Empirical Software Engineering
https://doi.org/10.1007/s10664-019-09687-9

Studying the characteristics of logging practices in mobile apps: a case study on F-Droid

Yi Zeng[1] · **Jinfu Chen**[1] · **Weiyi Shang**[1] · **Tse-Hsun (Peter) Chen**[1]

Published online: 07 February 2019

Abstract
Logging is a common practice in software engineering. Prior research has investigated the characteristics of logging practices in system software (e.g., web servers or databases) as well as desktop applications. However, despite the popularity of mobile apps, little is known about their logging practices. In this paper, we sought to study logging practices in mobile apps. In particular, we conduct a case study on 1,444 open source Android apps in the F-Droid repository. Through a quantitative study, we find that although mobile app logging is less pervasive than server and desktop applications, logging is leveraged in almost all studied apps. However, we find that there exist considerable differences between the logging practices of mobile apps and the logging practices in server and desktop applications observed by prior studies. In order to further understand such differences, we conduct a firehouse email interview and a qualitative annotation on the rationale of using logs in mobile app development. By comparing the logging level of each logging statement with developers' rationale of using the logs, we find that all too often (35.4%), the chosen logging level and the rationale are inconsistent. Such inconsistency may prevent the useful runtime information to be recorded or may generate unnecessary logs that may cause performance overhead. Finally, to understand the magnitude of such performance overhead, we conduct a performance evaluation between generating all the logs and not generating any logs in eight mobile apps. In general, we observe a statistically significant performance overhead based on various performance metrics (response time, CPU and battery consumption). In addition, we find that if the performance overhead of logging is significantly observed in an app, disabling the unnecessary logs indeed provides a statistically significant performance improvement. Our results show the need for a systematic guidance and automated tool support to assist in mobile logging practices.

Keywords Software logs · Logging practices · Logging performance · Mining software repositories

Communicated by: David Lo, Meiyappan Nagappan, Fabio Palomba, and Sebastiano Panichella

✉ Yi Zeng
ze_yi@encs.concordia.ca

Extended author information available on the last page of the article.

1 Introduction

Logging is a common practice and has been widely adopted in software engineering. Log messages, which are generated at runtime by logging statements that developers inserted into the source code, present rich details about the runtime behavior of software applications.

The importance of logs has been widely identified (Kernighan and Pike 1999) and are excessively studied for server and desktop applications (Yuan et al. 2012a; Chen and Jiang 2017; Shang et al. 2015). The valuable information in logs is leveraged in various software development and operation activities including bug fixing (Xu et al. 2009), test results analyses (Malik et al. 2013), anomaly detection (Tan et al. 2008), and system monitoring (Yuan et al. 2012c; Boulon et al. 2008). The vast application and usefulness of the logs have motivated server and desktop application developers to embed a large number of logging statements in their source code. For example, the PostgreSQL 8.4.7 database server contains 6,052 logging statements in its code base (Yuan et al. 2012a).

The benefit of logging has not only been exploited by server and desktop application development but also been embraced by mobile app development (Android 2017). Prior studies (Yuan et al. 2012a; Shang et al. 2011, 2014a) found that making optimal logging decisions is very challenging. Due to the limited resources on mobile devices, intuitively, making optimal logging decisions may be even more challenging for mobile app development. For example, logging too much may cause additional performance overhead, which may lead to both slow response of the mobile apps and additional battery consumption.

However, to the best of our knowledge, there exists only little research that studies logging practices in mobile apps. In particular, prior research (Chowdhury et al. 2017) on mobile logging focuses on the energy of logging instead of the characteristics of the logging practice. Hence, in this paper, we study the logging practices in Android apps. We analyze the logging characteristics of Android apps from F-Droid (2017), the software repository for free and open source real-world Android apps. In addition, we perform both firehouse email interview and qualitative annotation on the rationale of mobile logging. Finally, we conduct a case study to measure the performance overhead of mobile logging. In particular, we aim to answer the following three research questions.

RQ1 *What are the characteristics of mobile logging practices?*
Although we confirm that logging is widely used in mobile app development, we observe that mobile app logging practices are different from those of server and desktop applications. In particular, logging in mobile apps is less pervasive than server and desktop applications, while the majority of logging statements are in *debug* and *error* levels. In addition, the logging statements in mobile apps are much less maintained and much more likely to be deleted than server and desktop applications.

RQ2 *What are the rationales of mobile logging?*
We find eight rationales of mobile logging, including *Debug*, *Anomaly detection*, *Assist in development*, *Bookkeeping*, *Performance*, *Change for consistency*, *Customize logging library*, *From third-party library*. While the majority of the mobile logging are for *Debug* and *Anomaly detection* purposes, developers often (35.4%) choose a logging level that is not consistent with the rationale. Such inconsistent logging levels may lead to potential performance overhead, security issues and missing important runtime information.

RQ3 *How large can the performance impact of mobile logging be?*
In our case study, we perform an experiment by comparing the app performance between enabling and disabling logging. In general, we find that logging can cause

statistically significant performance overhead on Android apps. We also perform a further experiment to examine the impact when only disabling unnecessary logs (e.g., debugging logs that are generated in released versions). We find that when the overall logging overhead can be statistically significant, disabling the unnecessary logs provides statistically significant performance improvement on mobile apps, comparing to enabling all logs. However, when the overall logging overhead is not significantly observed, disabling unnecessary logs would not significantly improve performance.

Our results show the distinct logging practices of mobile apps from the widely-studied server and desktop applications. However, the logging infrastructure of mobile apps is not optimized in such a scenario. Making it worse, the developers of mobile apps may not beware of the negative impact of sub-optimal logging decisions. Our results advocate the need for automated and specially designed approaches and tooling support for logging decisions of mobile app development.

Paper Organization The remainder of the paper is organized as follows. Section 2 describes our case study setup. Section 3 presents the results of our mobile logging practices case study. Section 4 discusses the implications of our results. Section 5 discusses threats to the validity of our findings. Section 6 surveys work on software logs that has been done in the recent years. Finally, Section 7 draws conclusions.

2 Case Study Setup

In this section, we present the background of Android logging and our case study setup, including the subject selection process, and the methodology of data extraction.

2.1 Android Logging

In order to ease the use of logs in practice, Android has a default logging library to print logs and the Logcat tool to view logs (Android 2017). Figure 1 shows an example of logging statement from the Android's official website (Android 2017). Every Android logging statement has an associated *tag* and *verbosity level*. Developers record the logged event using *static texts* and *variable values* related to the event. The tag of a log message is a short string usually indicating the component from which the message originates (Android 2017). The verbosity level indicates the importance of the log. There are five general verbosity levels: *verbose*, *debug*, *info*, *warn*, *error* which can be specified by calling *Log.v*, *Log.d*, *Log.i*, *Log.w*, *Log.e*, respectively. It should be noted that since Java is the official programming language for writing Android apps, the Java standard output statements such as *System.out.println* can also be used in Android log printing code. The printed messages are by default redirected to Logcat and generated as *Log.i* (StackOverflow 2017).

```
private static final String TAG = "MyActivity";
Log.v(TAG, "index=" + i);
```

Fig. 1 A logging statement example from the Android official website

Table 1 An overview of the studied F-Droid apps

Metric	Mean	Min	25th quartile	Median	75th quartile	Max
SLOC	9,818	717	1,710	3,760	9,324	324,156
# Commits	661	2	44	125	416	83,337
# Files	434	17	86	166	412	24,058
# Authors	13	1	2	4	10	771

2.2 Subject Apps

In our study, we study apps from the F-Droid repository (F-Droid 2017), which is a software repository that hosts a large number of mature and popular Android apps that are free and open source. Many apps in F-Droid are actively maintained and are also published in Google Play Store and used by many users. For example, the application *WordPress*[1] is available on both F-Droid and Google Play Store, is actively updated and has been installed over five million times as of July 20th, 2018.

F-Droid repository contains, in total, around 2,300 Android apps[2] on GitHub. At the time when we collect the apps (December 13th, 2017), we find that there exist 1,925 apps that are still reachable with available source code in the repositories. Among them, some repositories only contain little or even no Android code, or the Android code is written in programming languages other than Java. These apps with no or few Java source code will show close to zero log density and close to zero logging maintenance activities, which will bias (potentially lower the results) our results when studying all F-Droid apps as a whole. We find that 25% of the apps that have the lowest source lines of code in Java only contributes less than 1% of the entire amount of source code in F-Droid (132,979 out of 14,295,880 of all F-Droid apps). Since such apps are not suitable for our study, we discard these apps. Table 1 shows an overview of our studied apps.

2.3 Data Gathering

Figure 2 presents an overview of our case study workflow.

To analyze code structures, we use srcML[3] to parse the apps' source code. The tool converts source code into the srcML format, which is a document-oriented XML format that explicitly embeds structural information directly into the source code. The syntactic structures from Abstract Syntax Tree (AST) are wrapped with tags and can be queried using XPath expressions. Logging code includes log printing code and log object initialization code. With the syntactic structure extracted by srcML, we can accurately extract the log printing code.

Similar to prior research (Chen and Jiang 2017; Shang et al. 2015), we analyze the statements extracted by srcML to identify logging statements. We first check whether the caller object is associated with logging libraries. Then, we check whether the called method is related to a logging level (e.g., *verbose* and *info*). In order to minimize the falsely identified logging statements, we remove the ones that have their caller containing "log", yet

[1] https://play.google.com/store/apps/details?id=org.wordpress.android
[2] https://f-droid.org/wiki/page/Repository_Maintenance
[3] http://www.srcml.org/

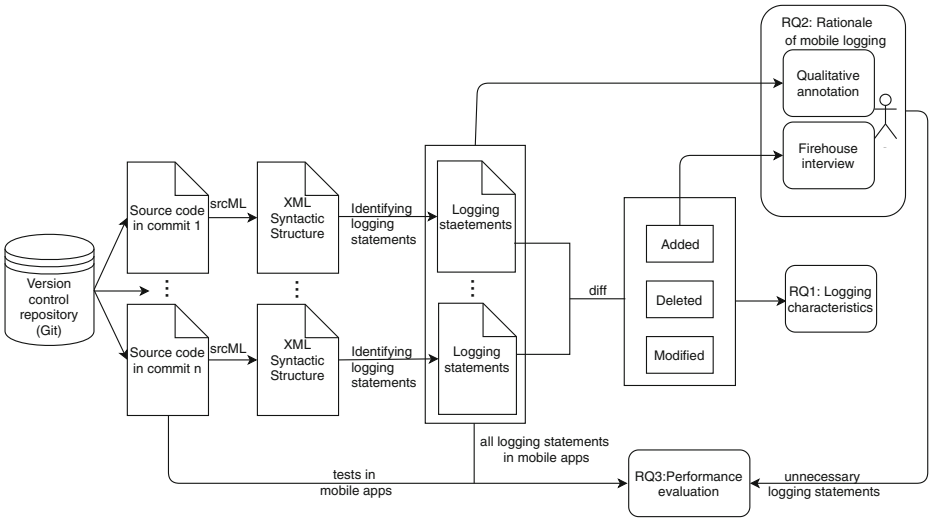


Fig. 2 An overview of our case study workflow

are not logging statements (e.g., "dialog" and "login"). We manually sample 384 pieces of logging code, which corresponds to a 95% of confidence level with a 5% confidence interval (Yamane 1973). The accuracy of our technique is 99%. The details of our logging statement identification script are available in our replication package.[4]

3 Case Study Results

In this section, we answer our three research questions. For each research question, we present the motivation for the question, the approach that we use to answer the question and the results we get.

3.1 RQ1: What are the Characteristics of Mobile Logging Practices?

3.1.1 Motivation

Prior research studies logging characteristics in open source server and desktop applications (Yuan et al. 2012a; Chen and Jiang 2017; Shang et al. 2015). The findings from prior studies advocate that logging is widely leveraged in practice, yet the logging practices require more systematic assistance and guidance. Based on those findings, follow-up research proposed various techniques in order to support logging decisions (Yuan et al. 2012b, c; Fu et al. 2014; Zhu et al. 2015; Zhao et al. 2017) (see Section 6). However, none of the prior case studies are conducted on mobile apps. Intuitively, due to the nature of mobile apps, developers may not follow the same logging characteristics (e.g., mobile apps have limited computing resources and are often UI-driven). The different characteristics of logging practices may introduce new challenges and opportunities for researchers and

[4] https://bitbucket.org/sense_concordia/mobilelogreplication

practitioners. Therefore, in this research question, we would like to study the characteristics of logging practices in mobile apps.

