BenchPress: Analyzing Android App Vulnerability Benchmark Suites

Joydeep Mitra Venkatesh-Prasad Ranganath Aditya Narkar
Kansas State University, USA
{joydeep,rvprasad,avnarkar}@ksu.edu

Created: January 29, 2019. Revised: September 19, 2019.

Abstract

In recent years, various benchmark suites have been developed to evaluate the efficacy of Android security analysis tools. The choice of such benchmark suites used in tool evaluations is often based on the availability and popularity of suites and not on their characteristics and relevance. One of the reasons for such choices is the lack of information about the characteristics and relevance of benchmarks suites.

In this context, we empirically evaluated four Android specific benchmark suites: DroidBench, Ghera, ICCBench, and UBCBench. For each benchmark suite, we identified the APIs used by the suite that were discussed on Stack Overflow in the context of Android app development and measured the usage of these APIs in a sample of 227K real world apps (coverage). We also compared each pair of benchmark suites to identify the differences between them in terms of API usage. Finally, we identified security-related APIs used in real-world apps but not in any of the above benchmark suites to assess the opportunities to extend benchmark suites (gaps).

The findings in this paper can help 1) Android security analysis tool developers choose benchmark suites that are best suited to evaluate their tools (informed by coverage and pairwise comparison) and 2) Android app vulnerability benchmark creators develop and extend benchmark suites (informed by gaps).

1 Introduction

1.1 Motivation

Effectiveness of Android security analysis tools is evaluated with benchmarks and real-world apps. The effectiveness of static taint analysis tools like AmanDroid [20], FlowDroid [2], HornDroid [4], and IccTA [9] has been evaluated by applying them to benchmarks from DroidBench, ICCBench, and UBCBench [14] benchmark suites and comparing tool verdicts with benchmark labels that indicate the presence/absence of specific vulnerability or malicious behavior.

Such tool evaluations have used benchmarks without evaluating the authenticity and the representativeness of the benchmarks. *Authenticity* is the truthfulness of the claim about the presence/absence of a vulnerability or malicious behavior in a benchmark (Section 2.2.2 in [11]). *Representativeness* is the similarity between the manifestation/occurrence of a vulnerability in a benchmark and in real-world apps (Section 3 in [15]). Consequently, the usefulness of findings from these evaluations is diminished in terms of the ability of tools and techniques to detect vulnerabilities or malicious behaviors (due to authenticity) and the general applicability of tools and techniques (due to representativeness).

Recently, there have been two efforts focused on the authenticity of benchmarks. Mitra and Ranganath [11] created Ghera, a suite of demonstrably authentic Android app vulnerability benchmarks, to address the issue of authenticity. They also established the representativeness of Ghera benchmarks (in terms of API usage) [15]. Pauck et al. [13] developed ReproDroid, a tool to help verify the authenticity of Android app vulnerability benchmarks. They found that not all claims about the presence/absence of vulnerabilities in benchmarks in DIALDroid, DroidBench, and ICCBench benchmark suites were true.

It is common in other communities to study and characterize benchmarks. In the program analysis community, Blackburn et al. [3] developed and used metrics based on static and dynamic properties of programs to characterize and compare the DaCapo benchmarks with SPEC Java benchmarks [17]. Isen et al. [8] measured several properties of embedded Java benchmarks and how well they represent real-world mobile apps. In the systems community, Pallister et al. [12] characterized benchmarks based on the energy consumption properties of embedded platforms. In the database community, such assessments have been around since 1990s [6]. However, such close scrutiny of benchmark suites have not occurred in the Android security community.

Motivated by the aforementioned efforts to analyze and characterize benchmarks, we undertook an effort to assess the representativeness of multiple Android app vulnerability benchmark suites.i.e., how well does a benchmark suite represent real-world apps?

1.2 Research Questions

The objective of our effort is to answer the following research questions:

- **RQ1** *In general, do Android app vulnerability benchmark suites use APIs that are used by real-world apps and discussed by Android app developers?* The question is aimed at understanding the representativeness of benchmark suites and the relevance of the APIs used to capture vulnerabilities in benchmark suites. The answer to this question can help associate an element of confidence to benchmarks and, consequently, to tool evaluations that use such benchmarks.

- **RQ2** *In the context of security, do Android app vulnerability benchmark suites use APIs that are used by real-world apps and discussed by Android app developers?* Similar to RQ1, this question is intended to understand the representativeness and relevance of the security-related APIs used by benchmark suites but in the specific context of security.

- **RQ3** *How do the considered benchmark suites differ in terms of API usage?* The purpose of this question is to identify the common and unique APIs between benchmark suite pairs. The answer to this question can help tool developers choose appropriate benchmark suites to test/evaluate their tools.

- **RQ4** *Do real-world apps use security-related APIs not used by any benchmark suite?* The purpose of this question is to identify gaps between existing benchmark suites and the real-world apps in terms of security-related APIs. The answer to this question can steer security analysis efforts towards unexplored Android APIs to possibly uncover new vulnerabilities and enhance existing benchmark suites.

1.3 Contributions

In this paper, we make the following contributions:

- Provide empirical evidence about the representativeness of four Android app vulnerability benchmark suites.

- Identify gaps between the evaluated benchmark suites and real world apps in terms of APIs that are used in real world apps but not in the benchmark suites.

- Extend and improve the framework for empirical evaluation of Android app vulnerability benchmarks introduced by Ranganath & Mitra [15] because we believe this framework can be used by other researchers to conduct similar studies in other domains as well.

In addition to these contributions, we hope this effort will spark the interest of the empirical software engineering community to study Android app vulnerability benchmarks and help improve Android app security.

The remainder of the paper is structured as follows. Section 2 outlines the metric of representativeness along with the benchmark suites and the real world app sample used in the study. Section 3 describes the experiment to measure representativeness. Sections 4-7 discuss the answers to posed research questions. Section 8 describes the threats to the validity of the experiment. Section 9 describes related work. Section 10 provides information about the artefacts used in this effort. Section 12 summarizes the findings from this effort.

2 Concepts and Subjects

2.1 API usage as a measure of representativeness

Representative vulnerability benchmarks should have two aspects. First, they should capture vulnerabilities that occur in the real world. Second, the manifestation of vulnerabilities in representative benchmarks should be similar (if not identical) to that in real-world apps.

Ranganath and Mitra [15] observed this challenge while establishing the representativeness of Ghera benchmarks. So, they introduced the notion of using API usage as a weak but general measure of representativeness of benchmarks. They reasoned "the likelihood of a vulnerability occurring in real-world apps is directly proportional to the number of real-world apps using the Android APIs involved in the vulnerability". Consequently, to measure the representativeness of benchmarks, they measured how often APIs used in benchmarks were used in real-world apps.

