The results show that (1) in terms of detecting APMs in apps, the precision of APMHunter is 97.7% and the recall is 87.5%; (2) in terms of identifying sensitive UI elements in apps, the precision of APMHunter is 95.5% and the recall is 94.1%; and (3) as for identifying

privacy leaks, the precision of APMHunter is 96.2% and the recall is 84.6%. Our large-scale empirical study on 500,000 Android apps implies that 23,403 apps collect sensitive data with APMs. Therefore, we suggest both app developers and APM vendors need to vigilant in protecting users' private data.

9 Acknowledgements

This work was supported by ShanghaiTech Start-up Research Fund (No. 2020F0203-000-14), Hong Kong RGC Projects (No. 152223/17E,152239/18E, CityU C1008-16G), the National Natural Science Foundation of China (No.62072046), Leading Innovative and Entrepreneur Team Introduction Program of Zhejiang (2018R01005) and the Fundamental Research Funds for the Central Universities (K20200019).

References

[1] AppBrain, "Android and google play statistics," https://bit.ly/2YZU9AI.
[2] 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.
[3] M. Karami, M. Elsabagh, P. Najafiborazjani, and A. Stavrou, "Behavioral Analysis of Android Applications Using Automated Instrumentation," in *Proc. SERE*, 2013, pp. 182–187.
[4] 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.
[5] 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.
[6] 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.
[7] 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.
[8] A. Streitz, M. Barnert, J. Rank, H. Kienegger, and H. Krcmar, "Towards model-based performance predictions of sap enterprise applications," in *Proc. ICPE*, 2018, pp. 1–3.
[9] Y. Tang, X. Zhan, H. Zhou, X. Luo, Z. Xu, Y. Zhou, and Q. Yan, "Demystifying application performance management libraries for android," in *Proc. ASE*, 2019, pp. 682–685.
[10] UMeng, "Umeng apm," https://umeng.com.
[11] Flurry, "Flurry," https://bit.ly/31RaWI1.
[12] CA, "Ca application performance management," https://bit.ly/2C2oHJ1.
[13] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, "Towards a scalable resource-driven approach for detecting repackaged android applications," in *Proc. ACSAC*, 2014.
[14] AppBrain, "Android library statistics," https://www.appbrain.com/stats/libraries.
[15] Techbencon, "Performance engineering survey: Findings from 400 dev, test, and it ops professionals."
[16] Networkbench, "Tingyun apm," http://tingyun.com.
[17] MTJBaidu, "Mtjbaidu apm," https://mtj.baidu.com/.
[18] Tencen, "Mobile tencent analytics apm," http://developer.qq.com.
[19] OpenInstall, "Openinstall apm," https://openinstall.io.
[20] NewRelic, "New relic apm," https://newrelic.com.
[21] AppDynamics, "Appdynamics apm," https://appdynamics.com.
[22] OneAPM, "Oneapm," https://bit.ly/2YZlO4F.
[23] GrowingIO, "Growingio," https://bit.ly/2VMXd0M.
[24] Google, "Google analytics for firebase," https://bit.ly/2VK9OSk.
[25] Dynatrace, "Dynatrace," https://bit.ly/3iwf8Td/.
[26] Site24x7, "Site24x7," https://bit.ly/3f3JqdO.
[27] M. Focus, "Apppulse," https://bit.ly/3e1T0gb.
[28] Apteligent, "Apteligent," https://bit.ly/2VL8leP.
[29] AppsFlyer, "Appsflyer," https://bit.ly/2VMfew9.
[30] Yandex, "Yandex metrica," https://bit.ly/2C9EiXi.
[31] Adjust, "Adjust," https://bit.ly/2VNSfB4.
[32] IronSource, "Ironsource," https://bit.ly/3gpClEO.
[33] Countly, "Countly," https://bit.ly/2AtNIfx.
[34] Sentry, "Sentry," https://bit.ly/3dZrhga.
[35] AndroidGodEye, "Androidgodeye," https://bit.ly/2VHf6yb.
[36] BlackCancary, "Blackcancary," https://bit.ly/31R4twJ.