3.1.2 Approach

We study three dimensions of the characteristics of logging practices that are studied by prior research (Yuan et al. 2012a; Chen and Jiang 2017; Shang et al. 2015; Kabinna et al. 2016b):

- **The number and density of logging statements.** We measure the density of logging statement in the latest version (at the time of the study) of the source code of each mobile app. The density of the logging statement can serve as an indicator of how many logs are leveraged by developers. Such information may imply the importance of logs for mobile apps. In particular, we measure the number of lines of code per logging statement (LOC per log). For the studied apps, we first calculate the total source lines of code (SLOC) using *cloc*[5] and the number of logging statements (NL) using srcML. We then calculate number of lines of code per logging statement as $\frac{SLOC}{NL}$. Such a measurement is also used in a prior research (Yuan et al. 2012a).
- **The verbosity levels of logging statements.** Verbosity level can be used as a proxy to understand the purpose of the logging statements (Li et al. 2017a). Verbosity level also directly decides whether the log will be generated during the runtime of mobile apps, leading to possible energy consumption – a particularly important aspect of mobile apps (Chowdhury et al. 2017). We study the distribution of the verbosity levels of logging statements. During the data gathering process, we find that developers use Android default logging library, third-party logging libraries or custom logging classes. Therefore, the log verbosity level name may not exactly match to that of the Android default logging library. Nevertheless, developers sometimes choose a meaningful naming convention such as "printdebug" and "log_fatal" as verbosity levels. Thus, we further manually categorize the custom verbosity levels into five levels, i.e., *verbose*, *debug*, *info*, *warn* and *error*, as suggested in the Android official document (Developer 2017). Note that although the Android default logging library has the *assert* level (when calling the method *Log.wtf()*), it is not a typical logging level that is used in other libraries. Especially considering the *assert* level will introduce bias to our results. As the Android document mentions (Android 2017), when using the *assert* level by calling *Log.wtf()* (refers to "What a Terrible Failure"), it means that the program runs into a condition which should never happen and may (depending on the system configuration) send an error report and terminate the program immediately with an error dialog. Therefore, we categorize the logging statements of the *assert* level into the *error* level. The detailed categorization can be found in our replication package.
- **The maintenance activities of logging statements.** Adding logging statements into the source code may indicate that developers acknowledge the usefulness of logs. Well maintained logging statements with modifications may indicate that they are up-to-date and leveraged in practice. Deletions of logging statements may indicate that the logs are leveraged on a temporary basis. Such information is valuable for us to understand how logs are used by mobile app developers.

[5] https://github.com/AlDanial/cloc

We first conduct a quantitative study on the maintenance activities of logging statements. We analyze the development history of each app and identify the addition and deletion of logging statements. In addition, we track each logging statement and identify the modifications to each logging statement (e.g., adding a variable into a logging statement) similar to a prior research (Kabinna et al. 2016b). We do not consider changing the white spaces in the logging statements as modifying logging statements. We use the added, deleted and modified logging statements in each commit as log churn.

Some mobile apps may undergo more development activities than others. The logging statements in such apps intuitively may be added, deleted or modified more often than others. Therefore, we also measure the code churn rate of each mobile apps. In particular, for each commit of a mobile app, we use git diff to calculate the total code churn of the commit. Then, we calculate code churn rate of one commit as $\frac{CodeChurn}{SLOC}$ (Nagappan and Ball 2005). Likewise, we calculate log churn rate as $\frac{LogChurn}{NL}$.

Finally, we focus on the modifications of logging statements, since modification on the logging statements is an indicator that the logging statements are still leveraged by developers. Prior studies present different types of modifications to logging statements (Yuan et al. 2012a; Chen and Jiang 2017). We study the distribution of those types in mobile apps.

We study the three above-mentioned characteristics of log maintenance on all 1,444 F-Droid apps as a whole to understand the general phenomenon of logging practices in F-Droid.

3.1.3 Results

We present the results based on all of the 1,444 F-Droid apps. In addition, we discuss the similarities and differences between our findings and the findings of prior studies on server and desktop applications. The comparison is summarized in Table 3 with respect to different items. We describe each item as below:

- **LOC per log** measures the lines of code per logging statement of the studied projects. It can be interpreted as the proxy of the pervasiveness of logging practices.
- **Major logging level** describes the most common logging level that developers use when inserting a logging statement. It can reflect the severity of an event or purpose of a logging statement to an extent.
- **Log churn / code churn** is the value of log churn divided by code churn. It measures the relative maintenance effort between logging statements and source codes.
- **Log deletion** is the value of number of deleted log statements divided by the total number of log changes in a software's evolution history. It measures how often log statements get deleted during software evolution.
- **Commits with log changes** is the ratio of code commits which contain changes to logging statements. It measures how often developers commit log changes to the source code.
- **Log level changes** is the ratio of log changes which are related to the log level modification. It examines how often developers reconsider the severity of an event or purpose of a logging statement.
- **Error-level-related log changes** examines deeper to specifically study the logging statements which change log level between error and non-error levels. The item investigates the developers' reconsideration about whether an event is an error or not.

– **Variable-related log changes** measures the percentage of log changes which add, delete or modify the variables. It examines developers' considerations with respect to dynamic information.

Finding 1: Logging is commonly used in mobile apps but less pervasive than server and desktop applications.

We find that 88.6% (1,280) of the studied F-Droid apps have at least one logging statement in their source code. This shows that logging is still a common practice in mobile app development. For the apps that contain logging statements, the mean LOC per log is 479, with the values ranging from 11 to 22,144 (Table 2). With the same measurement, the average LOC per log is 30 for four C++ applications (Yuan et al. 2012a), 51 for 21 Java applications (Chen and Jiang 2017) and 58 for four C# applications (Zhu et al. 2015), as found in prior studies on server and desktop applications. Such results show that logging is a less pervasive practice in mobile app development than server and desktop application development (Table 3).

Finding 2: The majority of the logging statements in mobile apps are in debug and error levels, while info level logging statements are the majority in server and desktop applications.

Table 4 shows the logging level distribution for all 1,444 F-Droid apps. More than half of the logging statements in F-Droid apps are in *debug* and *error* levels. The distribution implies that developers often leverage logs to debug and record runtime errors of mobile apps. Such a distribution is considerably different from prior findings on server software (Li et al. 2017a), where most logging statements have the *info* level.

Finding 3: Logging statements in mobile apps are less maintained compared to server and desktop applications but are more often to be deleted.

For all F-Droid apps, the mean log churn rate (7.5%) is 0.7 times the churn rate of the entire code (11%). Only around 10% (95,306 out of 951,023) of the commits contain addition, deletion or modification of logging statements. Compared to the findings of a prior study (Yuan et al. 2012a), where the log churn rate is 1.8 times of the entire code churn rate and 18% of the commits contains addition, deletion or modification of logging statements, the results show that mobile developers do not maintain logging statements as frequent as server and desktop application developers.

Table 2 Distribution of F-Droid apps with logging statements

Metric	Mean	Min	25th quartile	Median	75th quartile	Max
NL	85	1	10	31	90	1,734
LOC per log	479	11	71	145	373	22,144
Churn rate	11%	0.03%	3%	6.4%	13.6%	315%
Log churn rate	7.5%	0.03%	1.8%	4%	8.8%	253%

Table 3 Summarizing the comparison between previous studies on server/desktop applications and this study

Items	Notes	In Java mobile apps, from this paper	In C/C++ server applications, from Yuan et al. (2012a)	In Java desktop applications, from Chen and Jiang (2017)	In C# applications, from Zhu et al. (2015)	In Java server applications, from Li et al. (2017a)
LOC per log	Logging practices in mobile apps are less pervasive than server and desktop applications.	average 479, median 145	average 30	average 51	average 58	
Major logging level	Mobile developers mainly use logs for debugging and anomaly detection purposes.	debug(34%) and error(27%)				debug(24% to 47%) and info(11% to 35%)
Log churn/ code churn	Logs are less maintained in mobile apps when compared to server and desktop applications.	average 0.68, median 0.6	average 1.8	average 1.8		
Log deletion	Compared to server applications, mobile logs are more often to be deleted.	32%	2%	26%		
Commits with log changes	Mobile developers change logs less often than server and desktop developers.	10%	18%	21%		
Log level changes	The logging level in mobile apps are more stable than server and desktop applications.	10.8%	26%	21%		
Error-level-related log changes	The log level changes that include at least one error event in mobile apps are less than C/C++ server applications but more than Java server and desktop applications.	40.9%	72%	20%		
Variable-related log changes	Similar to the previous studies, the majority of mobile app log changes are related to variables, yet the proportion is higher in mobile apps.	45.6%	27%	18%		

Table 4 Logging level distribution of F-Droid apps

Verbose	Debug	Info	Warn	Error	Total
7,296 (6.8%)	36,351 (34%)	24,010 (22.4%)	10,456 (9.8%)	28,917 (27%)	107,030

On the other hand, as Table 5 shows, deleting logging statement accounts for 32.1% of all changes to logging statements. The percentage of deleting logging statement is much higher than the results of the prior study (only 2% for deletion) (Yuan et al. 2012a).

Finding 4: Text and variables of logging statements in mobile apps are modified more often. Comparing with server and desktop applications, verbosity levels are modified less often.

Table 6 shows the distribution of the types of log modification. Note that since different types of modification can happen together in one commit, the sum of all values may be greater than one. SIM refers to Method Invocation that has String as return type (e.g., *"Log.d(TAG, user.getUserName());"*) as defined by a prior research (Chen and Jiang 2017). Among the 64,456 log modifications, 11,171 (17.3%) of them only change the whitespace format, without changing the communicated information of the logs. This may be due to automated refactoring tools in IDEs (e.g., Android Studio, the Android recommended IDE) change code indentation automatically when code changes, or developers change it manually.

Verbosity Level Modification 10.8% of the total logging statement modification is modifying the verbosity levels. A prior study shows that practitioners have a hard time choosing levels and changes between two consecutive levels are common (e.g., between *warn* and *error*) (Li et al. 2017a). However, such a one-level mistake may result less impact than a mistake that is across highest to the lowest levels or vice versa. Therefore, we further focus on error and non-error logging level (see Table 7) since as studied in prior research (Yuan et al. 2012a), this may indicate that the misjudged criticality of the logged event by developers at the first place. 40.9% of the verbosity level modifications are between error level and non-error level, showing that developers might misunderstand the critical impact of an event at the first place. Compared to our results, the results of a prior study for server applications (Yuan et al. 2012a) have a much higher verbosity level modifications (26%) and error-level-related modifications (72%). On the other hand, a recent study (Chen and Jiang 2017) on Java applications has slightly closer results to ours with 21% verbosity level modifications and 20% error-level-related modifications. However, even if the logging verbosity level is stable, it does not guarantee that the level is correct (cf. RQ2). Further in-depth studies are needed to understand the rationale of updating verbosity levels in different subjects.

Table 5 Number of logging statements changes in F-Droid apps

Added	Deleted	Modified
287,362(55.5%)	165,915(32.1%)	64,456(12.4%)

Table 6 Distributions of the different types of log modification

# Log modification	Whitespace format	Logging library	Verbosity	Text	Variable	SIM
64,456	11,171 (17.3%)	10,193 (15.8%)	6,954 (10.8%)	25,320 (39.3%)	29,363 (45.6%)	9,716 (15.1%)

SIM refers to String Invocation Method (Chen and Jiang 2017)

Text-Related Log Modifications The static text is often (39.3%) changed in log modifications. Such a finding is similar to the prior studies (45% for C/C++ applications in the study by Yuan et al. 2012a, 44% for Java server and desktop applications in the study by Chen and Jiang et al. 2017). The finding implies that changing static text in logging statements is a common practice among developers of server, desktop and mobile applications. However, few prior studies aim to assist in suggesting static text in logging statements, except for the recent study by Pinjia et al. (2018). Yet, the recent study also illustrates the challenges of generating text in logging statements. Further studies that assist in suggesting the static text in logging statements can be helpful for logging decisions.

Variable-Related Log Modifications Variable changes account for the majority (45.6%) of all log modifications. Compared to server applications, which have 27% variable-related log modifications (Yuan et al. 2012a), mobile apps developers change variables in the logging statements more often. Our results motivate the need for further study for researchers and practitioners to find the most relevant variables to support logging decisions.

Logging Library Modifications 15.8% of the logging statement modifications are related to changing the logging library. For example, in *Quran*,[6] as developer described in the commit message: *"Switch to Timber instead of Log."*, one of the logging statements changed from *"Log.d(TAG, "got cursor of data")"* to *"Timber.d("got cursor of data")"*. The developer replaced Android default logging library with *Timber*[7] to print log messages. Although Android provides a default logging library for logging, we find that developers still have concerns about the logging library. We further study from and to which logging libraries do developers switch. Among all (10,193) these logging library modifications, 68.9% (7,025) of them change from Android default logging library or Java standard output library (e.g., *System.out.println*) to a third-party logging library (e.g., *Timber*). 23% (2,342) of them change between third-party libraries. We look into the source code of these third-party logging libraries, and find that they provide extra features which are not provided by the default library. Examples of the extra features include writing logs to files, logging level printing control, and supporting richer output format. For example, in an example from *Quran*,[8] developers clearly discuss the features that the *Timber* logging library provides and explain the benefits of changing to the *Timber* logging library. Only 8.1% (826) of the logging library modifications change from third-party libraries to the default logging library. The logging library modification may be due to the need of more advanced features in logging libraries, which motivates further development of logging libraries that are optimized for mobile apps.