In this evaluation, we use the above notion and a similar approach to measure the representativeness of benchmarks.

2.2 Benchmarks

For this study, we considered 4 benchmark suites related to Android app vulnerabilities: *DroidBench, Ghera, IccBench,* and *UBCBench.*

DroidBench [13] contains 211 benchmarks. Each benchmark is an Android app that captures zero or more information leak vulnerabilities. The vulnerabilities captured in DroidBench primarily stem from Inter-Component Communication (ICC) feature of Android and general features of Java.

Ghera [11] contains 60 benchmarks that capture mostly known Android app vulnerabilities along with few unknown Android app vulnerabilities. Each benchmark includes 3 Android apps: 1) a *benign* app that contains vulnerability x, 2) a *malicious* app that exploits vulnerability x in the benign app, and 3) a *secure* app that does not contain vulnerability x and thus cannot be exploited by the malicious app. In this evaluation, we considered only the benign apps from each benchmark and we will refer to them as Ghera benchmarks in the rest of this paper. Unlike in DroidBench, the vulnerabilities in Ghera stem from different Android features including ICC.

IccBench [13] contains 24 benchmarks. Each benchmark is an Android app that captures zero or more information leak vulnerabilities. IccBench focuses on capturing vulnerabilities that stem from communcation between apps via ICC.

UBCBench [14] contains 16 benchmarks. Each benchmark is an Android app that captures at most one information leak vulnerability. UBCBench captures information flow vulnerabilities primarily stemming from ICC and SharedPreferences[1] features of Android and general features of Java.

2.3 Real World Apps

We collected 700K apps from AndroZoo [1] in March 2019. From this set of 700K apps, we curated a set of 473K apps that target API levels 19 thru 27. An API level uniquely identifies the framework API revision offered by a version of the Android platform. In an Android app, the *minimum API level* is the least framework API version required by the app and *target API level* is the framework API version targeted by the app. For this evaluation, we initially picked target API level 19 thru 27 because most benchmarks targeted these API levels. However, we later discovered that Android currently does not support API levels 19 thru 22. Therefore, to make the evaluation current, from the set of 473K apps, we retained only apps that target API levels 23 thru 27. Hence, we ended up with a sample of 226K real-world Android apps. Table 1 provides the distribution of this sample across considered target API Levels.

[1]A SharedPreference is a file that stores key-value pairs and can be private to an app or shared

Target API level	# Real World Apps
23	146K
24	18K
25	16K
26	29K
27	17K
Total	226K

Table 1: Distribution of target API levels in the sample of real world apps

3 Experiment

3.1 Preparing the benchmarks

By design, each Android app is bundled as a self-contained APK file that contains all code and resources necessary to execute the app but are not provided by the underlying Android framework. However, *due to the build process of Android apps, the APKs may contain unnecessary code and resources (e.g. unused methods).* So, ProGuard [7] tool can be used as part of the Android app build process to remove unnecessary artifacts from APKs.

Every benchmark suite considered in this evaluation provides pre-built APKs and source files for each of its benchmarks. The pre-built APKs provided by DroidBench, ICCBench, and UBCBench contain unnecessary code and resources. Also, the benchmarks do not have the same minimum and target API levels. Specifically, DroidBench benchmarks have minimum API level 8 and target API level 14 thru 24, Ghera benchmarks have minimum API level 22 and target API level 27, ICCBench benchmarks have minimum and target API level 25, and UBCBench benchmarks have minimum and target API level 19.

Since we wanted to measure the representativeness of benchmark suites and compare them based on API usage, we needed to control for the effects of unnecessary APIs and API level on the findings of the evaluation. Therefore, we rebuilt every benchmark from its source with minimum API level set to 23, target API level set to 27, using appcompat support library version 27.1.1, and using Proguard to remove unnecessary APIs. We chose API levels 23 thru 27 because they are currently supported by Android.

We ensured the rebuilt benchmarks were indeed supported by API levels 23 thru 27 by executing each benchmark on an emulator running Android 23 and 27. As part of the execution, we manually interacted with the app until no further interaction was possible. Often, this meant interacting with various widgets on a screen and navigating to various screens in an app.

If the benchmark or app crashed, then we recorded the crash. Table 2 lists the total number of benchmarks in each suite, the number of benchmarks that we were able to successfully build, and the number of benchmarks that crashed during execution. In this evaluation, we considered all benchmarks that could be built successfully including the ones that crashed. We considered the crashed benchmarks because we were unsure if they crashed due to a vulnerability intentionally captured in the benchmark or other reasons such as change of API levels.

Observations

From Table 2, we see that, most benchmarks not only build but also execute on the currently supported versions of Android (even when they were not designed to run on those versions) – out of 311 benchmarks across all benchmark suites, only 35 crashed during execution and only 10 could not be built successfully. So, *while most benchmarks were not explicitly designed to run on recent versions of Android, they are well supported by recent versions of Android.*

The 10 benchmarks (from DroidBench) that failed to build imply *some benchmarks are not supported by recent versions of Android*; an important aspect that should be considered when using DroidBench to evaluate effectiveness of Android security analysis tools.

For the 35 benchmarks (32 from DroidBench and 3 from UBCBench) that crashed when executed on emulators running Android 23 and 27, we re-executed pre-built counterparts of these benchmarks on an emulator running the version of Android originally targeted by the benchmarks. Interestingly, all of the

Benchmark Suite	# Total benchmarks	# Benchmarks built successfully	# Benchmarks crashed
DroidBench	211	201	32
Ghera	60	60	0
ICCBench	24	24	0
UBCBench	16	16	3

Table 2: Total No. of benchmarks in each benchmark suite along with the No. of benchmarks that built successfully with minimum API level 23 and target API level 27 and crashed on an emulator running Android 23 and 27.

Repo	Number of APIs				
	Total	Considered	Filtered	Relevant	Security
DroidBench	2188	837	798	769	744
Ghera	1906	565	518	504	494
ICCBench	185	102	70	70	70
UBCBench	751	127	99	98	96

Table 3: Number of APIs used by the benchmark suites

benchmarks crashed during re-execution. *This raises the question "are these 35 benchmarks from DroidBench and UBCBench valid and, hence, authentic?"*

3.2 API-based App Profiling

The APK file of an app contains an XML-based manifest file and a DEX file that contains the code and data (i.e., resources, assets) of the app. Android apps use various features of the underlying Android framework via XML-based manifest files and published Android programming APIs. We refer to these published Android programming APIs and the XML elements and attributes of manifest files collectively as APIs.

