BenchPress: Analyzing Android App Vulnerability Benchmark Suites

Joydeep Mitra Venkatesh-Prasad Ranganath
Kansas State University, USA
{joydeep,rvprasad}@k-state.edu

January 29, 2019

Abstract

In recent years, various efforts have designed and developed benchmark suites to evaluate the efficacy of vulnerability detection tools in Android apps. The choice of benchmark suites used in tool evaluations is often based on availability and popularity of suites instead of on the characteristics and relevance of benchmark suites relative to real world native Android apps. One of the reasons for such choice is the lack of information about characteristics and relevance of benchmarks suites relative to real world apps.

In this paper, we report the findings from our effort aimed at addressing this gap. We empirically evaluated three Android specific benchmark suites – DroidBench, Ghera, and IccBench. For each suite, we report how well do these benchmark suites represent real world apps in terms of API usage: 1) coverage – how often are the APIs used in a benchmark suite used in a sample of real world native Android apps? and 2) gap – which of the APIs used in a sample of real world native Android apps are not used in any benchmark suite? Based on pairwise comparison, we also report how these suites fare relative to each other in terms of API usage.

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 specific benchmark creators improve API usage based representativeness of suites (informed by gaps).

1 Introduction

1.1 Motivation

Effectiveness of Android security analysis tools is evaluated with benchmarks and real world apps. For example, the effectiveness of static taint analysis tools like AmanDroid [22], FlowDroid [2], HornDroid [7], and IccTA [14] has been evaluated by applying them to benchmarks from DroidBench [8], IccBench [21], and UBCBench [18] benchmark suites and comparing tool verdicts with benchmark labels that indicate the presence/absence of specific vulnerability or malicious behavior. The effectiveness of tools like Covert [3] and MalloDroid [9] has been evaluated by applying them to real world native Android apps and manually examining the apps to confirm the veracity of tool verdicts as the presence/absence of vulnerabilities or malicious behaviors in real world apps is unknown a priori.

Such tool evaluations have used benchmarks without evaluating the authenticity and 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 [15]). *Representativeness* is the manifestation/occurrence of a vulnerability in a benchmark being representative of the manifestation/occurrence of the vulnerability in real world apps (Section 3 in [19]). Consequently, the usefulness of findings from these evaluations is reduced in terms of the basic ability of tools and techniques to detect vulnerabilities or malicious behavior (related to authenticity) and the general applicability of tools and techniques (related to representativeness).

Recently, there have been two efforts focused on the authenticity of benchmarks. Mitra and Ranganath [15] created Ghera, a suite of demonstrably authentic Android app vulnerability benchmarks. Pauck et al. [17] developed a tool called ReproDroid to help verify the authenticity of benchmarks. They found the claims

about presence/absence of vulnerabilities in numerous benchmarks in DIALDroid [5], DroidBench [8], and IccBench [21] benchmark suites were untrue.

In the space of representativeness, Ranganath and Mitra [19] recently measured the representativeness of Ghera benchmarks [15] in an effort to empirically evaluate 14 vulnerability detection tools and 5 malicious behavior detection tools related to Android apps.

Furthermore, it is common in other communities to study and characterize benchmarks. In the program analysis community, BlackBurn et al. [4] developed and used metrics based on static and dynamic properties of programs to characterize and compare the DaCapo benchmarks with SPEC Java benchmarks [20]. Isen et al. [13] measured several properties of embedded Java benchmarks and how well they represent real world mobile apps. In the systems community, Pallister et al. [16] extensively studied embedded benchmarks from multiple benchmark suites and characterized them based on their energy consumption properties. In database community, such assessments have been around since 1990s [11].

Drawing inspiration from efforts that have empirically evaluated benchmark suites and motivated by recent efforts that have explored the authenticity and representativeness of Android app vulnerability benchmarks, we undertook an effort to assess and compare the representativeness of multiple Android app vulnerability benchmark suites.

1.2 Research Questions

Our effort aims to answer the following research questions:

- **RQ1** *Are Android app vulnerability benchmark suites representative of real world native Android apps in terms of the use of APIs involved in or related to the vulnerabilities captured by benchmarks?* To answer this question, we compare the use of Android programming APIs (including XML attributes and features of configuration files) in the benchmarks of a benchmark suite with the use of the same APIs in a sample of 88K real world native Android apps.

- **RQ2** *Of the APIs used in real world native Android apps, which APIs are not used in benchmarks?* To answer this question, we identify the APIs used in a sample of 88K real world native Android apps but not used in any benchmark.

- **RQ3** *How do the benchmark suites compare with each other in terms of representativeness and applicability to tool evaluations?* To answer this question, we perform a pairwise comparison between benchmark suites in terms of API usage to gauge the similarities and differences.