[6] https://github.com/ahmedre/quran_android/commit/70991626d4cab1542a0e2069d69e752ed2828bea
[7] https://github.com/JakeWharton/timber
[8] https://github.com/ahmedre/quran_android/commit/ff00a294aab44c0b995edced93e1e16d1f3ff086

Table 7 Number of error and non-error logging level modifications

Error related			Between non-error	Total
Error to non-error	Non-error to error	Total		
1,740	1,107	2,847 (40.9%)	4,107 (59.1%)	6,954

In order to further support developers who may use logging statements during mobile app development, we aim to find out the rationale of leveraging logs in mobile development by performing qualitative studies on the logging statements in the next research question.

3.2 RQ2: What are the Rationales of Mobile Logging?

3.2.1 Motivation

In RQ1, we find a discrepancy between the logging practices of mobile apps and the logging practices of server and desktop applications as studied in prior research (Yuan et al. 2012a; Chen and Jiang 2017). In order to further understand the cause of such a discrepancy, we would like to understand the rationale of having these logging statements in mobile apps. The particular rationale of using logs in mobile apps may shed light on why there exist such discrepancies. The rationale of mobile app logging can guide researchers and practitioners in designing optimal logging infrastructures especially for mobile apps.

3.2.2 Approach

We follow a two-step qualitative study to investigate the rationale of mobile logging. First, we conduct a "firehouse email interview" (Murphy-Hill et al. 2015) with the developers who recently added logging statements into the source code. We complement the email interview with our qualitative annotation on the logging rationales as the second step.

Firehouse Email Interview First, we weekly monitor the F-Droid apps to identify whether there are newly added logging statements. An email is sent out when we observed a newly added logging statement. We assume developers have a clear memory of the most recently added logging statement. In addition, since the most recently added logging statement indicates the relatively most recent stage of the project, developers would have an understanding of the apps at the stage. Therefore, by considering the most recently added logging statement, developers are clearer about the rationale and hence more likely to respond to our query. In other words, we can increase the chance to get more responses with higher quality (Murphy-Hill et al. 2015).

In order to avoid putting extra overhead to the practitioners, we do not inquiry multiple questions to the developers of the same app during each time we identify recently added logs. If there exist multiple newly added logging statements in the same app, we randomly chose one for an email interview.

With the identified recently-added logging statements, we ask their authors (i.e., developers who made the change) about the rationale of adding these logging statements by sending emails. Since there might be many logging statements that are added in one project, to avoid making a burden for developers to answer, we select only one logging statement from each project. In particular, we send an email to the developer. In each email, we describe the

identified added logging statement and provide a GitHub URL for the commit where the logging statement is included. We ask developers the following question:

"*Why did you add the logging statement in this situation?*"

After the developers reply with their answers, we make the interpretation based on their answers. The first author follow an iterative approach by starting to put the rationale of logging statements into categories. If there exists a new category, the first author starts the categorization again, until there is no new category.

Qualitative Annotation Second, we perform a qualitative annotation to examine the rationale of mobile logging. From all the added logging statements in all the studied F-Droid apps, we randomly sampled 384 logging statements, corresponding to a 95% of confidence level with a 5% confidence interval. We examine the logging statement itself, the comment and code related to the log (e.g. variables, operation logic, etc), the commit message, and the issue report if it is mentioned in the commit messages, to determine the rationale why developers added the logging statement. The source code and code comment that are associated with the logging statements help us better understand the context of the logging statement. The commit messages and the issue reports provide us with the intention of the code change in each commit.

Because we analyze the rationale without asking the opinion of the developers who actually performed the logging, the interpretation of the motivation can be considered subjective and biased by the opinions and perspectives of the authors. To mitigate the bias, the first and the third authors of this paper study the logging code independently. If there exists any disagreement, a discussion is held with the second author in order to come to a consensus. The operation is iterated until the final consensus is established with no new types of rationale is identified.

After we categorize the rationale of logging, we further inspect whether developers use consistent logging level to print log messages. Specifically, we examine whether the logging level agrees with the rationale. For example, if a logging statement is for debugging purposes, while the logging level is *error*, the misleading logging level may become a burden for log analysis. In addition, even with a mobile logging library (e.g., Logger[9]) that controls the output of logs using verbosity level, such redundant logs will still be generated during runtime of the apps. On the other hand, if a logging statement is for anomaly detection purposes, while the logging level is *debug*, the valuable information may be lost. In RQ1, we find that only 10.8% of logging statement modifications are related to logging levels, yet, the logging levels are still often inconsistent.

3.2.3 Results

> *Finding 5: We identify eight different rationales of adding logging statements in mobile apps. The majority of the logging statements are for Debug and Anomaly detection purposes.*

In the firehouse email interview, we sent out 189 emails and received 63 response during the study period (145 days from Feb 27th to July 21st, 2018), achieving a response rate of 33.3%. Each response corresponds to a distinct logging statement, commit, developer and

[9] https://github.com/orhanobut/logger

Table 8 Rationales of logging statements

Rationale	Firehouse email interview		Qualitative annotation	
	# Instances	Percentage	# Instances	Percentage
Anomaly detection	14	22.2%	107	27.8%
Assist in development	1	1.6%	11	2.9%
Bookkeeping	2	3.2%	68	17.7%
Debug	41	65%	180	46.9%
Performance	1	1.6%	7	1.8%
Change for consistency	4	6.3%	0	0%
Customize logging library	0	0%	5	1.3%
From third-party library	0	0%	3	0.8%
False positive	0	0%	3	0.8%

project. The achieved response rate is higher than a typical 5% rate found in questionnaire-based software engineering surveys (Shull et al. 2007). This can be due to the nature of firehouse email interview (Murphy-Hill et al. 2015) and our approach of selecting the logging statements. Namely, developers provide their response shortly after adding a logging and developers may still have fresh memory regarding the rationale.

After analyzing the rationale of adding logging statements based on developers' responses, we find that the rationales can be classified into six categories: *Debug*, *Anomaly detection*, *Bookkeeping*, *Assist in Development Performance* and *Change for consistency*. Table 8 presents the distribution of these rationales.

In addition, we also find the rationale of logging statements from our qualitative annotation. Besides the rationales that are observed from developers' responses, we also find two more categories, i.e., *Customize logging library* and *From third-party library*.

Below, we discuss the rationales of logging statements that are observed by our study.

Debug The majority of the logging statements are for debugging purposes. Developers mainly use logging statements to help locate the bugs. Typically, developers print the variables (e.g., URL, file path, etc), the stack trace, or a string that indicates the runtime stage in the logging statements.

- **An example from qualitative annotation**. Developers of *Delta Wallet*[10] inserted a logging statement *"Log.d(TAG, "[dispatchKeyEvent] returning " + result)"* to trace the execution path and return value in order to check if menuKeyLongPress event is triggered in order to fix a bug. Besides variables that developers define in their apps, some device-related information is also logged, such as touch point offset, GPS coordination, and accelerometer data.
- **An example from email interview**. A developer described the rationale of the logging statement *Log.w("setNumberPickerTextCol", e)* in *Nextcloud News Reader*[11] as:

  *"I added this log statement for **debugging purposes only**. There was no "good" way to set the text color of the number picker widget in android. So I had to use reflection to access the corresponding attributes. By using reflection, several exceptions might be*

[10] https://github.com/HashEngineering/dash-wallet/commit/2a96b0d3799a3208e2642905f110a344b5a2d7a3
[11] https://github.com/owncloud/News-Android-App/commit/a768e55207e11d44d27caf27ea9178f3bd37db62

thrown. That's why I put the logging statement there. Since changing the color of the Number Picker is not required anymore, the corresponding logging statement has been removed."

Anomaly Detection We found logging statements from the both firehouse email interview and qualitative annotation, that are added to detect anomaly. Developers use logging statements to record the unexpected exceptions during runtime.

- **An example from qualitative annotation.** When dealing with bitmap, *BlackLight*[12] developers recorded *"Log.w(TAG, "Unsupported EXIF orientation: " + orientationAttr)"* to detect anomaly when there is an unsupported EXIF orientation.
- **An example from email interview.** A developer described the rationale of inserting logging statement *Log.e(TAG, "Error opening category")* in *CityZen*[13] as :

  "I replaced an e.printStackTace() statement with a proper log statement, because using printStackTrace in Android apps is an anti-pattern. Simply for proper exception handling / system design: At the point within the code where ***an exception is handled*** *(not re-thrown or substituted/wrapper by another exception) it should also be logged in _any_ case because if you later have to analyse bug/issues and the only thing you have gotten from the reporters the log file you'll want to see this information to understand what happened."*

  When developers cannot reproduce a problem, they print the relevant log messages and save the logs into a log file. The log file can be uploaded and sent to developers for further analysis. Unlike server applications, mobile apps are installed on users' phones, so it is not easy to get the logs from users' phones to perform anomaly analysis. Further research should investigate the logging practices to help detect anomalies in mobile apps.

Bookkeeping Only a small number of the logging statement changes are to help developers better understand the runtime behaviour of mobile apps.

- **An example from qualitative annotation.** In the app *SysLog*, developers used *"Log.d("Loading settings")"*[14] to record the app is loading settings at runtime.
- **An example from email interview.** A logging statement *appendLog(context, TAG, "request to nominatim in less than 1.4s - nextAlowedRequestTimestamp=" + nextAlowedRequestTimestamp + ", now=" + now);* is added into *Your local weather*,[15] where the developer described the rationale as below:

  "The reason for this particular log message is to ***inform a developer*** *that the request has been thrown away because of frequency of requests."*

  In this situation, the developer uses the logging statement to notify that a request would be dropped when there are too many requests.

Assist in Development Only a very small number of logging statements are added to assist developers in developing the mobile apps.

[12] https://github.com/PaperAirplane-Dev-Team/BlackLight/commit/b0ab5187f83c6688facd4acbae803579336e9397

[13] https://github.com/CityZenApp/Android-Development/commit/7a8a6d980e404b5b5a727bca103788670c14db13

[14] https://github.com/Tortel/SysLog/commit/8f5e4b7e6461f1d714c119e2bb33be555258c72c/

[15] https://github.com/thuryn/your-local-weather/commit/6e7be714f500071c0bfc368b0352c92782325ccd

– **An example from email interview.** A developer describes the rationale of logging statement *Logger.debug(this, "Moving receipt from position to position ", realFromPosition, realToPosition))* in *Smart Receipts*,[16] as below:

"This is a part of code for adapter that matches RecyclerView which shows to the user all his receipts. This RecyclerView presents two types of items: receipt and subheader with a date. Also, this RecyclerView supports drag&drop gesture that allows reordering receipts manually (just receipt items must be draggable, subheaders must be not). So this logs was very ***helpful during the development process*** *to fixate the fact of reordering and clearly see which receipt was moved to which position."*

The developer used logging statement to print the variable value at runtime to assist the development process. In addition, developers print the log messages which show the method's name to investigate the runtime execution of the applications. The rationale indicates that logging statements provide valuable information for the ease of app development.

Performance Logging statement are added to help in the performance measurement.

– **An example from email interview.** The rationale of a logging statement *Log.i("NimbleDroidV1" "Scenario.begin " + BuildConfig.FLAVOR + "_" + session.getUrl().getValue() + "_load")* in *Firefox Focus*,[17] is described as below:

"The log statement was added to use NimbleDroid (www.nimbledroid.com) ***performance measurement*** *solution. This log would enable us to measure the time taken to load a webpage on each commit to master, and warn us when a* ***performance regression*** *is detected."*

The developers used a third-party library called *NimbleDroid* to measure the performance, and they printed the log messages to measure the time taken to check whether there is a performance regression. The response from the developers shows the performance concern of the mobile app and the value of logging to assist in performance analysis.

Change for Consistency There exist logging statements added just to follow the existing logging practices rather than an explicit intention.

– **An example from email interview.** One of the developers briefly say *"****Following existing code format****."* about the logging statement *Timber.i("NoteEditor:: Reposition button pressed")* in *AnkiDroid*.[18]

Customize Logging Library We find that logging statement changes may be due to the use of the customized logging libraries to provide extra logging support. Instead of using Android default logging library or importing a third-party library, in some apps, developers build their own customized logging library based on wrapping the Android default logging library.