We use an adaptation of the method used by Ranganath and Mitra [15] to determine the APIs used by the sample of real-world Android apps and by the benchmarks.

In this method, for each app, the elements and attributes in its manifest along with all callback methods and all methods that were used but not defined in the app are considered while ignoring obfuscated methods, i.e., methods with single character names. Further, to make apps comparable, for overridden methods and fields, class hierarchy analysis is used to consider the overridden methods and fields as opposed to the overriding methods and fields. Of these APIs, only APIs whose fully qualified name (FQN) contained the prefix *android, com.android, java,* or *org* are considered because the method focuses on measuring representativeness in terms of Android APIs.

Numerous APIs are commonly used in almost all Android apps and are related to aspects (e.g., UI renderin g) that are not the focus of vulnerability benchmarks related to Android apps. To avoid their influence on the experiment, such APIs are ignored while determining the APIs used in the benchmarks. For this p urpose, we created a baseline app with minimum and target API levels as 23 and 27, respectively. This app did not exhibit any interesting functionality but contained graphical widgets commonly used by Andr oid apps. Out of the 1847 APIs used in this baseline app, we manually identified 1586 APIs as commonly used in Android apps; almost all of them were basic Java APIs or related to UI rendering and XML proces sing. For each benchmark suite, we ignored these 1586 APIs from the list of recorded APIs (from the pre vious step) to create the set of *considered* APIs. Even in the *considered* set, we identified APIs orthogonal to the focus of these benchmarks, e.g., *android.graphics*. So, we filtered out these APIs to create a set of *filtered* APIs. This additional filtering did not change the API sets drastically as can be seen from the 2nd and 3rd columns in Table 3.

3.3 Using Android app developer discussions in Stack Overflow to identify relevant and security-related APIs

Ranganath and Mitra [11] considered the set of filtered APIs as *relevant* to Android app development. From this set, they manually identified a subset of (*security-related*) APIs as related to Android security. Since the experimenter's subjectivity and bias could influence the findings of the evaluation via this manual identification, we decided to use Stack Overflow data to identify security-related APIs. Further, we also decided to use Stack Overflow data to prune the set of filtered APIs into the set of relevant APIs.

For this purpose, we used a snapshot of Stack Overflow posts from March 2019 [18] as follows:

Identified Android related posts We considered all posts from the Stack Overflow snapshot with the *Android* tag. This resulted in 1.2 million posts.[2]

Identified Android security related posts There is no readily available information in Stack Overflow data to identify security related posts. So, we used the data from *Security Stack Exchange*, a forum for discussing software security issues, to identify security related posts on Stack Overflow [18]. We gathered all tags from Security Stack Exchange as the set of *security-related tags*. We also added the *security* tag to this set. From the Stack Overflow posts that had *Android* tag, we identified posts with at least one *security-related* tag. We ended up with a set of 460K posts related to *Android security*.

Filtered posts based on API levels Since API level 23 was released in 2015, we considered only posts that had some activity – created, were answered, edited, commented on, voted on, or marked as favorite or accepted – on or after 2015. An API that did not garner interest after 2015 is likely deprecated in API levels 23 thru 27 or well understood by developers. In either case, we deemed such APIs as irrelevant to our evaluation. After this filtering, we ended up with 831K Android related posts and 318K Android security related posts.

Identified relevant and security-related APIs If the class name and the method/field name of an API co-occurred in a post, then we considered the post as discussing the API. Based on this notion, if a filtered API was discussed in an Android related post, then the API was deemed as a *relevant API*, i.e., relevant to Android app development. Similarly, if a filtered API was discussed in an Android security related post, then we deemed it as a *security-related API*.

3.4 Measuring Representativeness with API Usage Percentage

For each benchmark suite, for each corresponding relevant API, we calculated the percentage of real world apps that used the API. We did the same for each security-related API.

3.5 Examining Gap between Real World Apps and Benchmarks

Of the 2118K APIs used by the apps in our real world app sample, we ignored the APIs used in all benchmarks in DroidBench, Ghera, ICCBench, and UBCBench. Of the remaining 2115K APIs, we ignored APIs related to UI and third party libraries because such APIs are not the focus of the benchmarks being evaluated. The remaining 26K APIs were spread across 31 unique package-prefixes[3]. From this set, we identified security-related APIs using Stack Overflow posts with at least one security-related tag (as described in Section 3.3).

[2]Once we considered every Stack Overflow posts with the *Android* tag, we wondered if we were missing out on posts that were related to Android but did not have the *Android* tag. To account for such posts, we collected all tags (including synonyms) from *Android Stack Exchange*, a forum for discussing issues related to Android. From this set of 1347 tags, we removed 433 tags (including synonyms) that had been considered in the 1.2 million Stack Overflow posts. Of the remaining 914 tags, we removed tags related to companies as they were not related to Android app development. Of the remaining 445 tags, only 6 tags – instapaper, pebble, sdhc, task-management, xfat – were associated with *Stack Overflow* posts. Upon examination, we decided that none of the six tags were related to Android app development and hence we ignored them. Since this extra step did not influence the number of posts related to Android, we did not use it in our experiment.

[3]A package-prefix of an API is a substring of its package name from the first character to the character before the second occurring separator. For example, the package-prefix of the package *android/app/Activity* is *android/app* or the package-prefix of the package *android/telephone/gsm/SMSManager* is *android/telephony*.