[37] Qihu360, "Argusapm," https://bit.ly/38uzOXm.
[38] D. M. Ritchie, B. W. Kernighan, and M. E. Lesk, *The C programming language*. Bell Laboratories, 1975.
[39] B. Gough, *GNU Scientific Library Reference Manual - Third Edition*. Network Theory Ltd., 2009.
[40] Google, "Google breadpad," https://bit.ly/2C7qc8w.
[41] V. O. Safonov, *Using aspect-oriented programming for trustworthy software development*. John Wiley & Sons, 2008, vol. 5.
[42] Y. Li, T. Tan, and J. Xue, "Understanding and analyzing java reflection," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 2, 2019.
[43] A. Colyer, A. Clement, G. Harley, and M. Webster, *Eclipse aspectj: aspect-oriented programming with aspectj and the eclipse aspectj development tools*. Addison-Wesley Professional, 2004.
[44] B. Muschko, *Gradle in Action*. Manning Publications Co., 2014.
[45] Z. R. Mednieks, L. Dornin, G. B. Meike, and M. Nakamura, *Programming android*. " O'Reilly Media, Inc.", 2012.
[46] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt, "Identity, location, disease and more: Inferring your secrets from android public resources," in *Proc. CCS*, 2013, p. 1017–1028.
[47] Y. Fratantonio, C. Qian, S. Chung, and W. Lee, "Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop," in *Proc. S&P*, 2017.
[48] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the android permission specification," in *Proc. CCS*, 2012, pp. 217–228.
[49] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, and S. Weisgerber, "On demystifying the android application framework: Re-visiting android permission specification analysis," in *Proc. USENIX Security*, 2016, pp. 1101–1118.
[50] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android Permissions Demystified," in *Proc CCS*, 2011, pp. 627–638.
[51] G. Hecht, O. Benomar, R. Rouvoy, N. Moha, and L. Duchien, "Tracking the software quality of android applications along their evolution (t)," in *Proc. ASE*, 2015, pp. 236–247.
[52] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: Abstraction and reuse of object-oriented design," in *Proc. ECOOP*, O. M. Nierstrasz, Ed., 1993, pp. 406–431.
[53] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proc. ICSE*, 2014, pp. 1013–1024.
[54] S. S. Afjehei, T.-H. P. Chen, and N. Tsantalis, "iperfdetector: Characterizing and detecting performance anti-patterns in ios applications," *Empirical Software Engineering*, 2019.
[55] Z. Chen, B. Chen, L. Xiao, X. Wang, L. Chen, Y. Liu, and B. Xu, "Speedoo: Prioritizing performance optimization opportunities," in *Proc. ICSE*, 2018, pp. 811–821.
[56] PKGDiff, "Pkgdiff," https://bit.ly/2AzDLxm.
[57] X. Zhan, L. Fan, T. Liu, S. Chen, L. Li, H. Wang, Y. Xu, X. Luo, and Y. Liu, "Automated third-party library detection for android applications: Are we there yet?" in *Proc. ASE*, 2020.
[58] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "Viewdroid: Towards obfuscation-resilient mobile application repackaging detection," in *Proc. WiSec*, 2014, pp. 25–36.
[59] M. Linares-Vásquez, A. Holtzhauer, C. Bernal-Cárdenas, and D. Poshyvanyk, "Revisiting android reuse studies in the context of code obfuscation and library usages," in *Proc. MSR*, 2014, pp. 242–251.
[60] L. Xue, H. Zhou, X. Luo, L. Yu, D. Wu, Y. Zhou, and X. Ma, "Packergrind: An adaptive unpacking system for android apps," *IEEE Transactions on Software Engineering*, 2020.
[61] L. Xue, X. Luo, L. Yu, S. Wang, and D. Wu, "Adaptive unpacking of android apps," in *Proc. IEEE/ACM ICSE*, 2017.
[62] Y. Zhang, X. Luo, and H. Yin, "Dexhunter: Toward extracting hidden code from packed android applications," in *Proc. ESORICS*, 2015.