[16] https://github.com/wbaumann/SmartReceiptsLibrary/commit/c0676d4c5ed01e3e86e53ce52a9183a3c33bacb1

[17] https://github.com/mozilla-mobile/focus-android/commit/301818afa33216d9a6421c11908813331a468c9c

[18] https://github.com/ankidroid/Anki-Android/commit/161ef99ab324eb427bc92586e524e42f68886284

```
/* project: https://github.com/whirish/Tri-Valley-Buses
 * commit id: db63698e25b263e9c57f649da555f41814088f52
 * file: platforms/android/CordovaLib/src/org/apache/cordova/LOG.
     java
 */

public static void v(String tag, String s) {
    if (LOG.VERBOSE >= LOGLEVEL) Log.v(tag, s);
}
```

(a) Add logging level control.

```
/* project: https://github.com/tejado/Authorizer
 * commit id: 68ed954d9965f4ec594a01d06a0be7e26f1fdb82
 * file: lib-owncloud/src/main/java/com/owncloud/android/lib/
     common/utils/Log_OC.java
 */

public static void e(String TAG, String message){
    Log.e(TAG, message);
    appendLog(TAG + " : " + message);
}
```

(b) Save logs to files.

Fig. 3 Code examples of customized logging library

– **An example from qualitative annotation.** In *Tri-Valley-Buses*,[19] the developer added logging level control to determine whether a log should be printed to complement the default logging library (see Fig. 3a). In *Authorizer*,[20] the developer added the extra logging support that saves logs to files (see Fig. 3b). Both logging statements do not have any particular rationale, but act as a general wrapper for the default logging library. This indicates that Android default logging library does not provide all the features that developers need, while third-party logging libraries might introduce extra dependency and increase the maintenance effort which developers would like to avoid.

From Third-Party Library We find logging statements that come from third-party libraries. In some apps, developers directly copy the source code from somewhere else, instead of referencing to a complied library. Such copied code may include some logging statements.

Finding 6: Developers often choose logging levels that are inconsistent to their rationale.

We find that on one hand, 45 (11.7%) of the logging statements use *verbose* or *debug* level for anomaly detection or bookkeeping purposes, which may mask important information. On the other hand, 91 (23.7%) of the logging statements use *info*, *warn*, *error* level for

[19] https://github.com/whirish/Tri-Valley-Buses/blob/db63698e25b263e9c57f649da555f41814088f52/platforms/android/CordovaLib/src/org/apache/cordova/LOG.java

[20] https://github.com/tejado/Authorizer/blob/68ed954d9965f4ec594a01d06a0be7e26f1fdb82/lib-owncloud/src/main/java/com/owncloud/android/lib/common/utils/Log_OC.java

Table 9 An overview of the eight subject apps

App	# Commits	NL	SLOC	LOC per log	# Files	# Authors	Development history
OsmAnd	44,290	841	276,677	329	5,947	771	(2017-12-09, 2010-04-25)
WordPress	25,932	934	103,009	110	1,855	101	(2017-11-23, 2009-09-10)
c:geo	10,975	608	77,992	128	2,927	123	(2017-12-02, 2011-07-11)
Nextcloud	9,227	780	52,715	68	1,261	108	(2017-12-12, 2011-08-19)
AnkiDroid	8,666	572	50,049	87	1,781	172	(2017-12-12, 2009-06-03)
K-9 Mail	7,438	701	92,797	132	1,565	238	(2017-11-27, 2008-10-27)
OpenKeychain	6,580	564	75,541	134	2,672	112	(2017-12-07, 2010-03-28)
AntennaPod	4,015	802	46,856	58	1,255	93	(2017-10-24, 2011-12-23)

debugging purposes, producing fine-grained but redundant information that should not be logged in an app's released version. Such inconsistent logging levels may cause the exposure of unnecessary information, leading to performance overhead. Therefore, developers should consider ensuring the consistency between logging levels and their rationale. In addition, this result motivates the leverage of automated techniques in suggesting appropriate logging levels (Hassani et al. 2018; Li et al. 2017a).

3.3 RQ3: How Large Can the Performance Impact of Mobile Logging Be?

3.3.1 Motivation

Our previous research question shows that 23.7% of logging statements produce unnecessary information such as the debugging and tracing information with high verbosity levels (see RQ2). Such unnecessary information is not expected in app's released version. The unnecessary logging information cannot be simply ignored since they consume extra system resources (e.g. CPU and battery), and it may bring significant performance impact to the apps (Chowdhury et al. 2017). For example, in *Mandelbrot Maps*,[21] the developer removed logging statements as the commit message indicated: *"Also removed logging that was happening every time the pin moved at all, which made it a fair bit less laggy"*. Therefore, in this research question, we would like to examine what the performance impact of such unnecessary logging can have in Android apps.

3.3.2 Approach

We conduct our case study on eight selected apps based on the following criteria:

1. *Well maintained:* the apps have the highest number of commits.
2. *With logging practices:* the apps contain more than 500 logging statements.
3. *With log-related test files:* the apps have test files which produce logs output, in this way we can conduct the experiment on logging performance overhead.

Specifically, we first sort the F-Droid apps in descending order of their number of commits, then select eight apps which satisfy the last two criteria described above. Table 9

[21] https://github.com/withad/Mandelbrot-Maps-on-Android/commit/2bf86fc239ea063950ebc8039ba8084b04a6bad1

presents an overview of these eight apps. All of the eight apps are actively maintained with six to nine years of development history, and contain a considerable number of logging statements, with the lines of code per logging statement ranging from 58 to 329.

We study the eight selected Android apps to examine the performance impact of unnecessary logging. Similar to prior research (Chen and Shang 2017; Chen et al. 2014, 2016), we leverage the tests of these apps to exercise the apps and measure the performance of the apps during test execution. We first identify all the logging statements that will be executed for each test of each app. We then identify the unnecessary logs, i.e., the logging statements that generate unexpected debugging information in a released version (with a high logging level). We look into each logging statement that gets called during test execution and determine the rationale behind the code. Unnecessary logging statements with purposes such as *Debugging*, *Assist development* and *Performance* are commented out, while keeping logging statements for *Anomaly detection* and *Bookkeeping*.

For each test, we measure the performance with all the logging enabled of the released version of the apps; disabling all logging statements in the apps and enabling only the necessary logs. We use both physical level performance metrics, i.e., CPU and battery consumption, and domain level performance metrics, i.e., response time, as measurements of performance impact. In order to minimize the impact of performance monitoring itself, we do not monitor all performance metrics at the same time. We redo all our experiments with only measuring *one* performance metric at a time. Since the frequency of logging may serve as an indicator of performance overhead, we also measure the frequency of logs generated in each test.

First, we select tests for our experiments. Since we would like to examine the performance impact of unnecessary logging, tests that do not generate any logs are irrelevant. We run each test of these selected apps and check if there is any log generated. The tests that do not generate any logs are not selected for our experiments. Furthermore, in order to reduce the environmental noise when performing these experiments, we first run all tests once and monitor their network access. With the monitoring data, we do not select the tests which would send network requests to eliminate the negative effect of network fluctuation. We identify both *local unit tests* and *instrumented tests* in our experiments. Finally, since these tests may not reflect the usage of end-users using the apps, the first author of the paper uses the app as an end user while being monitored by the *Espresso* test automation framework.[22] Based on the recorded monitoring data, we created *Espresso tests*.

Local Unit Tests The local unit tests are the ones that can run with a unit testing framework like JUnit[23] and can run with little or no dependencies on Android system environment. Thus, we execute the local unit tests on a Linux machine (4-core Intel i5-4690 CPU, 8GB memory with Debian 9). During test executions, we use a performance monitoring tool named *psutil*[24] to collect the performance metrics of the test running processes.

Instrumented Tests The instrumented test can take the advantage of the Android framework and supporting APIs. Therefore, such tests run on physical devices or emulators. We

[22] https://developer.android.com/training/testing/espresso/

[23] https://junit.org/junit5/

[24] https://github.com/giampaolo/psutil

conduct the instrumented tests on a physical Android device, i.e., a Google Pixel 2 XL cellphone (Android 8.1.0). We target at the application process and inspect the response time, CPU usage and battery consumption during test executions. The CPU usage is extracted by inspecting the */proc/stat* (for device) and */proc/[pid]/stat* (for process) files.[25] Android's kernel is derived from Linux thus we can leverage the same technique to evaluate the CPU usage. We do not use Android *dumpsys cpuinfo*[26] tool to perform the data collection since it gets the information from */proc/stat* and experiences delay. To get a precise result, we manually make each test wait for 15 seconds before and after execution, while the inspection tool starts collecting data at the first 10 seconds and stops at the last 10 seconds. In this way, we can capture the precise CPU usage that the test consumes. For battery consumption, we use Android *dumpsys batterystats* tool to extract the usage for every specific application. The battery consumption is estimated in milliampere-hour (mAh). Before running each test, we reset the battery data, then we record the battery consumption after each test finishes.

Espresso Tests In order to generate *Espresso tests*, the first author performs interactions with the mobile apps while being recorded by the *Espresso* testing framework. Afterward, the recorded actions are transformed into testing scripts. Since some apps, such as Nextcloud, require internet connections during runtime, we pre-fetch the content from the internet and save the data in the testing device to avoid the noise from the actual internet connection. The *Espresso tests* are also conducted on the same physical device as the instrumented tests with the same performance monitoring infrastructure. Our generated *Espresso tests* are included in our replication package.

Noise and uncertainty always exist when measuring performance (Mytkowicz et al. 2009a). For example, given the same workload, an application can consume different CPU usage in two different executions. To mitigate the impact of noise, we adopted the approach of repeated measurement as used in prior studies (Chen and Shang 2017; Chen et al. 2014). In particular, we repeat the performance evaluation 30 times independently in each round.

Since some tests finish in a very short period of time, (e.g., 15ms for Nextcloud's unit tests), there is no enough time for our tool to collect system resources' consumption details. To resolve this issue, we manually add an iteration loop for these test code to make them run longer (around one minute) to enable our tool to collect data. Because our goal is to compare the system resources' consumption between enabling logging and disabling logging, and test gets run the same times in the comparison, the modification of code would not change the results.

With the performance evaluation data, we perform a statistically rigorous approach to measure the performance impact. Specifically, we use the Wilcoxon Rank Sum test (Moore et al. 2012) to check whether there exist statistically significant differences in system performance between: 1) logging enabled and disabled; 2) logging enabled and removing unnecessary logging. If there exists statistically significant difference ($p \leqslant 0.05$), we use Cliff's δ (1993) to calculate the effect sizes of such differences. Cliff's δ ranges from -1 to $+1$, where a 0 value indicates two identical distributions. A positive value indicates that

[25] http://man7.org/linux/man-pages/man5/proc.5.html
[26] https://developer.android.com/studio/command-line/dumpsys

values in the first sample tend to be greater than those in the second sample, while a negative value indicates the opposite. The strength of the effects and the corresponding range of Cliff's δ values are defined as follows (Romano et al. 2006):

$$effect\ size = \begin{cases} trivial & \text{if } Cliff's\ \delta \leqslant 0.147 \\ small & \text{if } 0.147 < Cliff's\ \delta \leqslant 0.33 \\ medium & \text{if } 0.33 < Cliff's\ \delta \leqslant 0.474 \\ large & \text{if } 0.474 < Cliff's\ \delta \leqslant 1 \end{cases}$$

3.3.3 Results

Tables 10, 11 and 12 shows our experimental results on the performance impact of generating logs in mobile apps. Apps that do not have corresponding tests are not presented in the tables.

> *Finding 7: There can be a statistically significant performance overhead when generating logs in mobile apps, especially for end-user impacting measurements such as response time and battery consumption.*

We find that there can be a statistically significant performance overhead, with large effect sizes, in terms of response time and battery consumption when enabling logging in mobile apps (see Tables 10, 11 and 12). The maximum performance overhead is observed in the instrumented test #1 in K-9 Mail, where the response time and battery consumption with logging enabled is around three times of that without logging. Response time and battery consumption are two important performance measures that can directly impact end-users. Therefore, such finding demonstrates the importance of making optimal logging decisions in order to improve the user experience of mobile apps.