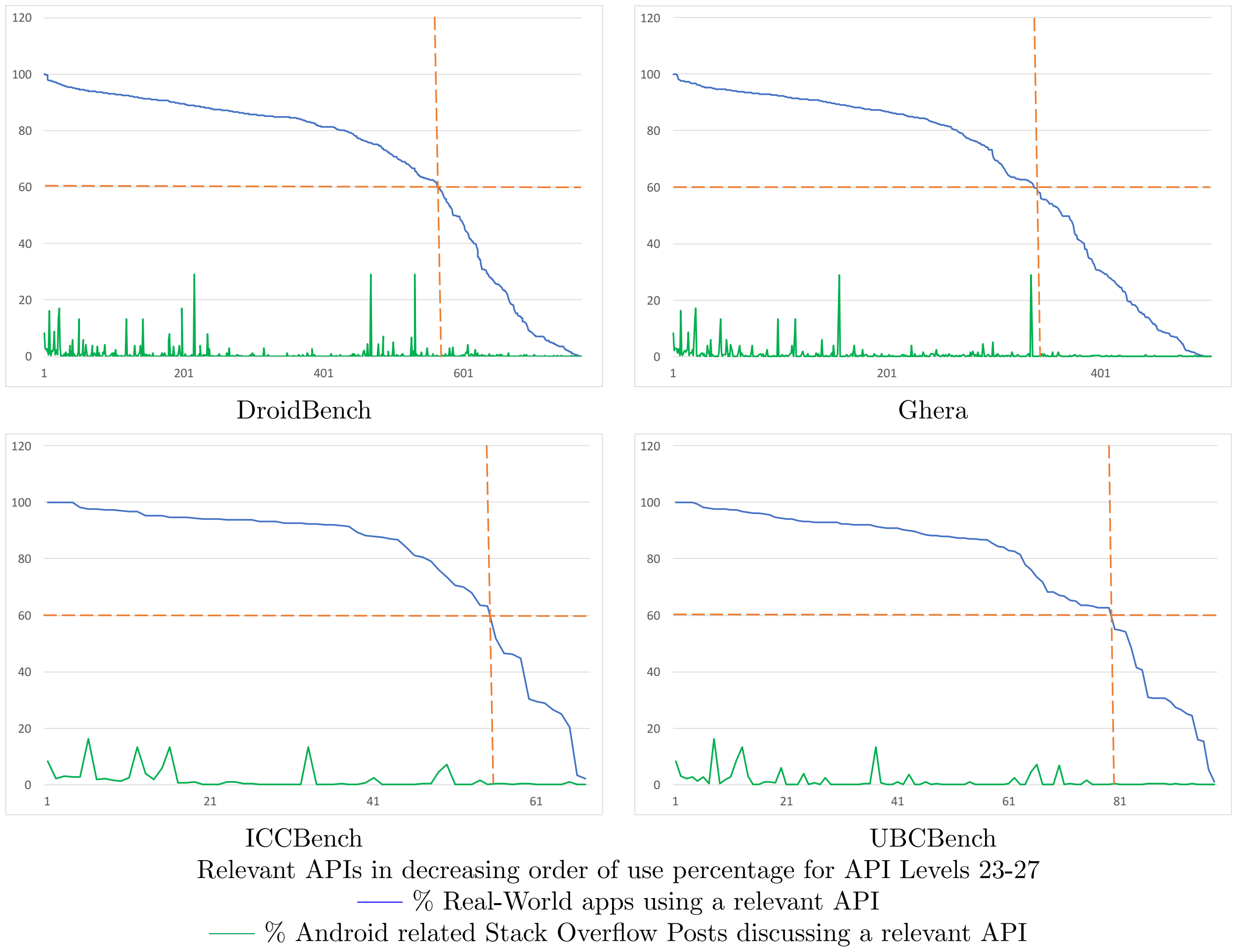

Figure 1: Percentage of real-world apps that use a relevant API and the percentage of Stack Overflow posts that discuss a relevant API in a benchmark suite.

Based on the frequency of Stack Overflow posts discussing an API, we selected the top 10 APIs across 27 package-prefixes and manually examined them to see if they could lead to a vulnerability and, consequently, to the creation of a new benchmark. We ignored 4 package-prefixes – *android/print, android/printservices, android/inputmethodservice,* and *android/Manifest* retrospectively since we realized that they contain APIs pertaining to features (e.g., UI and printing services) that are not the focus of the considered benchmark suites.

4 Answering RQ1

In general, do Android app vulnerability benchmark suites use APIs that are used by real-world apps and discussed by Android app developers?

For each benchmark suite, the corresponding graph in Figure 1 shows the percentage of real-world apps using a relevant API that is used by the suite along with the percentage of Stack Overflow posts discussing the same API.

Table 4 and Table 5 list the five-number summary of the percentage of posts discussing relevant and security-related APIs, respectively.

DroidBench We found that only 29 of the 798 filtered APIs used by DroidBench are not discussed by any Android related Stack Overflow post. The remaining 769 APIs are discussed in at least 1 post. As seen in Table 4, more than half the 769 relevant APIs used by DroidBench are discussed by at least 385 posts. Therefore, *APIs used by DroidBench are being discussed by Android app developers.*

The graph for DroidBench in Figure 1 shows that all relevant APIs are used by real-world apps and 562 (73%) relevant APIs are used by more than 60% of real-world apps. Therefore, *DroidBench is representative of real-world apps in terms of API usage.*

Ghera We found that only 14 of the 518 filtered APIs used by Ghera are not discussed in any Android related Stack Overflow post. The remaining 504 (relevant) APIs are discussed in at least one *Stack Overflow* post. As seen in Table 4, more than half the 504 relevant APIs are discussed by at least 845 posts. Therefore, *APIs used by Ghera are being discussed by Android app developers.*

The graph for Ghera in Figure 1 shows that all relevant APIs are used by real-world apps and 340 (67%) relevant APIs are used by more than 60% of real-world apps. Therefore, *Ghera is representative of real-world apps in terms of API usage.*

ICCBench We found that all of the 70 filtered APIs used by ICCBench are discussed by at least one Android related Stack Overflow post. As seen in Table 4, more than half of the 70 relevant APIs used by ICCBench are discussed by at least 2446 posts. Therefore, *APIs used by ICCBench are being discussed by Android app developers.*

The graph for ICCBench in Figure 1 shows that all relevant APIs are used by real-world apps and 55 (78%) relevant APIs are used by more than 60% of real-world apps. Therefore, *ICCBench is representative of real-world apps in terms of API usage.*

UBCBench We found that 98 of the 99 filtered APIs used by ICCBench are discussed by at least one Android related Stack Overflow post. As seen in Table 4, more than half the 98 relevant APIs used by UBCBench are discussed by at least 1406. Therefore, *APIs used by UBCBench are being discussed by Android app developers.*

The graph for UBCBench in Figure 1 shows that all relevant APIs are used by real-world apps and 79 (81%) relevant APIs are used by more than 60% of real-world apps. Therefore, *UBCBench is representative of real-world apps in terms of API usage.*

In short, *DroidBench, Ghera, ICCBench, and UBCBench are all representative of real-world apps in terms of API usage.*

4.1 Discussion

Comparatively, DroidBench (769) and Ghera (504) use more than five times the number of relevant APIs used by UBCBench (98) and ICCBench (70). In terms of the percentage of relevant APIs used by more than 60% of the real-world apps, DroidBench (562) and Ghera (340) use more than four times the number of relevant APIs used by UBCBench (79) and ICCBench (55). So, in terms of coverage of APIs used by real-world apps, DroidBench and Ghera fare better than UBCBench and ICCBench.