1.3 Contributions

In this paper, we make the following contributions:

- Provide empirical evidence about the (extent of) representativeness of Android app vulnerability benchmark suites. This information can help associate an element of confidence with benchmarks and, consequently, with tool evaluations that use them.

- Identify gaps between existing benchmark suites and real world apps in terms of APIs that are used in real world apps but not used in benchmarks. This information can help security analysis efforts to focus on unexplored parts of Android programming APIs to possibly uncover new vulnerabilities and enhance existing benchmark suites.

- Identify differences between benchmark suites in terms of representativeness based on API usage. This information can help tool developers choose appropriate benchmark suites to test/evaluate their tools.

In addition to these contributions, we hope this effort will spark the interest of 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 study including the used benchmark suites the sample of real world apps, and the metric used to measure representativeness. Section 3 describes the experiment to measure representativeness. Sections 4-6 discuss the answers to posed research

questions. Section 7 describes the threats to the validity of the experiment. Section 8 provides information about the artefacts used in this effort. Section 9 list few possibilities to extend and build on this effort. Section 10 summarizes the findings from this effort.

2 Study Approach

2.1 Using API usage as a measure of representativeness

Most benchmarks related to Android app vulnerabilities capture vulnerabilities that result from the use of APIs in different contexts. In some cases, benchmarks can capture vulnerabilities that stem from factors not related to APIs, e.g., an app using an outdated library. Since all but two (from Ghera) benchmarks used in this evaluation capture vulnerabilities that result from the contextual use of APIs, representativeness of these benchmarks can be measured by comparing the usage of APIs used to capture a vulnerability in a benchmark with the usage of the same APIs in real world native Android apps. This notion of API usage based representativeness was introduced by Ranganath and Mitra in Section III of their paper [19] and we use this notion in our evaluation.

2.2 Benchmarks

For this study, we considered 3 benchmark suites – *DroidBench, Ghera, and IccBench*.

DroidBench [8] contains 211 benchmarks. Each benchmark is an Android app that either captures a specific information leak vulnerabilitiy x or does not capture x.

Ghera [15] contains 56 benchmarks that capture mostly known Android app vulnerabilities along with few unknown Android app vulnerabilities. Each benchmark contains 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 in each benchmarks (and refer to them as Ghera benchmarks in the rest of this paper).

IccBench [21] contains 25 benchmarks. Each benchmark is an Android app that either captures a specific information leak vulnerabilitiy x or does not capture x.

Both DroidBench and IccBench contain benchmarks that focus only on information leak vulnerabilities stemming from Inter-Component Communication (ICC) both within and in-between Android apps.

2.3 Real World Apps

We started with the set of real world native Android apps used by Ranganath and Mitra [19] from AndroZoo [1] in April 2018. This set contained 105K apps that target API level 19-27. API level is an integer value that uniquely identifies the framework API revision offered by a version of the Android platform. *Minimum API level* is the least framework API version that an app can run on and *target API level* is the framework API version that an app targets.

From this set, we ignored apps with target API level 20 because it corresponds to Android wearables. We also ignored apps with target API level 21 because none of the benchmarks considered in our evaluation target API level 21. Hence, we ended up with a sample of 88K real world native Android apps.

Table 1 shows the target API level distribution in our sample of real world apps. The sample size of real world apps that target more recent API levels is lower than the number of real world apps that target older API levels. This skew is to be expected since older API levels have existed for longer times.

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 is not provided by the underlying Android framework or runtime. However, as a result of the basic build process of Android apps, the APKs may contain *unnecessary* code and resources.

Target API level	# Real World Apps
19	9604
22	23394
23	47579
24	4673
25	2260
26	481
27	122
Total	88113

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

Consequently, ProGuard [12] tool is used as part of Android app build process to eliminate *unnecessary* bits while building release version APKs.

While both DroidBench and IccBench provide APKs and source files for each of their benchmarks, the provided APKs contain unnecessary code and resources. To control for the effects of unnecessary APIs on the findings of this evaluation, we rebuilt the APKs of these benchmarks from source using ProGuard.

When building an Android app from its source, both minimum API level and target API level need to be specified. All DroidBench benchmarks have minimum API level 8 and target API level 14-24. Since most of these benchmarks target API level 19 and API level 19 is relatively recent, we rebuilt all DroidBench benchmarks with minimum API level 8 and target API level 19. This helps control the effect of API level on evaluation of representativeness (while considering recency of API levels).

While rebuilding IccBench benchmarks, we used API level 25 as both minimum and target API levels because all of these benchmarks had minimum and target API level 25.