On the other hand, the performance overhead in other physical metrics is not conclusive. When we examine the CPU cycles and CPU percentage of the apps, although many of the differences are statistically significant, some of the effect sizes are trivial. Such finding shows that the CPU may not be the bottleneck of producing logs. Since the CPU is idle when producing logs, the CPU percentage is often statistically significantly lower with logging enabled. However, this lower CPU percentage is not a positive phenomenon since it shows that the CPU is blocked and cannot contribute to providing calculation power while the response time and battery consumption are sacrificed in such a case. This finding implies the need of more advanced logging libraries for mobile apps, which may avoid blocking the CPU when producing logs as fast as possible. For example, the async logging feature is typically used in server and desktop logging libraries like Log4j2,[27] while not supported by either the Android default logging library nor the popular third-party libraries like Timber.[28]

OpenKeychain is the only app that shows almost no statistically significant performance overhead of logging in their local unit tests (see Table 10). In order to understand this result, we manually investigate the source code and local unit tests in OpenKeychain. We find that OpenKeychain leverages a special testing framework called Robolectric[29] in each of their local unit test. Each test needs to initialize the Robolectric testing environment, which makes

[27] https://logging.apache.org/log4j/2.x/
[28] https://github.com/JakeWharton/timber
[29] http://robolectric.org/

Table 10 Performance impact from local unit tests between enabling and disabling logging

Local unit tests

App	Test	Logs per second	Stage	Response time			CPU time			CPU percentage		
				Median	Median diff	Effect sizes	Median	Median diff	Effect sizes	Median	Median diff	Effect sizes
Osmand	1	4.07	With Log	15.0s	+2.0s	+large	17.6s	+3.2s	+large	24.50%	−26.73%	-large
			Without Log	13.0s			14.4s			51.23%		
	2	8.00	With Log	18.0s	+11.0s	+large	22.9s	+3.1s	+large	38.05%	+0.08%	N/A
			Without Log	7.0s			19.8s			37.97%		
Nextcloud	1	120.00	With Log	20.0s	+12.0s	+large	17.4s	+12.6s	+large	40.61%	+10.45%	+large
			Without Log	8.0s			4.8s			30.16%		
K-9 Mail	1	42.38	With Log	12.2s	+0.2s	+medium	14.3s	+0.5s	+large	70.36%	+2.67%	large
			Without Log	12.0s			13.8s			67.69%		
	2	7.12	With Log	11.1s	+0.1s	+medium	11.7s	−0.3s	N/A	71.89%	−0.56%	N/A
			Without Log	11.0s			12.0s			72.45%		
OpenKeychain	1	2.45	With Log	24.1s	−0.1s	N/A	18.4s	+0.2s	N/A	57.82%	−1.38%	N/A
			Without Log	24.2s			18.2s			59.20%		
	2	1.09	With Log	22.9s	−0.6s	N/A	18.6s	+0.1s	N/A	59.44%	+0.60%	N/A
			Without Log	23.5s			18.5s			58.84%		
	3	11.57	With Log	36.2s	−0.2s	N/A	20.9s	+0.2s	N/A	27.85%	+1.80%	+medium
			Without Log	36.4s			20.7s			26.05%		
	4	0.70	With Log	25.8s	-0.2s	N/A	18.9s	+0.5s	N/A	57.33%	+2.33%	N/A
			Without Log	26.0s			18.4s			55.00%		
	5	0.34	With Log	23.3s	−0.4s	N/A	18.8s	+0.1s	N/A	61.58%	+0.38%	N/A
			Without Log	23.7s			18.7s			61.20%		
	6	0.25	With Log	24.3s	−0.3s	N/A	19.2s	+0.2s	N/A	59.72%	+1.54%	N/A
			Without Log	24.6s			19.0s			58.18%		

N/A in effect sizes means that the difference is statistically insignificant

Table 11 Performance impact from instrumented tests between enabling and disabling logging

Instrumented tests

App	Test	Logs per second	Stage	Response time			CPU cycles			CPU percentage			Battery		
				Median	Median diff	Effect sizes	Median	Median diff	Effect sizes	Median	Median diff	Effect sizes	Median	Median diff	Effect sizes
WordPress	1	849.07	With Log	74.1s	+5.4s	+large	10,847	+1,074	+large	16.54%	−0.77%	–large	6.24	+1.13	+large
			Without Log	68.7s			9,773			17.31%			5.11		
	2	1265.30	With Log	62.3s	+2.4s	+large	8,772	–79	N/A	16.22%	−0.32%	–large	5.06	+0.84	+large
			Without Log	59.9s			8,851			16.54%			4.22		
c:geo	1	43.42	With Log	70.4s	+2.9s	+large	4,396	–25	N/A	8.93%	−0.05%	–large	1.28	+0.06	+large
			Without Log	67.5s			4,421			8.98%			1.22		
	2	59.47	With Log	70.7s	+3.6s	+large	5,545	+38	N/A	10.57%	−0.27%	–large	1.94	+0.09	+large
			Without Log	67.1s			5,507			10.84%			1.85		
	3	71.75	With Log	62.5s	+3.5s	+large	4,547	339	+large	9.68%	+0.20%	+large	1.39	+0.07	+large
			Without Log	59.0s			4,208			9.48%			1.32		
	4	61.92	With Log	67.2s	+3.8s	+large	4,515	–23	N/A	9.32%	−0.15%	–large	1.66	+0.28	+large
			Without Log	63.4s			4,538			9.47%			1.38		
	5	59.34	With Log	65.7s	+5.4s	+large	5,010	+1,253	+large	8.60%	+0.74%	+large	2.48	+0.61	+large
			Without Log	60.3s			3,757			7.86%			1.87		
	6	74.00	With Log	72.3s	+4.2s	+large	8,465	0	N/A	15.72%	+0.01%	N/A	5.23	+0.44	+large
			Without Log	68.1s			8,465			15.71%			4.79		
	7	294.61	With Log	75.6s	+5.7s	+large	9,707	+664	+large	18.40%	+0.47%	+large	5.85	+0.33	+large
			Without Log	62.9s			9,043			17.93%			5.52		
	8	137.46	With Log	61.2s	+32.5s	+large	8,710	+4,632	+large	17.17%	−0.17%	–medium	5.66	+2.79	+large
			Without Log	28.7s			4,078			17.34%			2.87		
AnkiDroid	1	4.01	With Log	53.6s	+0.6s	+medium	3,120	+104	+large	7.63%	+0.13%	+large	1.04	+0.03	+large
			Without Log	53.0s			3,016			7.50%			1.01		

Table 11 (continued)

Instrumented tests

App	Test	Logs per second	Stage	Response time			CPU cycles			CPU percentage			Battery		
				Median	Median diff	Effect sizes	Median	Median diff	Effect sizes	Median	Median diff	Effect sizes	Median	Median diff	Effect sizes
	2	0.17	With Log	57.9s	+0.7s	+medium	3,046	+28	N/A	6.99%	+0.02%	N/A	0.99	−0.01	N/A
			Without Log	57.2s			3,018			6.97%			1.00		
	3	28.23	With Log	68.4s	+2.4s	+large	4,688	+255	+large	8.39%	–0.02%	N/A	1.63	+0.09	+large
			Without Log	66.0s			4,433			8.41%			1.54		
K-9 Mail	1	1.23	With Log	66.4s	+44.3s	+large	8,506	+5,647	+large	15.99%	−0.31%	−large	5.26	+3.56	+large
			Without Log	22.1s			2,859			16.30%			1.70		
AntennaPod	1	47.79	With Log	67.1s	+2.1s	+large	7,717	+184	+large	11.99%	−0.22%	−large	3.90	+0.18	+large
			Without Log	65.0s			7,533			12.21%			3.72		
	2	27.68	With Log	73.4s	+9.6s	+large	355	+22	+large	0.69%	+0.04%	+large	0.25	+0.15	+large
			Without Log	63.8s			333			0.65%			0.10		
	3	6.03	With Log	56.7s	+0.1s	+medium	224	+64	+large	2.06%	+0.13%	N/A	0.091	+0.004	+large
			Without Log	56.6s			160			1.93%			0.087		
	4	114.66	With Log	65.3s	+3.3s	+large	4,188	+787	+large	6.75%	+0.55%	+large	1.33	+0.04	+large
			Without Log	62.0s			3,401			6.20%			1.29		
	5	109.53	With Log	63.1s	+4.3s	+large	4,156	+1,036	+large	8.47%	+0.69%	+large	1.24	+0.07	+large
			Without Log	58.8s			3,120			7.78%			1.17		
	6	129.39	With Log	68.0s	+8.3s	+large	4,591	+684	+large	8.50%	+0.30%	+large	1.39	+0.14	+large
			Without Log	59.7s			3,907			8.20%			1.25		
	7	189.27	With Log	66.6s	+5.4s	+large	4,247	+663	+large	8.56%	+0.28%	+large	1.34	+0.07	+large
			Without Log	61.2s			3,584			8.28%			1.27		
	8	165.01	With Log	62.2s	+4.5s	+large	4,302	+942	+large	9.20%	+0.72%	+large	1.39	+0.10	+large
			Without Log	57.7s			3,360			8.48%			1.29		

N/A in effect sizes means that the difference is statistically insignificant

Table 12 Performance impact from *Espresso tests* between enabling and disabling logging

Espresso tests

App	Test	Logs per second	Stage	Response time			CPU cycles			CPU percentage			Battery		
				Median	Median diff	Effect sizes	Median	Median diff	Effect sizes	Median	Median diff	Effect sizes	Median	Median diff	Effect sizes
OsmAnd	1	1.22	With Log	81.4s	0	N/A	1,486	0	N/A	2.11%	0	N/A	1.18	+0.01	N/A
			Without Log	81.4s			1,486			2.11%			1.17		
Wordpress	1	1.62	With Log	55.7s	+0.1s	+medium	823	–3	N/A	1.73%	0	N/A	0.001	0	N/A
			Without Log	55.6s			826			1.73%			0.001		
c:geo	1	1.17	With Log	76.7s	+0.2s	+medium	1,909	+257	+large	2.90%	+0.4%	+large	1.17	+0.02	+large
			Without Log	76.5s			1,652			2.50%			1.15		
Nextcloud	1	9.90	With Log	67.3s	0	N/A	1,516	+7	+medium	2.94%	0	N/A	0.75	+0.01	N/A
			Without Log	67.3s			1,509			2.94%			0.74		
AnkiDroid	1	11.43	With Log	86.8s	0	N/A	1,758	+35	+large	2.51%	+0.05%	+large	0.024	+0.001	+small
			Without Log	86.8s			1,723			2.46%			0.023		
K-9 Mail	1	0.23	With Log	104.8s	+0.1s	N/A	1,595	+1	N/A	1.88%	0	N/A	0.88	0	N/A
			Without Log	104.7s			1,594			1.88%			0.88		
OpenKeychain	1	3.18	With Log	94.3s	0	N/A	1,812	–20	N/A	2.57%	+0.09%	N/A	0.37	0	N/A
			Without Log	94.3s			1,832			2.48%			0.37		
AntennaPod	1	10.37	With Log	92.5s	–0.1s	N/A	1,885	+18	+large	2.56%	+0.03%	+large	0.67	0	N/A
			Without Log	92.6s			1,867			2.53%			0.67		
	2	52.13	With Log	75.5s	+0.1s	N/A	1,935	+19	+large	3.21%	0	N/A	0.74	0	N/A
			Without Log	75.4s			1,916			3.21%			0.74		
	3	1.32	With Log	244.0s	+0.3s	+large	2,630	+123	+large	1.36%	+0.05%	+large	0.99	+0.01	+large
			Without Log	243.7s			2,507			1.31%			0.98		

N/A in effect sizes means that the difference is statistically insignificant

the performance of each test unstable. The overhead caused by the logging statements is not statistically significant in such an environment due to the large noise.

We also find that, compared to local unit tests and instrumented tests, the performance overhead of logging is less likely to be statistically significant during *Espresso tests* (see Table 12). In particular, only three of the *Espresso tests* shows statistically significant battery consumption overhead. Such a result agrees with the findings from a prior study by Chowdhury et al. (2017), which finds that energy consumption of logging may not be significant in regular usage of mobile apps.

We cannot observe any relationship between the frequency of generated logs and whether the logging overhead is statistically significant. For example, *Espresso test* #2 in AntennaPod has the highest logging frequency while logs do not introduce any statistically significant performance overhead. On the other hand, *Espresso test* #3 in AntennaPod has a much lower logging frequency yet introduces significant performance overhead with large effect sizes in every metric. Such results show the complex nature of performance overhead from logging and motivates further in-depth studies on logging overhead in different usage scenarios.

> *Finding 8: If disabling logging statement provides significant performance improvement, the performance overhead from unnecessary logging statements generated by the current tests can be statistically significant.*

To filter out the unnecessary logs, we manually categorize the purpose of the eight selected apps' logging codes. For OsmAnd and OpenKeychain, there exists no logging code for anomaly detection or bookkeeping. Similarly, for Nextcloud, all log messages printed during tests stem from the test files. It means that all logs are unnecessary thus there is no candidate to be compared to for these three apps. For WordPress, some unnecessary logs come from code in an external jar file which we cannot exclude. The remaining apps (c:geo, AnkiDroid, K-9 Mail and AntennaPod) have both necessary and unnecessary logging code that can be optimized.

We evaluate the runtime impact of unnecessary logs of four apps (c:geo, AnkiDroid, K-9 Mail and AntennaPod) as well as all the *Espresso tests* of the eight apps. We observe many cases where that such unnecessary logs typically do not show statistically significant performance overhead (see Table 13). By carefully examine such cases, we find that there are cases that have very few unnecessary logs or undergo low performance overhead of logging. For example, on one hand, we manually examine the test in c:geo and find that the number of unnecessary logs generated by the instrumented test is low (almost 0%). Therefore, disabling the unnecessary logs or even all the logs in the instrumented test of c:geo does not improve performance significantly. On the other hand, with a more realistic workload in the *Espresso tests*, the number of generated unnecessary logs is much higher (61.1% to 99.3%). However, the performance improvement of disabling these unnecessary logs is minimized due to the observed low performance overhead of logging in *Espresso tests* (see Tables 10, 11 and 12). In other words, if even disabling all logs would not provide any performance improvement, based on the results in Tables 10, 11 and 12, we cannot expect any performance improvement by only disabling unnecessary logs. Nevertheless, it is clear to see that in the case of c:geo, AnkiDroid and AntennaPod, if performance improvement is statistically significant when disabling all the logs in the *Espresso tests* (see Table 12), the unnecessary logging overhead is also statistically significant (see Table 13).