In Figure 1, most of the spikes in the line corresponding to Android related Stack Overflow posts are associated with relevant APIs that are used by more than 60% of the real-world apps. Hence, the benchmarks are not only representative but also relevant since they are using APIs that are not only used by a large number of real-world apps but are also being discussed widely by Android app developers.

5 Answering RQ2

In the context of security, do Android app vulnerability benchmark suites use APIs that are used by real-world apps and discussed by Android app developers?

As shown in Table 3, the number of *security-related* APIs as deemed by Stack Overflow data is similar to the number of relevant APIs. For example, in ICCBench, the number of security-related APIs is identical

Repo	% (No.) of relevant posts discussing APIs									
	Min		Q1		Median		Q3		Max	
DroidBench	0.0001	(1)	0.0068	(57)	0.04	(385)	0.29	(2423)	29.0	(240K)
Ghera	0.0001	(1)	0.0156	(130)	0.10	(845)	0.44	(3668)	29.0	(240K)
ICCBench	0.0007	(6)	0.0588	(489)	0.29	(2446)	1.71	(14242)	15.3	(127K)
UBCBench	0.0001	(1)	0.0348	(289)	0.17	(1406)	0.95	(7880)	15.3	(127K)

Table 4: Five-Number summary of 831K relevant posts discussing APIs in a benchmark suite

Repo	% (No.) of security-related posts discussing APIs									
	Min		Q1		Median		Q3		Max	
DroidBench	0.0003	(1)	0.007	(22)	0.05	(168)	0.31	(997)	35.0	(111K)
Ghera	0.0003	(1)	0.020	(65)	0.11	(359)	0.49	(1559)	35.0	(111K)
ICCBench	0.0009	(3)	0.056	(!79)	0.273	(869)	1.55	(4940)	14.3	(45K)
UBCBench	0.0006	(2)	0.033	(106)	0.20	(640)	1.08	(3458)	14.3	(45K)

Table 5: Five-Number summary of 318K security-related posts discussing APIs in a benchmark suite

to the number of relevant APIs. Moreover, as seen in Table 4, the distribution of posts discussing security-related APIs is similar to that of relevant APIs as can be seen from the five-number summary. Consequently, the observations for relevant APIs carries over to security-related APIs.

This observation differs from that made by Ranganath & Mitra [15] because they *manually* identified security-related APIs while we used Stack Overflow data to identify security-related APIs. They deemed 601 APIs in Ghera as relevant and identified that 117 of the them were related to security. On the contrary, we deemed 504 APIs as relevant and discovered that 494 of them were related to security. Interestingly, all the 117 security-related APIs they identified, were deemed as security-related by us as well.

Caveat The numbers in Table 4 suggest that almost all of the relevant APIs used by a benchmark suite are related to security. These numbers are based on our approach of identifying relevant and security-related APIs using Stack Overflow data as explained in Section 3.3. While our approach considers the occurrence of APIs in posts to identify an API as relevant or security-related, it does not consider the context in which the API occurs in posts, i.e., an API can occur in a post as part of a code snippet that is being discussed but yet not be discussed in the post. Hence, our approach can conservatively identify APIs that are irrelevant or not related to security as relevant or security-related. Therefore, the number of relevant APIs and security-related APIs used by a benchmark suite is likely lower than the numbers reported here.

6 Answering RQ3

How do the considered benchmark suites differ in terms of API usage?

We answer RQ3 by presenting observations based on pairwise comparison of the benchmark suites.

For each pairwise comparison, Table 6 shows the number of filtered APIs common and unique to the compared benchmarks.

6.1 DroidBench vs Ghera

Observation 1 DroidBench uses 1.5 times the number of APIs used by Ghera but it contains almost 3 times the number of benchmarks in Ghera; see Table 3. Therefore, *the difference in API usage between DroidBench and Ghera is not comparable to the difference in the number of benchmarks in them.* This is most likely because DroidBench focuses on heavily ICC (depth) whereas Ghera focuses on ICC and other Android features (breadth). Consequently, the benchmarks in DroidBench use more common APIs compared to the benchmarks in Ghera.

Benchmark Suite Pair (X-Y)	Common APIs	APIs unique to X	APIs unique to Y
DroidBench-Ghera	344	454	174
DroidBench-ICCBench	67	731	3
DroidBench-UBCBench	89	709	10
Ghera-ICCBench	42	476	28
Ghera-UBCBench	85	433	14
ICCBench-UBCBench	16	54	83

Table 6: Filtered APIs based pairwise comparison of benchmark suites

Observtion 2 DroidBench uses 454 APIs not used by Ghera; see Table 6. 438 of these APIs were identified as relevant using Stack Overflow data. Moreover, 299 (68%) relevant APIs are used by at least 60% real-world apps. Of the 438 relevant APIs, approximately 100 are not specific to Android but related to Java. Of the remaining APIs, close to 50% are related to ICC. *Since a large number of APIs unique to DroidBench are related to ICC, evaluations of Android app vulnerability detection tools, especially the ones that focus on ICC, should consider DroidBench.*

Moreover, 344 APIs are common to DroidBench and Ghera. Of these, 331 APIs are relevant and 280 (85%) APIs are used by at least 60% of real-world apps. 200 of the 344 APIs are related to ICC. This is not surprising as DroidBench focuses on ICC. Therefore, tools focusing on detecting vulnerabilities stemming from ICC can use either DroidBench or Ghera for evaluation. However, *since only 65 of the 174 APIs unique to Ghera are related to ICC, such tools should prefer DroidBench over Ghera.*

Observation 3 From Table 6, we see Ghera uses 174 APIs that are not used by DroidBench. 173 of these APIs were identified as relevant using Stack Overflow data. Of these relevant APIs, 69 (40%) APIs are used by at least 60% real-world apps. Further, 93 (54%) of the 173 relevant APIs are related to Android features such as web, crypto, storage, and networking features. *Since Ghera benchmarks capture vulnerabilities stemming from the use of APIs not related to ICC, evaluations of tools that detect vulnerabilities not related to ICC should consider Ghera.*

Observation 5 All benchmarks in Ghera capture vulnerabilities that can be reproduced and exploited on API Levels 23 thru 27. However, DroidBench benchmarks were designed to run on older API Levels. While we were able to build the benchmarks and install them on emulators running API Levels 23 thru 27, there is no evidence to suggest that the captured vulnerabilities can be reproduced and exploited on API levels 23 thru 27. Therefore, evaluations based on DroidBench should be aware of this limitation. *A prudent tool evaluation strategy is to equally consider both DroidBench and Ghera.*