[63] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proc. CCS*, 2016, pp. 356–367.
[64] C. Zuo, Z. Lin, and Y. Zhang, "Why does your data leak? uncovering the data leakage in cloud from mobile apps," in *Proc. IEEE S&P*, 2019.
[65] S. Holavanalli, D. Manuel, V. Nanjundaswamy, B. Rosenberg, F. Shen, S. Y. Ko, and L. Ziarek, "Flow permissions for android," in *Proc. ASE*, 2013, pp. 652–657.
[66] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework," in *Proc. NDSS*, 2015.
[67] 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.
[68] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," 2010, p. 214–224.
[69] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon, "Effective inter-component communication mapping in android: An essential step towards holistic security analysis," in *Proc. USENIX Security*, 2013, pp. 543–558.
[70] L. Li, A. Bartel, T. F. Bissyandé, 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.
[71] Y. Nan, Z. Yang, M. Yang, S. Zhou, Y. Zhang, G. Gu, X. Wang, and L. Sun, "Identifying user-input privacy in mobile applications at a large scale," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 3, pp. 647–661, 2017.
[72] H. Zhou, T. Chen, H. Wang, L. Yu, X. Luo, T. Wang, and W. Zhang, "Ui obfuscation and its effects on automated ui analysis for android apps," in *Proc. ASE*, 2020.
[73] Apktool, "Apktool: A tool for reverse engineering android apk files," https://ibotpeaches.github.io/Apktool/.
[74] Charles, "Charles: Web debugging proxy application," https://www.charlesproxy.com/.
[75] Android, "Monkey framework," https://bit.ly/3bu7FDa.
[76] S. E. Middleton, N. R. Shadbolt, and D. C. De Roure, "Ontological user profiling in recommender systems," *ACM Trans. Inf. Syst.*, vol. 22, no. 1, p. 54–88, 2004.
[77] P. Voigt and A. Von dem Bussche, "The eu general data protection regulation (gdpr): A practical guide," 2017.
[78] M. Fan, L. Yu, S. Chen, H. Zhou, X. Luo, S. Li, Y. Liu, J. Liu, and T. Liu, "An empirical evaluation of gdpr compliance violations in android mhealth apps," in *Proc. ISSRE*, 2020.
[79] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective Automated Testing for Android Applications," in *Proc. ISSTA*, 2016, pp. 94–105.
[80] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: detecting malicious apps in official and alternative android markets." in *Proc. NDSS*, 2012, pp. 50–52.
[81] L. Li, D. Li, T. F. Bissyandé, J. Klein, Y. Le Traon, D. Lo, and L. Cavallaro, "Understanding android app piggybacking: A systematic study of malicious code grafting," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 6, pp. 1269–1284, 2017.
[82] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu, "Malton: Towards on-device non-invasive mobile malware analysis for art," in *Proc. USENIX Security Symposium*, 2018.
[83] L. Xue, C. Qian, H. Zhou, X. Luo, Y. Zhou, Y. Shao, and A. T. Chan, "Ndroid: Toward tracking information flows across multiple android contexts," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 3, pp. 814–828, 2019.
[84] L. Yu, X. Luo, J. Chen, H. Zhou, T. Zhang, H. Chang, and H. Leung, "Ppchecker: Towards accessing the trustworthiness of android apps' privacy policies," *IEEE Transactions on Software Engineering*, 2019.
[85] L. Yu, T. Zhang, X. Luo, and L. Xue, "Autoppg: Towards automatic generation of privacy policy for android applications," in *Proc. SPSM*, 2015.
[86] L. Yu, X. Luo, C. Qian, S. Wang, and H. Leung, "Enhancing the description-to-behavior fidelity in android apps with privacy policy," *IEEE Transactions on Software Engineering*, 2018.
[87] L. Yu, X. Luo, X. Liu, and T. Zhang, "Can we trust the privacy policies of android apps?" in *Proc. DSN*, 2016.
[88] 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 *Proc. USENIX ATC*, 2017, pp. 445–457.
[89] 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.
[90] W. Li, D. Wu, R. K. C. Chang, and R. K. P. Mok, "Toward accurate network delay measurement on android phones," *IEEE Transactions on Mobile Computing*, vol. 17, no. 3, pp. 717–732, 2018.
[91] 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.
[92] S. Lee, C. Yoon, and H. Cha, "User interaction-based profiling system for android application tuning," in *Proc. UbiComp*, 2014, pp. 289–299.
[93] L. Xue, C. Qian, and X. Luo, "Androidperf: A cross-layer profiling system for android applications," in *Proc. IWQoS*, 2015, pp. 115–124.
[94] 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.
[95] M. Backes, S. Bugiel, O. Schranz, P. v. Styp-Rekowsky, and S. Weisgerber, "Artist: The Android Runtime Instrumentation and security Toolkit," in *Proc. European S&P*, 2017, pp. 481–495.
[96] T. Gu, C. Cao, T. Liu, C. Sun, J. Deng, X. Ma, and J. Lü, "Aimdroid: Activity-insulated multi-level automated testing for android applications," in *Proc. ICSME*, 2017, pp. 103–114.