Table 13 Performance impact between enabling all logging and disabling unnecessary logging

App	Test	% Unnecessary logs	Stage	Logs per second	Response time			CPU cycles			CPU percentage			Battery		
					Median	Median diff	Effect sizes	Median	Median diff	Effect sizes	Median	Median diff	Effect sizes	Median	Median diff	Effect sizes
Instrumented tests																
c:geo	8	~0.0%	With Log	137.46	61.2s	–0.1s	N/A	8,710	+32	N/A	17.17%	−0.02%	N/A	5.66	0	N/A
			With Necessary Log	137.46	61.3s			8,678			17.19%			5.66		
AnkiDroid	1	51.6%	With Log	4.01	53.6s	+1.1s	+large	3,120	+48	+medium	7.63%	0	N/A	1.04	+0.05	+large
			With Necessary Log	1.98	52.5s			3,072			7.63%			0.99		
	2	50.0%	With Log	0.17	57.9s	+1.3s	+large	3,046	−26	N/A	7.00%	−0.02%	N/A	0.99	+0.01	+medium
			With Necessary Log	0.09	56.6s			3,072			7.02%			0.98		
	3	89.1%	With Log	28.23	68.4s	+0.6s	+medium	4,688	+669	+large	8.39%	+0.15%	+large	1.63	+0.10	+large
			With Necessary Log	3.10	67.8s			4,019			8.24%			1.53		
K-9 Mail	1	76.8%	With Log	1.23	66.4s	+19.1s	+large	8,506	+2,229	+large	15.99%	+0.71%	+small	5.26	+1.54	+large
			With Necessary Log	0.40	47.3s			6,277			15.28%			3.72		
AntennaPod	2	95.3%	With Log	27.68	73.4s	+10.0s	+large	355	+49	+large	0.69%	+0.09%	+large	0.25	+0.16	+large
			With Necessary Log	1.58	63.4s			306			0.60%			0.09		
Espresso tests																
OsmAnd	1	84.8%	With Log	1.22	81.4s	–0.1s	N/A	1,486	0	N/A	2.11%	0	N/A	1.18	+0.01	N/A
			With Necessary Log	0.18	81.5s			1,486			2.11%			1.17		
Wordpress	1	74.4%	With Log	1.62	55.7s	0	N/A	823	+1	N/A	1.73%	0	N/A	0.001	0	N/A
			With Necessary Log	0.41	55.7s			822			1.73%			0.001		

Table 13 (continued)

App	Test	% Unnecessary logs	Stage	Logs per second	Response time			CPU cycles			CPU percentage			Battery		
					Median	Median diff	Effect sizes	Median	Median diff	Effect sizes	Median	Median diff	Effect sizes	Median	Median diff	Effect sizes
c:geo	1	61.1%	With Log	1.17	76.7s	+0.2s	+medium	1,909	+52	+large	2.90%	+0.08%	+medium	1.17	0	N/A
			With Necessary Log	0.46	76.5s			1,857			2.82%			1.17		
Nextcloud	1	83.9%	With Log	9.90	67.3s	–0.1s	N/A	1,516	+1	N/A	2.94%	+0.01%	N/A	0.75	0	N/A
			With Necessary Log	1.59	67.4s			1,515			2.93%			0.75		
AnkiDroid	1	97.9%	With Log	11.43	86.8s	0	N/A	1,758	+37	+large	2.51%	+0.04%	+large	0.024	+0.001	+small
			With Necessary Log	0.24	86.8s			1,721			2.47%			0.023		
K-9 Mail	1	66.7%	With Log	0.23	104.8s	+0.2s	N/A	1,595	–2	N/A	1.88%	0	N/A	0.88	0	N/A
			With Necessary Log	0.08	104.6s			1,597			1.88%			0.88		
OpenKeychain	1	99.3%	With Log	2.84	94.3s	+0.1s	N/A	1,812	–65	N/A	2.57%	0	N/A	0.37	–0.12	N/A
			With Necessary Log	0.02	94.2s			1,877			2.57%			0.49		
AntennaPod	1	92.8%	With Log	10.37	92.5s	+0.1s	N/A	1,885	+21	+large	2.56%	+0.03%	+large	0.67	0	N/A
			With Necessary Log	0.75	92.4s			1,864			2.53%			0.67		
	2	94.5%	With Log	52.13	75.5s	+0.1s	N/A	1,935	–4	N/A	3.21%	−0.03%	–small	0.74	0	N/A
			With Necessary Log	2.88	75.4s			1,939			3.24%			0.74		
	3	78.9%	With Log	1.32	244.0s	+1.0s	+large	2,630	+58	+large	1.36%	+0.02%	+large	0.99	+0.02	+large
			With Necessary Log	0.28	243.0s			2,572			1.34%			0.97		

N/A in effect sizes means that the difference is statistically insignificant

4 Discussion

In this section, we discuss the implications of our study results.

4.1 The Different Logging Practices Between Mobile Apps and Server/Desktop Applications (RQ1)

From our empirical study results, we observe that logging practices of mobile apps is much different from those of server and desktop applications.

There Exist Less Pervasive Logging Practices in Mobile App Development We consider the reason to be two folds. First of all, mobile apps typically have a much smaller code base (3,760 median SLOC in 1,444 F-Droid apps, 202K median SLOC in the studied C/C++ server applications (Yuan et al. 2012a), 116K median SLOC in the studied Java server and desktop applications Chen and Jiang 2017) and fewer contributors (4 median authors in 1,444 F-Droid apps, 34 median authors in the studied C/C++ server applications (Yuan et al. 2012a), 37 median authors in the studied Java server and desktop applications Chen and Jiang 2017). With less uncertainty of the application and the development activity, there may exist less needs for using logs to tackle the challenges of program comprehension. Second, although Android provides a default logging library, the logging library is not optimized for mobile app development. In particular, default Android logs can only be viewed in Logcat with the device connected to a computer, while developers cannot retrieve the logs from a disconnected mobile device (Developer 2017). Although in the study we also consider the logging statements with third-party logging libraries such as Logger[30] and Timber,[31] the naive features of the default logging library may prevent developers from leveraging logs in practice.

The Logging Statements are Less Maintained During Development; While There Exists Much More Deletion to Logging Statements Compared to Server and Desktop Applications We find that all too often, logging statements in mobile apps are only used temporarily in the source code, i.e., developers add logging statements for particular tasks and then delete the logging statements after they finish the task. For example, in *DAPNETApp*,[32] the developer removed logging statements such as *"Log.i(TAG,"saveAdmin: admin: "+admin);"* as she or he indicated in the commit message: *"Remove debug logging"*. These logging statements were added in one previous commit,[33] which aimed to fix an issue as the commit message mentioned: "This fixes #18, version bump to 1.0.2". As we examined the issue description and source code, we found that these logging statements were used to present user information and program execution status in order to fix an HTTP 403 Error when a user tried to view phone calls. After the bug was fixed, the developer removed the debug logging statements. Therefore, many logging statements are not meant to stay in the source code, hence, without the need for maintenance. Such a practice is not reported in prior studies on the logging practices in server and desktop applications.

[30] https://github.com/orhanobut/logger

[31] https://github.com/JakeWharton/timber

[32] https://github.com/DecentralizedAmateurPagingNetwork/DAPNETApp/commit/bd7427d825a02cde2584858396fa170f7dd0a44d

[33] https://github.com/DecentralizedAmateurPagingNetwork/DAPNETApp/commit/f4a84c261a910b2b9d702028275299f6a4c04663

The Majority of the Mobile App Logging Statements are in the *Debug* and *Error* Level Although Android default logging library provides different verbosity levels to print log messages, the verbosity level itself only controls whether the *verbose* level logging statements are compiled into the source code (Elye 2018). For example, a logging statement "*Log.d(TAG, "get token:" + token);*" in the *debug* level, can still generate log messages in the release build of the app. In such a case, the verbosity level itself is more about indicating the severity and purpose of the logs rather than controlling how verbose the logs are. Therefore, by considering our finding that a large portion of the logging statements are in the *debug* level, it may be the case that developers often add logging statements when they start to debug and remove the logging statement after debugging, making logging statements only exist temporarily in the development history.

Take-home message 1: Developers and software engineering researchers of mobile apps should be aware of the differences between mobile and server/desktop logging practices. Prior findings on logging practices on server/desktop applications may not hold for mobile apps.

4.2 The Rationale and the Verbosity of Mobile App Logging (RQ2)

Debug Accounts for the Majority of Logging Rationale From the results of our firehouse email interview and qualitative annotation, we find that the majority of the logging statements in mobile apps are for debugging purposes. Such debugging logging statements are meant to be temporary in the source code and should not be shipped to the users, due to performance and security reasons.

As the example from email interview that we show for the debugging logging statement (see RQ2), the developer mentioned that the logging statement for debugging has been removed after the feature is not required anymore. This indicates that developers have some concern that logging statements might have adverse impact and should be managed well. Therefore, we further examine whether developers remove the logging statements after debugging. Following the firehouse email interview, we find that out of these 41 debugging logging statements, only 10 of them actually get removed. In one case, the developer mentioned will remove it, but he/she has not already done so the last time we checked (July 26th, 2018). The rest 31 (75.6%) of them still exist in the source code. For the three of the logging statements, the developers mentioned that they use logging level to control the appearance of the logs and these logging statements would not get shipped to users. Such results indicate that developers may be aware that debugging logging statements is unnecessary, or even harmful, for the end-users of mobile apps.

There is one response of logging statement (*Log.i(TAG, "Selected file: " + filePath)* in *Video Transcoder*[34]) for which the developer explicitly mentioned that he wanted to keep the logging statement after debugging. The developer mentioned that:

"This commit adds a file picker which is embedded into the application. During development I was learning how to use this library, and the log statement allowed me to debug the application in a few test scenarios to ensure that it was working as expected. The log statement was kept (instead of being removed after debugging) for two reasons: 1) In case I did

[34] https://github.com/brarcher/video-transcoder/commit/55c22c594cba15abae60528a55e0309c00c035ca

not get the usage of the file picker library correct and a user hits a problem, the log statement may help diagnose what data the file picker is returning, and hopefully help triage the issue; and 2) This log statement represents the first data selected by the user for processing. If something went wrong later in the program, this log statement will trace back the input used by the application. It will tell valuable information such as what partition the data resides on (internal storage, SD card) and what media type the file represents (via its extension).”

From the response, we can see that there is a need for the debugging logging statement in production because it provides valuable information for further problem diagnosis. Therefore, further research can be conducted on how to balance the potential adverse impact and the valuable information that logging statements provide.

4.3 The Energy Overhead of Mobile App Logging (RQ3)

Insignificant Energy Overhead by Logging Found by a Prior Study The prior study by Chowdhury et al. (2017) evaluates the logging overhead on energy consumption in mobile apps. Chowdhury et al.’s study finds that most of the mobile apps do not have a significant energy overhead when enabling logs. We consider the reason is that our experiments are conducted with running tests, which may have less noises from internal factors such as memory management. Such noises are discussed to be one of the reasons that the logging overhead is not significant in the prior study. By examining the context of the experiments and the results, we consider that such a finding does not conflict with our finding of statistically significant performance overhead of logging. In particular, our findings on local unit tests and instrumented tests confirm the existence of logging overhead in mobile apps, since the tests are conducted in a controlled manner with less noise. On the other hand, our results with *Espresso tests* and the findings by Chowdhury et al. show that, with the noise of other factors that may impact performance, the overhead is often not noticeable on a typical workload and current logging density. However, with more logging added into the source code and heavier workload, the impact may start to be noticed in real life.

There Exists a Strong Need for Automated Logging Library and Tooling Support for Mobile Apps Our results show that logging has a significant performance impact on mobile apps. It consumes extra system resources such as battery and cost longer response time. If developers do not have the optimized logging decisions, the overhead of logging may have a large negative performance impact on end-users’ experience of mobile apps, due to the limited computing power and energy. Therefore, specially designed strategies for mobile logging are needed in such cases. In addition, some of the overhead is contributed by the suboptimal logging libraries on mobile apps, while the logging libraries for server and desktop machines are much more advanced (e.g., the async feature in Log4j2[35]). Therefore, specially designed mobile app logging libraries would help in reducing such overhead.

> *Take-home message 2: Practitioners should be aware of the energy overhead of mobile logging. In particular, energy should not be wasted on outputting information that is not needed in mobile app runtime.*

[35] https://logging.apache.org/log4j/2.x/

5 Threats to Validity

This section discusses the threats to the validity of our study.