The APKs for Ghera benchmarks were used as is since they were built using *ProGuard* with minimum API level 22 and target API level 27.

3.2 API-based App Profiling

Every Android app is bundled as an APK file that 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 used the method used by Ranganath and Mitra [19] to determine the APIs used by the sample of real world native Android apps and by the benchmarks.

In this method, for each benchmark, we considered an API if it was used but not defined in the app. We ignored the APIs that were obfuscated, i.e., APIs with single character names. Further, to make apps comparable, for overridden methods and fields, we used class hierarchy analysis to consider the overridden methods and fields as opposed to the overriding methods and fields. Of these APIs, we only recorded APIs whose Fully Qualified Name (FQN) contained the prefix *android, com.android, java,* and *org* because we wanted to measure representativeness in terms of Android APIs.

Numerous APIs are commonly used in almost all Android apps and are related to aspects (e.g., UI rendering) that are not the focus of vulnerability benchmarks related to Android apps. To avoid their influence on the evaluation, such APIs are ignored by the method while determining the APIs used by apps and benchmarks. For this purpose, we created a baseline app that exhibited no interesting functionality but contained graphical widgets commonly used by Android apps. To account for API level specific common Android APIs used in each benchmark suites, we created one dedicated activity corresponding to each benchmark suites in the baseline app. Out of the 1878 APIs used in this baseline app, we manually identified 1669 APIs as commonly used in Android apps; almost all of them were basic Java APIs or related to UI rendering and XML processing. For each benchmark suite, we ignored these 1669 APIs from the list of recorded APIs (from the previous step) to create the set of *relevant* APIs. From each set of relevant APIs, we then identified a set of *security-related* APIs that we deemed can influence the security of apps.

Repo	Number of APIs		
	Total	Relevant	Security-Related
DroidBench	1134	569	82
Ghera	1864	469	154
IccBench	186	101	30

Table 2: Number of APIs used by the benchmark suites

Table 2 shows the total number of APIs, number of *relevant* APIs, and the *security-related* APIs for each benchmark suite.

3.3 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 also did this for each security related API corresponding to the benchmark suite. Since the benchmark suites use different API levels, we calculated this percentage based on the number of real world apps targeting API level 19, API levels 22-27, and API level 25 for DroidBench, Ghera, and IccBench, respectively.[1]

3.4 Examining Gap between Real World Apps and Benchmarks

Of the 167K APIs used by the apps in our real world app sample, we ignored APIs also used by all benchmarks in DroidBench, Ghera, and IccBench. Of the remaining 165K APIs, we ignored APIs related to UI and third party libraries. The remaining 25K APIs were spread across 45 packages with prefixes *android, java,* or *javax*. We refer to this set of 25K APIs as *unexplored API set*. Of the 45 packages, we selected the five packages that were most used by the apps in our real world app sample. From these 5 packages, we manually selected and examined 9,747 APIs to determine if they could be used to create new benchmarks.

4 Answering RQ1

Are Android app vulnerability benchmark suites representative of real world native Android apps in terms of the use of APIs involved in or related to the vulnerabilities captured by benchmarks?

The graphs in FIGURE 1, 2 and 3 show the frequency of real world apps using both relevant and security-related APIs found in Droidbench, Ghera, and IccBench benchmark suites, respectively, in the decreasing order of frequency. The horizontal dashed line marks the 50% usage and the vertical dashed line marks the point on the x-axis that corresponds to 50% usage.

DroidBench As seen in the graphs in FIGURE 1, all of the 569 relevant APIs used in DroidBench benchmarks are used by some real world app. Of these APIs, 357 (63%) APIs are used by more than 50% of real world apps. Further, all of the 82 security-related APIs used in DroidBench benchmarks are used by some real world app and 49 (60%) of these APIs are used by more than 50% of the real world apps.

From the above numbers, we conclude *DroidBench is highly representative of real world native Android apps in terms of API usage*. Since DroidBench is focused on capturing information flow vulnerabilities related to ICC, this conclusion seems intriguing at first. However, it is explained by two reasons. First, ICC is the most common method for both intra- and inter-app communication in Android; hence, many real world apps are highly likely to use ICC related APIs. Second, 105 relevant APIs used in DroidBench are from commonly used packages – *java.io*, *java.lang*, *java.net*, *java.security*, and *java.util* – from the Java standard library. Since Android apps in the considered app sample are Java apps, they are highly likely to use these APIs.

[1]While Ranganath and Mitra [19] measured representativeness using API usage percentage and sampling proportions, both measurements led to similar observations. Further, measuring using API usage percentage is easy. So, we chose to measure representativeness using only API usage percentage.