6.2 DroidBench vs ICCBench

Observtion 6 ICCBench uses only 3 APIs that are not used by DroidBench. *Since DroidBench uses almost all the APIs used by ICCBench, DroidBench should be preferred over ICCBench.*

Observtion 7 ICCBench benchmarks were designed to run on API level 25 whereas DroidBench benchmarks were designed to run on API levels 22 and less. Consequently, two of the three APIs related to runtime checking of permissions are used by ICCBench by not by DroidBench as these APIs were introduced in the API level 23. *Therefore, if more current aspects of Android need to be considered, then ICCBench should be used in conjunction with Ghera.*

6.3 DroidBench vs UBCBench

Observation 8 As per Table 6, UBCBench and DroidBench share 89 APIs. *Since only 10 APIs are unique to UBCBench and DroidBench uses almost all the APIs in UBCBench, DroidBench should be preferred over UBCBench.*

Observation 9 Nine of the 10 APIs unique to UBCBench are related to general Java features and one API is related to SharedPreferences, a storage related feature in Android apps. This API is used by 90% of the real-world apps and discussed by close to 1000 Android security related Stack Overflow posts which makes this API highly relevant. So, *UBCBench should be considered in conjunction with DroidBench for tools that target vulnerabilities stemming from the use of SharedPreferences.*

6.4 Ghera vs ICCBench

Observation 10 Ghera covers most of the ICC related APIs used by ICCBench as seen by the fact that Ghera uses 42 of the 70 APIs used by ICCBench. Furthermore, Ghera uses 476 APIs not used by ICCBench. *Since Ghera uses more than 50% of the APIs in ICCBench and is not limited to ICC, Ghera should be preferred over ICCBench.*

Observation 11 ICCBench uses 28 APIs not used by Ghera. All 28 are relevant and 25 of these APIs are used by more than 60% of real-world apps. Moreover, 23 of these APIs are related to ICC which is to be expected since ICCBench focuses on ICC. *Since ICCBench uses a non-trivial number of ICC-related relevant APIs not used by Ghera, ICCBench should be considered in conjunction with Ghera.*

6.5 Ghera vs UBCBench

Observation 12 As per Table 6, UBCBench and Ghera share 85 APIs and UBCBench has only 14 unique APIs. *Since Ghera uses almost all the APIs used by UBCBench, Ghera should be preferred over UBCBench.*
.

Observation 13 6 of the 14 APIs unique to UBCBench are related to SharedPreferences and ICC while the remaining 8 are related to general Java features. *Since all 14 APIs are used by more than 2000 real-world apps and 10 of the 14 APIs are used by 60% of the real-world apps or more, UBCBench should be considered in conjunction with Ghera when evaluating tools that target vulnerabilities stemming from the use of SharedPreferences.*

Observation 14 Similar to DroidBench, the benchmarks in UBCBench were designed to run on API Level 19. Therefore, the authenticity of these benchmarks on API levels 23 thru 27 is unknown. Consequently, since DroidBench uses almost all of the APIs used by UBCBench, *the combination of Ghera and DroidBench should be preferred over the combination of Ghera and UBCBench.*

7 Answering RQ4

Do real-world apps use security-related APIs not used by any benchmark suite?

As explained in Section 3.5, we identified 26K APIs were used by apps in our real-world app sample but not used by any benchmark. For each of these APIs, we determined the number of Android security related Stack Overflow posts that discussed the API. We discovered that 18K of the 26K APIs are not discussed in any Android security related posts. Table 7 shows a five-number summary of the 8K APIs that are discussed by at least one Android security related Stack Overflow post. The numbers suggest that approximately 2K APIs (25%) are discussed by at least 52 posts. Moreover, approximately 300 APIs are discussed in more than 1000 posts.

When we consider the package-prefixes of the 8K APIs deemed as security-related, there were 31 unique package-prefixes. From the perspective of package-prefixes, the benchmarks use APIs with only 19 of these package-prefixes; they do not use APIs with 12 of the 31 package-prefixes. We refer to these 19 package-prefixes as the *known package-prefixes* and the 12 package-prefixes as the *unknown package-prefixes.* Table 8 and Table 9 show that the number of APIs with *known package-prefixes* is more than the number of APIs with *unknown package-prefixes. Since 60% of the package-prefixes are known, the benchmarks are doing a good job of covering unique package-prefixes, i.e., posses breadth.* However, *the benchmarks are not using all security-related APIs in a known package-prefix, i.e., lack depth.*

Min	1Q	Median	3Q	Max
0.0002 (1)	0.0005 (2)	0.002 (9)	0.13 (52)	18.1 (69800)

Table 7: Five-Number summary of 318K security-related posts discussing APIs used in real-world apps but not in any benchmark suite.

Package-prefix	# APIs
android.app	1041
android.media	772
android.content	741
android.net	530
android.os	436

Table 8: Top 5 known package-prefixes ranked as per the number of APIs with a package-prefix

Package-prefix	# APIs
android.preference	197
android.renderscript	141
android.nfc	133
android.service	92
android.speech	83

Table 9: Top 5 unknown package-prefixes ranked as per the number of APIs with a package-prefix

7.1 Suggestions to extend the benchmark suites

We wanted to know how many of the 8K APIs could be used to create new benchmarks. Consequently, we categorized the APIs based on their package-prefix to create 27 sets. We ignored 4 package-prefixes because they were related to UI and printing services that are not the focus of the benchmarks we are evaluating. Finally, we selected the top 10 APIs from each of the 27 sets based on the number of posts discussing an API to create a set of 270 APIs. We manually examined these APIs and discovered that 17 (6%) of these APIs can be used to create new benchmarks. These APIs are related to Android app features such as ICC, loading web content, communicating via Bluetooth, interacting with a database, crypto, account management, and low-level file management.

Following are two examples of these 17 APIs can lead to vulnerabilities and can be used to create new benchmarks.

- Prior effort has shown that misuse of *android.webkit.WebView.loadUrl(url)* can lead to vulnerability [11]. In this exercise, we discovered *android.webkit.WebView.loadUrl(url, headers)* which is a variant of *loadUrl(url)* with an additional input parameter. Given the functional similarities of these methods, *loadUrl(url, headers)* can lead to similar vulnerabilities as *loadUrl(url)*.
- The *android.system.Os.open(file, mode)* API is used to access/create a file in a particular mode. If an app X saves sensitive information in a file using this API in a mode that allows other apps to access the file, then a malicious app can access this file and steal sensitive information.