5.1 External Validity

In our study, the selected mobile apps are all free and open source Android apps written in Java. However, compared to the vast number of Android apps, our results may not be generalizable to other, especially the non-free and closed source, Android apps. In particular, Gaming apps make a great portion in the Android apps in the Google Play Store. Such Gaming apps may have a strong need for performance improvement (Lin et al. 2018) and may benefit from learning user behaviours from logs (Harpstead et al. 2015). However, Gaming apps only take 10% of the F-Droid apps. Our study is based on the logging statements in the source code. We do not have access to the source code of closed source mobile apps. Even by decompiling the APK files, we cannot identify logging statements in the code due to code obfuscation. Collaborating with close-source mobile app developers from industry may address this challenge. In addition, we cannot claim that the performance testing results of the eight selected apps can be generalized to other apps. Future studies may provide more insights on the mobile app logging performance overhead of other types of apps to address this threat.

Our *Espresso tests* are created based on the monitoring data of the usage of the first author on the apps. The usage may not be generalizable for other end users. More *Espresso tests* that are generated by more end users can minimize this threat.

Since we only studied logging practices in Android apps that are implemented in Java, our findings may not be generalizable to apps that are implemented in other programming languages (e.g., Kotlin or Swift) and mobile operating systems (e.g., iOS). Further studies are needed on the logging practices for apps with other programming languages and in other mobile operating systems.

5.2 Internal Validity

In the data gathering process to identify log modification, we use a Levenshtein ratio of 0.5 as the threshold to determine whether the log change belongs to modification or not. Although we leverage the same threshold as prior studies (Zhao et al. 2017) and the result is found to be accurate, the threshold may have an impact on our findings.

The manual classification on the rationale of adding logging statement may be subjective. To mitigate this threat, two authors of the paper examine the logging statements and we have a third person as a tie-breaker when the two authors cannot make a consensus. The Cohen's kappa statistic (Fleiss and Cohen 1973) of our manual classification is 0.68, i.e., substantial agreement. More user studies and case studies are needed to further address this threat and to provide deeper understanding on the rationale of mobile logging.

In our approach, we choose one popular domain performance metric response time and other physical level performance metrics to measure performance. However, there exist a large number of other performance metrics especially for mobile apps, such as the network traffic over the air. Future studies can consider more performance metrics to complement our study.

5.3 Construct Validity

We use git diff to determine the type of logging statement changes. However, when a file gets renamed and the code is changed more than 50%, git would identify the original file as deleted and the renamed version as added. Therefore, the logging statement in such files would be identified as deleted and added, even though they are not changed at all. In such cases, altering the threshold values would not help eliminate the threat. In order to examine the impact of such threat, we extract added and deleted logging statements that are exactly the same and are from same commit. Such cases may be prone to be impacted by this threat. We only find a total of 3,744 logging statements that belong to this situation, which only accounts for 0.7% of the entire dataset. Therefore, we consider the impact of the threat is minimal.

When calculating logging lines of code, we count the number of logging statements. However, when one logging statement takes more than one lines of source code, the result may be biased. From a sample of 384 randomly selected (95% of confidence level with a 5% confidence interval) logging statements, we find that only 27 of them contain multiple lines of source code.

We find that Android app developers may use Android default logging library, third-party logging libraries, as well as implementing their own custom logging class. With the wide range of subject apps we selected, our regular expression match approach may contain false positives. We manually sampled 384 pieces of logging statements, which corresponds to a 95% of confidence level with a 5% confidence interval. We find that our approach can effectively extract the true logging statements with an accuracy of 99%.

There exist environmental noises when we run tests to measure performance impact of mobile logging (Mytkowicz et al. 2009b). In order to minimize noises in performance measurement, we ignore the tests that are associated with network to eliminate the influence of network fluctuation. Furthermore, we perform repetitive measurement to evaluate performance. Future studies can further increase the number of repeated executions and mock the network access to further complement our study.

6 Related Work

In this section, we discuss prior research related to this paper with regards to empirical studies on logging practices, assisting in logging decisions, and logging performance.

6.1 Empirical Studies on Logging Practices

In practice, logging statements are widely used to expose valuable information of runtime system behavior. Such information helps developers trace, monitor and debug the system. Due to the value of logs, extensive empirical studies on logging practices have been conducted. Yuan et al. (2012a) performed the first empirical study on quantitatively characterizing the logging practices. The authors studied four large open-source software written in C/C++ and found that logging is pervasive, and developers often do not make the log messages right at the first time. They built a simple log-level checker to detect problematic logging levels based on inconsistent verbosity levels within similar code snippets. The work done by Chen and Jiang (2017) is a replication study of Yuan et al. (2012a) by analyzing the

logging practices of 21 Java applications from the Apache Software Foundation (ASF). The applications they studied are selected from server-side, client-side or support-component-based applications. They found that certain aspects of the logging practices in Java-based applications are different from C/C++ based applications.

Shang et al. (2011) and Weiyi et al. (2013) studied the evolution of logs in the forms of both logging statements and the generated logs over multiple releases of large software systems. The findings illustrate that logging statements are often changed by developers without considering the need of other practitioners who highly depend on these logs.

Prior research also presents that logging has a relationship with software quality and often support to trace software issues to improve software quality. Syer et al. (2013) proposed an automated approach that combines performance counters and execution logs to diagnose memory-related issues in load tests. The approach that they developed is fully automated and scales well to ultra-large-scale open-source and enterprise systems. Shang et al. (2015) studied the relationship between logging characteristics and software quality on four releases of Hadoop and JBoss. They found that log related metrics (e.g., log density) have strong correlations with post-release defects. Their findings suggested that software maintainers should allocate more preventive maintenance effort on source code files with more logs and log churn.

Kabinna et al. (2018) and Kabinna et al. (2016b) conducted studies on the stability of logging statements in four open source applications, i.e., Liferay, ActiveMQ, Camel and CloudStack, to reduce the effort that is required to maintain log analysis. The authors found that 20% to 45% of the logging statements are changed at least once. They also built a random forest classifier to determine the change risk of just-introduced and long-lived logging statements. Kabinna et al. found that developers' experience, file ownership, log density and SLOC are the most important change risks of both just-introduced and long-lived logging statements.

Shang et al. (2014b) studied three open source applications: Hadoop, Cassandra and Zookeeper to recover development knowledge for log lines. In their study, they manually examined 300 randomly sampled logging statements and used 45 real-life inquiries in the user mailing lists. They found that it is sometimes difficult for practitioners to understand log lines, development knowledge such as issue reports are useful for log line understanding.

6.2 Assisting Logging Decisions

Although logs are of much value for software practitioners, the usefulness of logs highly depends on the quality of logs. There exists a significant challenge for developers to make proper logging decisions. Such decisions include choosing the logging level, the logging location, the text in the logs and even whether the logging statements needs to be updated.

Prior research has proposed automated approaches to help developers choose the appropriate logging level for newly inserted logging statements. Li et al. (2017a) built an ordinal regression model to automatically provide the suggestion of logging level when developers add logging statements. Hassani et al. (2018) also found that incorrect logging level is one of the typical log-related issues and Hassani et al. implemented a tool to detect incorrect logging levels based on the words that only appear in one logging level.

Logging too little information limits its ability to assist development while too much would cost extra resources. In order to assist logging decision, prior studies have proposed and implemented automated tools supporting to guide logging improvement. Yuan et al. (2012b) analyzed 250 randomly sampled reported issues and found that more than half of them could not be diagnosed due to the lack of log messages. Yuan et al. proposed a tool

name Errlog to analyze the source code to detect exceptions without logging and automatically insert the missing logging statements. On the other hand, for existing logging statements, Yuan et al. (2012c) built a tool called LogEnhancer to enrich the recorded contents by automatically adding additional critical variable information to existing logging statements to aid in future failure diagnosis.

Variance approaches are proposed to automatically suggest where to put logging statements in source code. Fu et al. (2014) studied the logging practices in two industrial software applications written in C# at Microsoft. The authors manually investigated the logged code snippet and found that there exist five categories of logged code snippets. Furthermore, the authors found six factors that are considered for logging and proposes an automatic classification approach to predict whether to log for a code snippet. In another study, Zhu et al. (2015) examined four applications written in C# and performed a machine learning technique to extract structural, textual and syntactic features of code snippets in order to build a logging suggestion tool, LogAdvisor. Their tool automatically learns the common logging practices from existing logging instances and suggest developers with logging locations. Similarly, Li et al. (2018) studied the relationship between logging locations and topics of the source code that contain logging statements. Heng et al. found that logging related topics often vary across projects. However, with an automated classifier built with the topics of the source code, Heng et al. proposed a tool that can successfully suggest logging locations. Zhao et al. (2017) proposed an approach called Log20 to automate the placement of logging statements by measuring the effectiveness of each logging statement in disambiguating code paths by information theory. Given a specified amount of performance overhead, the tool determined a near-optimal placement of log printing statements under the constraint of adding less than the overhead.

Pinjia et al. (2018) conducted an empirical study on the usage of natural language descriptions in logging statements to facilitate the maintenance of logging decisions. In particular, the authors studied the context of logging statements in ten Java applications and seven C# applications and find that there are three categories of logging descriptions in logging statements namely: description for program operation, description for error condition, and description for high-level code semantics. Furthermore, this paper proposed a simple effective method implying the feasibility of automated logging description generation.

Finally, motivated by Shang et al.'s findings (2011, 2013), changing logging statement also needs automated tool support. Li et al. (2017b) analyzed the reasons for log changes and proposed an approach that can provide developers with log change suggestions as soon as they commit a code change. They manually examined 380 random sample of log changes from four open source applications: Hadoop, Directory Server, Commons HttpClient and Qpid, and found that the reasons of log changes can be categorized to four categories: block change, log improvement, dependence-driven change, and logging issue. Based on the reasons, they applied random forest classifiers to provide accurate just-in-time suggestions for logging statement changes.

Prior work that provides automated support on logging decisions are often based on machine learning techniques from a large-scale prior logging data (Li et al. 2017a, b, 2018; Zhu et al. 2015). However, mobile apps often do not have such large amount of data to build these models. In addition, the techniques from prior research are often used as a black box without giving concrete reasons (Li et al. 2017a, b, 2018; Zhu et al. 2015). On the contrary, our findings do not contribute as an automated approach but particularly focus on one aspect, i.e., the inconsistency between the rationale and the verbosity level of logging statements, leading to potential waste of energy from logging. Such a finding may be more easily understood and accepted by developers.

6.3 Logging Performance

Despite the value of logs and the advances of logging libraries, such as Log4j2,[36] logging does require extra system resources and may pose a significant impact on system performance (Chen and Shang 2017). Ding et al. (2015) conducted a survey on 84 Microsoft engineers to understand the participants' experience in logging systems and logging overhead. 80% of the interviewees agreed that logging has a non-trivial overhead. The authors proposed a cost-aware logging mechanism called Log^2 to selectively record useful logs based on a given logging output frequency.

Kabinna et al. (2016a) conducted a case study on logging library migrations within Apache Software Foundation (ASF) projects. They manually analyzed JIRA issues and find that 33 out of 223 ASF projects exist logging library migration. They examined all the JIRA issues that attempt a migration and find that flexibility and performance improvement are the two primary motivations for logging library migrations. However, their finding shows that performance is rarely improved after the logging library migrations.

An exploratory study conducted by Chowdhury et al. (2017) investigates the energy cost of logging in Android apps using GreenMiner, an automated energy test-bed for mobile apps. Chowdhury et al. studied 24 Android apps that were tested with logging enabled and disabled and found that execution logs have a negligible effect on energy consumption for most of the mobile apps tested.

Our work differs from prior work in three dimensions: 1) our study is the first large-scale empirical study on logging practices in mobile apps; 2) we study the rationale of mobile logging in order to deeply understand the unique logging practices; and 3) we perform a statistically rigorous approach to examine the system resources' consumption including response time, CPU and battery consumption.

7 Conclusion

Logging has been widely used by developers to understand, debug and perform failure diagnosis. Prior studies have focused on logging practices in server and desktop applications, helping to make logging decisions and leverage logs to reduce maintenance effort. However, few studies have been conducted on logging practices of mobile apps. Therefore, to fill the gap, in this paper, we investigate the logging practices in mobile apps. Specifically, we study 1,444 open source Android apps in the F-Droid repository.

By studying the characteristics of mobile logging, we find that logging in mobile apps is less pervasive and less actively maintained than logging in server and desktop application, while much more often to be leveraged in a temporary manner. In order to further understand the uniqueness of mobile logging, we conduct both firehouse email interview and qualitative annotation on the rationale of mobile logging. We find that the majority of the logging statements are used for debugging purposes, which helps explain the unique characteristics of mobile logging. However, such debugging logs are often still generated in the released version of the apps, causing potential performance overhead.

To understand the performance impact of mobile logging, we conduct an experiment by comparing the system resource consumption between enabling and disabling logging on eight selected apps. We find that logging can introduce a statistically significant performance overhead such as longer response time and higher battery consumption. Disabling

[36] https://logging.apache.org/log4j/2.x/

the unnecessary logs (e.g., logs for tracing and debugging information) may also provide a statistically significant performance improvement in many scenarios.