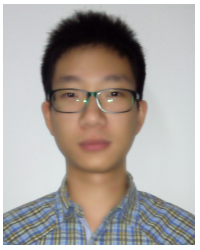
Yutian Tang received the BSc degree in Computer Science from Jilin University, China, and the PhD degree in software engineering from The Hong Kong Polytechnic University, Hong Kong SAR, China. He is currently an assistant professor with School of Information Science and Technology, ShanghaiTech University. His current research interests include mobile security and privacy, software product line, empirical software engineering, and testing. He has published papers in top-tier software engineering conferences and journals. He is a member of IEEE, HKCS, CCF and EuroSys. More information is available at https://www.chrisyttang.org/.

Haoyu Wang is an an associate Professor in the School of Computer Science at Beijing University of Posts and Telecommunications (BUPT). His research covers a wide range of topics in Software Analysis, Privacy and Security, eCrime, Internet/System Measurement, and AI Security. He received his PhD degree in Computer Science from Peking University in 2016. More information is available at https://howiepku.github.io/ .

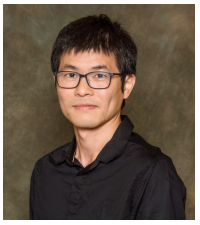
Xian Zhan received her BEng degree in computer science and technology from Wuhan University, Hubei, China. Currently, she is a Ph.D candidate in the Department of Computing, the Hong Kong Polytechnic University. Her research interests include program analysis, mobile privacy and security, Android analysis, third-party library and open-source software analysis and machine learning.

Xiapu Luo is an associate professor in the Department of Computing, The Hong Kong Polytechnic University. His current research interests include Mobile/IoT Security and Privacy, Blockchain and Smart Contracts, Network Security and Privacy, and Software Engineering. His work appeared in top venues in the areas of security, software engineering and networking. He has received eight best paper awards (e.g., INFOCOM'18, ISSRE'16, etc.) and an ACM SIGSOFT Distinguished Paper Award from ICSE'21.

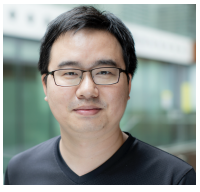
Yajin Zhou is a ZJU 100-Young professor, with both the College of Computer Science and Technology and the School of Cyber Space and Technology at Zhejiang University, China. He earned his Ph.D. in Computer Science from North Carolina State University in 2015. He has published more than 40 papers, with 7500+ citations (Google Scholar). His current research spans software security, operating systems security, hardware-assisted security and confidential computing.



Hao Zhou received the BS and MS degrees from the Nanjing University of Posts and Communications. He is currently working toward the PhD degree with the Department of Computing, The Hong Kong Polytechnic University. He was with PolyU as a research assistant from 2016 to 2018. His current research interests include system security, mobile security, app analysis, and software testing.



Qiben Yan is an Assistant Professor in Department of Computer Science and Engineering of Michigan State University. He received his Ph.D. in Computer Science department from Virginia Tech, an M.S. and a B.S. degree in Electronic Engineering from Fudan University in Shanghai, China. He is currently an IEEE senior member, ACM member, and serves as TPC Chair of International Conference on Machine Learning for Cyber Security (ML4CS 2020). He also served as Student Travel Chair of IEEE CNS 2020, 2021, Publicity Chair of IEEE INFOCOM 2017 and INFOCOM 2018. His current research interests include wireless communication, wireless network security and privacy, mobile and IoT security, and big data privacy.



Yulei Sui is a Senior Lecturer at School of Computer Science, Faculty of Engineering and Information Technology, University of Technology Sydney (UTS). He is broadly interested in Program Analysis, Software Engineering and Security. His papers have been published in the top-tier conferences and journals in the field of software engineering and program analysis such as TSE, TOSEM, ICSE, FSE and ASE. He was a plenary talk speaker at EuroLLVM 2016, and has been awarded a 2020 OOPSLA Distinguished Paper, a 2019 SAS Best Paper, a 2018 ICSE Distinguished Paper, a 2013 CGO Best Paper, an ACM CAPS award, and an ARC Discovery Early Career Researcher Award (2017-2019).

Jacky Keung received the BSc (Hons) degree in Computer Science from the University of Sydney, and the PhD degree in Software Engineering from the University of New South Wales, Australia. He is Associate Professor in the Department of Computer Science, City University of Hong Kong. His main research interests include software effort and cost estimation, empirical modeling and evaluation of complex systems, and intensive data mining for software engineering datasets. He has published papers in prestigious journals including the IEEE Transactions on Software Engineering, the Empirical Software Engineering, and many other leading journals and conferences. He is a member of the IEEE.