Considering we found 17 APIs out of 270 manually examined APIs to lead to vulnerabilities, *more APIs are likely to exist in the set of unexamined APIs could lead to vulnerabilities and be used to create new benchmarks.*

7.2 What about APIs with no discussions?

Out of curiosity, we explored the 18K APIs that did not appear in Android security related posts in Stack Overflow. Out of the 18K APIs, 4K were methods and the rest were fields. We examined a sample of these 4K methods.

We discovered that the method, *KeyGenParameterSpec.Builder.setAttestationChallenge(bytes)*, belonging to the package *android.security.keystore* is related to security. This method takes a collection of bytes as input and uses it to create an attestation challenge for a public key. An entity receiving the public key with the attestation challenge can use the challenge to verify if the public key was created in response to a specific request. However, if *bytes*, needed to create the attestation challenge, is null, then the public key created will be signed with a self-signed certificate or a dummy signature. A Public key signed with a self-signed certificate or a dummy certificate is known to be insecure since such a certificate cannot be authenticated.

This API is being used by 17 apps in our real-world app sample. These 17 apps are related to finance and have a cumulative download of at least 100 million. Clearly, this method/API is critical. Hence, we plan to create a benchmark involving this method and contribute it to Ghera.

We also examined methods of *DevicePolicyManager* class. These methods help apps control the security policies of a device such as disabling/enabling KeyGuard[4] among other things. Not all apps can use the *DevicePolicyManager*. Android requires apps to satisfy a particular constraint to be able to access the *DevicePolicyManager*. So, an app using the *DevicePolicyManager* must implement an *exported* Broadcast Receiver protected by a permisson that is granted only to system entities (system permission). The methods in *DevicePolicyManager* should only be accessed through the corresponding hooks in the Broadcast Receiver. For example, when the user changes the device's password the corresponding hook in the Broadcast Receiver will get invoked. The app can then implement its own logic to accept or reject the password.

While the Android documentation states that the Broadcast Receiver must be protected with a system permission, Android does not enforce it. Consequently, it is possible to implement the Broadcast Receiver without securing it with the required system permission. Hence, a malicious app without the required permission can use the Broadcast Receiver to control the device's security policies thus resulting in a privilege escalation attack. We plan to further explore these APIs.

The presence of security-related APIs in the set of 18K APIs suggests that not all security-related APIs are discussed by Android app developers on Stack Overflow. Therefore, *Stack Overflow should not be used as a comprehensive source for identifying security issues in Android apps.*

7.3 Discussion

The APIs used in the benchmarks are discussed much more than the APIs that are used in real-world apps but not in any benchmark; see *median* column in Table 7 and Table 4. Additionally, almost all the filtered APIs across benchmark suites are discussed in at least one Stack Overflow post related to Android security; see Table 5. On the contrary, 75% of the APIs used in real-world apps but not in any benchmark is not discussed in any Android security related post. Discussions around an API in the context of security implies that app developers not only use the API but also have less clarity about it. In that sense, *Android app vulnerability benchmarks are doing a good job of using security-related APIs that concern app developers.*

Caveat As discussed in Section 7.2, an API not being discussed in Android security related Stack Overflow posts could still be related to security. An API may not be discussed on Stack Overflow for various reasons such as 1) the API might not be directly used by app developers, 2) app developers are well aware of the security implication of the API and do not feel the need to discuss it, 3) app developers use the API but are not aware of its security implications, and 4) app developers discuss the API in another developer forum. If the APIs are not being deemed as security-related due to the third and the fourth reason, then the findings for RQ4 should be considered with caution as the 18K APIs used in real world apps but not in the benchmarks could be related to security. This aspect does not affect the results for RQ1, RQ2, and RQ3 as very few APIs used by the benchmark suite are not discussed by Stack Overflow posts.

8 Threats to Validity

This evaluation is based on the methods and metrics proposed by Ranganath and Mitra [15]. Therefore, the threats to validity applicable to their effort applies to our effort as well. According to Ranganath and Mitra, API usage is a weak measure of possibility of presence of vulnerabilities as it ignores the influence of richer

[4] KeyGuard controls the lock screen of a mobile device.

but hard to measure aspects such as API usage context, security considerations of data, and data/control flow path connecting the various uses of API. Consequently, the influence of such aspects on measuring representativeness needs to be verified.

The baseline app from which the APIs to be ignored for representativeness calculation was determined can introduce bias based on how the baseline app was created. The effect of this bias can be measured and mitigated by using a different baseline app.

We identified the set of security-related APIs from the set of relevant APIs using Stack Overflow data. Specifically, we used the occurrence of APIs in posts to associate posts to APIs. Since an occurrence of an API does not always imply the discussion of the API, the reported numbers of posts discussing an API may be inflated. This threat can be addressed by using richer search techniques to associate posts to APIs. Similarly, we used tags associated with posts to identify Android security related posts, and this identification can be inaccurate due to incorrect tagging of posts. This threat can be addressed by using information retrieval techniques to identify incorrect tagging.

9 Related Work

While there has been considerable interest in developing solutions for detecting vulnerabilities in Android apps, very few efforts are focused on developing and measuring benchmarks used to evaluate the effectiveness of the solutions. Recently, Pauck et al. [13] developed a framework for verifying the authenticity of benchmarks in DroidBench and IccBench and refining them. In a similar vein, Qiu et al. [14] discovered that a few benchmarks in DroidBench and IccBench captured multiple aspects, which made their evaluation of the effectiveness of static taint analysis tools difficult. Therefore, they developed UBCBench and used it in conjunction with DroidBench and IccBench in their evaluation. In contrast, we focus on measuring the representativeness of benchmark suites along with comparing them and identifying gaps in them. Ranganath and Mitra [15] performed a similar evaluation. However, their evaluation was limited to measuring the representativeness of Ghera benchmarks. Moreover, our approach of identifying security-related APIs differs from theirs as we used data from Stack Overflow to identify APIs while they manually identified APIs.

Other efforts have used API usage to categorize vulnerabilities affecting Android apps. For example, Gorla et al. [5] used an app's description from app markets to infer its *advertised* behavior and the APIs used by the app to determine any anomalies. Similarly, Sadeghi et al. [16] measured the likelihood of a vulnerability pattern occurring in an Android app based on the app's source code patterns and API usage patterns. Distinct from such evaluations, the goal of our effort is to study benchmark suites and not Android apps or solutions related to Android app security.