The contributions of this paper are as follows:

- Our work both quantitatively and qualitatively analyze the logging characteristics of Android apps.
- We perform a qualitative annotation investigating the rationale behind mobile logging.
- We find that a considerable number of logs is unnecessarily generated during runtime of mobile apps.
- We perform a statistically rigorous approach to measure the performance impact of logging can have in mobile apps.

Our findings advocate for the need for automated tooling support and more advanced logging infrastructure, such as logging libraries, that are specially optimized for mobile apps.

Acknowledgements The authors gratefully thank the developers who participated and shared their thoughts in our email interview.

References

Android (2017) Log. https://developer.android.com/reference/android/util/Log.html

Boulon J, Konwinski A, Qi R, Rabkin A, Yang E, Yang M (2008) Chukwa, a large-scale monitoring system. In: Proceedings of CCA, vol 8, pp 1–5

Chen B, Jiang ZMJ (2017) Characterizing logging practices in java-based open source software projects–a replication study in apache software foundation. Empir Softw Eng 22(1):330–374

Chen J, Shang W (2017) An exploratory study of performance regression introducing code changes. In: 2017 IEEE international conference on software maintenance and evolution (ICSME). IEEE, pp 341–352

Chen TH, Shang W, Jiang ZM, Hassan AE, Nasser M, Flora P (2014) Detecting performance anti-patterns for applications developed using object-relational mapping. In: Proceedings of the 36th international conference on software engineering. ACM, pp 1001–1012

Chen TH, Shang W, Jiang ZM, Hassan AE, Nasser M, Flora P (2016) Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks. IEEE Trans Softw Eng 42(12):1148–1161

Chowdhury S, Di Nardo S, Hindle A, Jiang ZMJ (2017) An exploratory study on assessing the energy impact of logging on android apps. Empir Softw Eng, 1–35

Cliff N (1993) Dominance statistics: ordinal analyses to answer ordinal questions. Psychol Bull 114(3):494

Developer A (2017) Write and view logs with logcat. https://developer.android.com/studio/debug/am-logcat#WriteLogs

Ding R, Zhou H, Lou JG, Zhang H, Lin Q, Fu Q, Zhang D, Xie T (2015) Log2: a cost-aware logging mechanism for performance diagnosis. In: USENIX annual technical conference, pp 139–150

Elye (2018) Debug messages your responsibility to strip it before release!. https://medium.com/@elye.project/debug-messages-your-responsible-to-clear-it-before-release-1a0f872d66f

F-Droid (2017) Free and open source android app repository. https://f-droid.org/

Fleiss JL, Cohen J (1973) The equivalence of weighted kappa and the intraclass correlation coefficient as measures of reliability. Educ Psychol Measur 33(3):613–619

Fu Q, Zhu J, Hu W, Lou JG, Ding R, Lin Q, Zhang D, Xie T (2014) Where do developers log? An empirical study on logging practices in industry. In: Companion proceedings of the 36th international conference on software engineering. ACM, pp 24–33

Harpstead E, Zimmermann T, Nagapan N, Guajardo JJ, Cooper R, Solberg T, Greenawalt D (2015) What drives people: creating engagement profiles of players from game log data. In: Proceedings of the 2015 annual symposium on computer-human interaction in play. ACM, pp 369–379

Hassani M, Shang W, Shihab E, Tsantalis N (2018) Studying and detecting log-related issues. Empir Softw Eng, 1–33

Kabinna S, Bezemer CP, Shang W, Hassan AE (2016a) Logging library migrations: a case study for the apache software foundation projects. In: 2016 IEEE/ACM 13th working conference on mining software repositories (MSR), pp 154–164

Kabinna S, Shang W, Bezemer CP, Hassan AE (2016b) Examining the stability of logging statements. In: 2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER), vol 1. IEEE, pp 326–337

Kabinna S, Bezemer CP, Shang W, Syer MD, Hassan AE (2018) Examining the stability of logging statements. Empir Softw Eng 23(1):290–333. https://doi.org/10.1007/s10664-017-9518-0

Kernighan BW, Pike R (1999) The practice of programming. Addison-Wesley Longman Publishing Co., Inc., Boston

Li H, Shang W, Hassan AE (2017a) Which log level should developers choose for a new logging statement? Empir Softw Eng 22(4):1684–1716

Li H, Shang W, Zou Y, Hassan AE (2017b) Towards just-in-time suggestions for log changes. Empir Softw Eng 22(4):1831–1865

Li H, Chen THP, Shang W, Hassan AE (2018) Studying software logging using topic models. Empir Softw Eng 23(5):2655–2694. https://doi.org/10.1007/s10664-018-9595-8

Lin D, Bezemer CP, Zou Y, Hassan AE (2018) An empirical study of game reviews on the steam platform. Empir Softw Eng, 1–38

Malik H, Hemmati H, Hassan AE (2013) Automatic detection of performance deviations in the load testing of large scale systems. In: Proceedings of the 2013 international conference on software engineering, ICSE '13. IEEE Press, Piscataway, pp 1012–1021. http://dl.acm.org/citation.cfm?id=2486788.2486927

Moore DS, Craig BA, McCabe GP (2012) Introduction to the practice of statistics. WH Freeman

Murphy-Hill E, Zimmermann T, Bird C, Nagappan N (2015) The design space of bug fixes and how developers navigate it. IEEE Trans Softw Eng 41(1):65–81

Mytkowicz T, Diwan A, Hauswirth M, Sweeney PF (2009a) Producing wrong data without doing anything obviously wrong! ACM Sigplan Not 44(3):265–276

Mytkowicz T, Diwan A, Hauswirth M, Sweeney PF (2009b) Producing wrong data without doing anything obviously wrong! SIGPLAN Not 44(3):265–276

Nagappan N, Ball T (2005) Use of relative code churn measures to predict system defect density. In: 27th international conference on software engineering, 2005. ICSE 2005. Proceedings. IEEE, pp 284–292

Pinjia H, Zhuangbin C, Shilin H, Lyu MR (2018) Characterizing the natural language descriptions in software logging statements. In: Proceedings of the 33rd IEEE/ACM international conference on automated software engineering. IEEE Press

Romano J, Kromrey JD, Coraggio J, Skowronek J (2006) Appropriate statistics for ordinal level data: should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys. In: Annual meeting of the Florida association of institutional research, pp 1–33

Shang W, Jiang ZM, Adams B, Hassan AE, Godfrey MW, Nasser M, Flora P (2011) An exploratory study of the evolution of communicated information about the execution of large software systems. In: 2011 18th working conference on reverse engineering, pp 335–344

Shang W, Jiang ZM, Adams B, Hassan AE, Godfrey MW, Nasser M, Flora P (2014a) An exploratory study of the evolution of communicated information about the execution of large software systems. J Softw Evol Process 26(1):3–26

Shang W, Nagappan M, Hassan AE, Jiang ZM (2014b) Understanding log lines using development knowledge. In: 2014 IEEE international conference on software maintenance and evolution (ICSME). IEEE, pp 21–30

Shang W, Nagappan M, Hassan AE (2015) Studying the relationship between logging characteristics and the code quality of platform software. Empir Softw Eng 20(1):1–27. https://doi.org/10.1007/s10664-013-9274-8

Shull F, Singer J, Sjøberg DI (2007) Guide to advanced empirical software engineering. Springer

StackOverflow (2017) Why doesn't "system.out.println" work in android? https://stackoverflow.com/a/2220559

Syer MD, Jiang ZM, Nagappan M, Hassan AE, Nasser M, Flora P (2013) Leveraging performance counters and execution logs to diagnose memory-related performance issues. In: 2013 IEEE international conference on software maintenance, pp 110–119

Tan J, Pan X, Kavulya S, Gandhi R, Narasimhan P (2008) Salsa: analyzing logs as state machines. In: Proceedings of the first USENIX conference on analysis of system logs, WASL'08. USENIX Association, Berkeley, pp 6–6. http://dl.acm.org/citation.cfm?id=1855886.1855892

Weiyi S, Ming JZ, Bram A, HA E, GM W, Mohamed N, Parminder F (2013) An exploratory study of the evolution of communicated information about the execution of large software systems. J Softw: Evol Process 26(1):3–26. https://doi.org/10.1002/smr.1579

Xu W, Huang L, Fox A, Patterson D, Jordan MI (2009) Detecting large-scale system problems by mining console logs. In: Proceedings of the ACM SIGOPS 22nd symposium on operating systems principles, SOSP '09. ACM, New York, pp 117–132. https://doi.org/10.1145/1629575.1629587

Yamane T (1973) Statistics: an introductory analysis

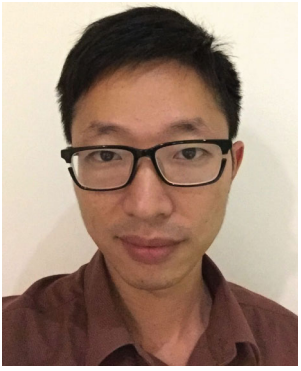
Yuan D, Park S, Zhou Y (2012a) Characterizing logging practices in open-source software. In: Proceedings of the 34th international conference on software engineering. IEEE Press, pp 102–112

Yuan D, Park S, Huang P, Liu Y, Lee MMJ, Tang X, Zhou Y, Savage S (2012b) Be conservative: enhancing failure diagnosis with proactive logging. In: OSDI, vol 12, pp 293–306

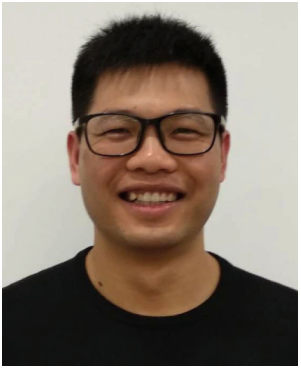
Yuan D, Zheng J, Park S, Zhou Y, Savage S (2012c) Improving software diagnosability via log enhancement. ACM Trans Comput Syst (TOCS) 30(1):4

Zhao X, Rodrigues K, Luo Y, Stumm M, Yuan D, Zhou Y (2017) Log20: fully automated optimal placement of log printing statements under specified overhead threshold. In: Proceedings of the 26th symposium on operating systems principles. ACM, pp 565–581

Zhu J, He P, Fu Q, Zhang H, Lyu MR, Zhang D (2015) Learning to log: helping developers make informed logging decisions. In: Proceedings of the 37th international conference on software engineering, vol 1. IEEE Press, pp 415–425



Yi Zeng is a Master's student at the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. He obtained his B.Sc. from Sun Yat-sen University in Guangzhou, China. His research interests are mining software repositories, software log analysis and software performance engineering.



Jinfu Chen is a Ph.D. student in the Department of Computer Science and Software Engineering at Concordia University, Montreal. He has received his M.Sc. degree from Chinese Academy of Sciences and he obtained B.Eng. from Harbin Institute of Technology. His research interest lies in empirical software engineering, software performance engineering, performance testing, performance mining.

Weiyi Shang is an Assistant Professor and Concordia University Research Chair in Ultra-large-scale Systems at the Department of Computer Science and Software Engineering at Concordia University, Montreal. He has received his Ph.D. and M.Sc. degrees from Queens University (Canada) and he obtained B.Eng. from Harbin Institute of Technology. His research interests include big data software engineering, software engineering for ultra-largescale systems, software log mining, empirical software engineering, and software performance engineering. His work has been published at premier venues such as ICSE, FSE, ASE, ICSME, MSR and WCRE, as well as in major journals such as TSE, EMSE, JSS, JSEP and SCP. His work has won premium awards, such as SIGSOFT Distinguished paper award at ICSE 2013 and best paper award at WCRE 2011. His industrial experience includes helping improve the quality and performance of ultra-large-scale systems in BlackBerry. Early tools and techniques developed by him are already integrated into products used by millions of users worldwide. Contact him at shang@encs.concordia.ca; http://users.encs.concordia.ca/~shang.

Tse-Hsun (Peter) Chen is an Assistant Professor in the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. He obtained his BSc from the University of British Columbia, and MSc and Ph.D. from Queen's University. Besides his academic career, Dr. Chen also worked as a software performance engineer at BlackBerry for over four years. His research interests include performance engineering, database performance, program analysis, log analysis, and mining software repositories. Early tools developed by Dr. Chen were integrated into industrial practice for ensuring the quality of large-scale enterprise systems. More information at http://petertsehsun.github.io/.

Affiliations

Yi Zeng[1] · Jinfu Chen[1] · Weiyi Shang[1] · Tse-Hsun (Peter) Chen[1]

Jinfu Chen
fu_chen@encs.concordia.ca

Weiyi Shang
shang@encs.concordia.ca

Tse-Hsun (Peter) Chen
peterc@encs.concordia.ca

[1] Department of Computer Science and Software Engineering, Concordia University, Montreal, Quebec, Canada