Stack Overflow has been used in the past as a source for understanding security issues in Android apps. Stevens et al. [19] used Stack Overflow to study the relationship between the popularity of a *permission* and the number of times a permission is overused in an app. Similarly, Vasquez et al. [10] used Stack Overflow posts related to mobile development to understand the issues discussed by mobile app developers. In a similar vein, we also used Stack Overflow to identify security-related APIs. However, Stack Overflow data was used as an enabler and was not the focus of our evaluation.

10 Evaluation Artifacts

We used the version/bundle of DroidBench and IccBench benchmarks available under *DroidBench (extended)* (MD5Sum 9a165494eec309ff49f1b72895308a13) and *ICC-Bench 2.0* (MD5Sum: d479d07c94a9415868b420c1f289a0b2) sections at `https://github.com/FoelliX/ReproDroid`. The version of UBCBench we used is available at `https://github.com/LinaQiu/UBCBench`. The version of Ghera we used is available at `https://bitbucket.org/secure-it-i/android-app-vulnerability-benchmarks/src/Jan2019/`.

The raw and processed data from the experiment along with supporting scripts are available at `https://bitbucket.org/secure-it-i/evaluate-repr-droidbench-jan2019/src/ASE-2019/` for DroidBench, `https://bitbucket.org/secure-it-i/evaluate-repr-ghera-jan2019/src/ASE-2019/` for Ghera, `https://bitbucket.org/secure-it-i/evaluate-repr-iccbench-jan2019/src/ASE2019/` for IccBench, `https://bitbucket.org/secure-it-i/evaluate-repr-ubcbench-apr2019/src/ASE-2019/` for UBCBench, `https://bitbucket.`

org/secure-it-i/prepare-benchmarks/src/ASE-2019/ for benchmark preparation, and https://bitbucket.org/secure-it-i/stackoverflow-march2019/src/ASE-2019/ for Stack Overflow.

11 Future Work

Given the increasing focus on support for securing Android apps, here are few ways to improve benchmark suites and, consequently, help improve Android security analysis tools.

1. Explore richer aspects of real world apps such as call graphs, memory profiles, and API usage contexts to develop richer metrics to measure the representativeness of benchmarks.

2. Explore other metrics for indentifying *relevant* and *security-related* APIs.

3. Explore the APIs not covered by considered benchmark suites to develop and extend benchmark suites.

12 Summary

In this paper, we set out to understand how well do existing Android app vulnerability benchmark suites represent real world apps in terms of the manifestation of vulnerabilities. We considered DroidBench, Ghera, IccBench, and UBCBench benchmark suites and used API usage as a metric to measure representativeness.

We discovered that these benchmark suites are representative of real-world apps. Based on considered metrics, Droidbench was the most representative benchmark suite followed by Ghera, UBCBench, and IccBench. However, based on exploration of real-world APIs, we discovered that the benchmark suites are not comprehensive and they can be extending with new benchmarks. Finally, in the context of tool evaluations, the results suggest that DroidBench and Ghera should be considered equally but IccBench and UBCBench could be used to complement/strengthen the evaluations that use DroidBench and Ghera.

As an aside, we discovered that the tags associated with Stack Overflow posts are not good markers to identify posts that are likely related to security in Android apps.

References

[1] Kevin Allix, Tegawéndé F. Bissyande, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 468–471. ACM, 2016.

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 259–269. ACM, 2014. https://github.com/secure-software-engineering/FlowDroid, Accessed: 21-Nov-2017.

[3] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. *SIGPLAN Not.*, pages 169–190, 2006.

[4] Stefano Calzavara, Ilya Grishchenko, and Matteo Maffei. Horndroid: Practical and sound static analysis of android applications by SMT solving. In *2016 IEEE European Symposium on Security and Privacy*, pages 47–62, 2016. https://github.com/ylya/horndroid, Accessed: 05-May-2018.

[5] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1025–1035. ACM, 2014.

[6] Jim Gray, editor. *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993.

[7] Google Inc. Shrink code and resources with ProGuard. https://developer.android.com/studio/build/shrink-code, 2017. Accessed: 17-Jan-2019.

[8] C. Isen, L. John, Jung Pil Choi, and Hyo Jung Song. On the representativeness of embedded java benchmarks. In *2008 IEEE International Symposium on Workload Characterization*, pages 153–162, 2008.

[9] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, pages 280–291, Piscataway, NJ, USA, 2015. IEEE Press. https://github.com/lilicoding/soot-infoflow-android-iccta, Accessed: 05-May-2018.

[10] M. Linares-Vasquez, B. Dit, and D. Poshyvanyk. An exploratory analysis of mobile development issues using stack overflow. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 93–96, 2013.

[11] Joydeep Mitra and Venkatesh-Prasad Ranganath. Ghera: A repository of android app vulnerability benchmarks. In *International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*, pages 43–52. ACM, November 2017. https://bitbucket.org/secure-it-i/android-app-vulnerability-benchmarks/, Accessed: 21-Nov-2017.

[12] James Pallister, Simon J. Hollis, and Jeremy Bennett. Beebs: Open benchmarks for energy measurements on embedded platforms. *CoRR*, abs/1308.5174, 2013.

[13] Felix Pauck, Eric Bodden, and Heike Wehrheim. Do android taint analysis tools keep their promises? In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 331–341. ACM, 2018. https://foellix.github.io/ReproDroid/.

[14] Lina Qiu, Yingying Wang, and Julia Rubin. Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 176–186. ACM, 2018. https://github.com/LinaQiu/UBCBench.

[15] Venkatesh-Prasad Ranganath and Joydeep Mitra. Are free android app security analysis tools effective in detecting known vulnerabilities? *Empirical Software Engineering*, 2019.

[16] Alireza Sadeghi, Naeem Esfahani, and Sam Malek. Mining mobile app markets for prioritization of security assessment effort. In *Proceedings of the 2Nd ACM SIGSOFT International Workshop on App Market Analytics*, WAMA 2017, pages 1–7. ACM, 2017.

[17] SPEC. SPECJava Benchmarks. https://www.spec.org/benchmarks.html, 2006. Accessed: 17-Jan-2019.

[18] Stack Overflow. Stack Overflow Archives. https://archive.org/download/stackexchange, 2019. Accessed: 11-May-2019.

[19] Ryan Stevens, Jonathan Ganz, Vladimir Filkov, Premkumar Devanbu, and Hao Chen. Asking for (and about) permissions used by android apps. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 31–40. IEEE Press, 2013.

[20] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1341. ACM, 2014. http://pag.arguslab.org/argus-saf, Accessed: 05-May-2018.