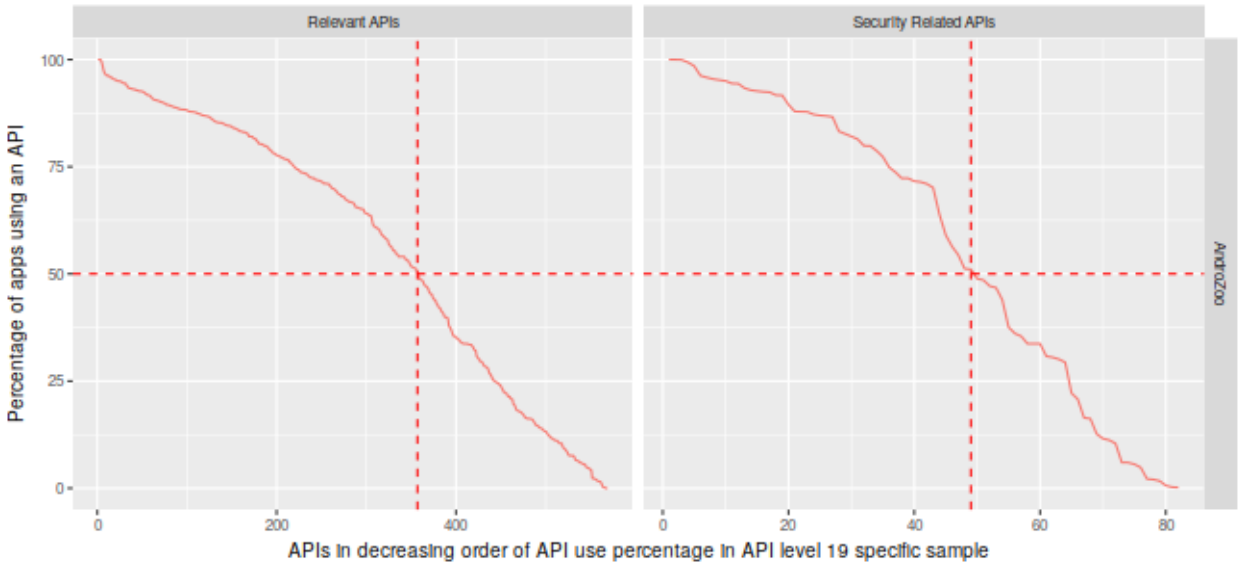

FIGURE 1: Plot showing representativeness of DroidBench benchmarks

Ghera As seen in the graphs in FIGURE 2, all of the 469 relevant APIs used by Ghera benchmarks are used by some real world app. Of these APIs, 339 (72%) APIs are used by more than 50% of real world apps. Further, all of the 154 security-related APIs are used by at least some real world app and 92 (60%) of these APIs are used by more than 50% of the real world apps.

From the above numbers, we conclude *Ghera is highly representative of real world native Android apps in terms of API usage*. Unlike in the case of DroidBench, this conclusion is obvious as Ghera captures vulnerabilities related to different features of the Android framework – cryptography, ICC, networking, permission, storage, system, and web.

IccBench As seen in the graphs in FIGURE 3, out of the 101 relevant APIs used in IccBench, 71 are used by some real world app. Further, all 30 security-related APIs are used by some real world app and 21 (70%) of these APIs are used by more than 50% of the real world apps.

While IccBench uses APIs that are used by real world apps, it also uses 30 APIs that are not used by any app in the sample of 88K real world apps (including the apps that target API level 25). Thus, *IccBench is not highly representative in terms of API usage.* IccBench contains 5 benchmarks which focus on inter-process communication (IPC) using the Android Interface Definition Language (AIDL). The 30 relevant APIs that are not used by any real world app are all related to IPC via AIDL. This suggests that IccBench benchmarks cover niche aspects that are not prevalently used in real world apps.

5 Answering RQ2

Of the APIs used in real world native Android apps, which APIs are not used in benchmarks?

The *unexplored API set* of 25K APIs is spread across 45 packages. Of these 45 packages, none of the APIs in 27 packages are used by any of the considered benchmark suites. We refer to these 27 packages as the *unexplored package set*. In the remaining 18 packages, some (not all) APIs have been been used by some of the considered benchmark suite. We refer to these 18 packages as the *partially explored packages*. From the set of *partially explored packages*, we identified the top five packages based on the number of real world apps that use an API in a package. These packages enable capabilities/features such as displaying web content, managing network connections, managing media related content, performing inter-component communication (ICC), managing device policy configurations, and managing storage. Table 3 shows the distribution of the top five partially explored packages. Of the 9,747 APIs in these five packages, we identified 1,736 APIs that are related to app security.

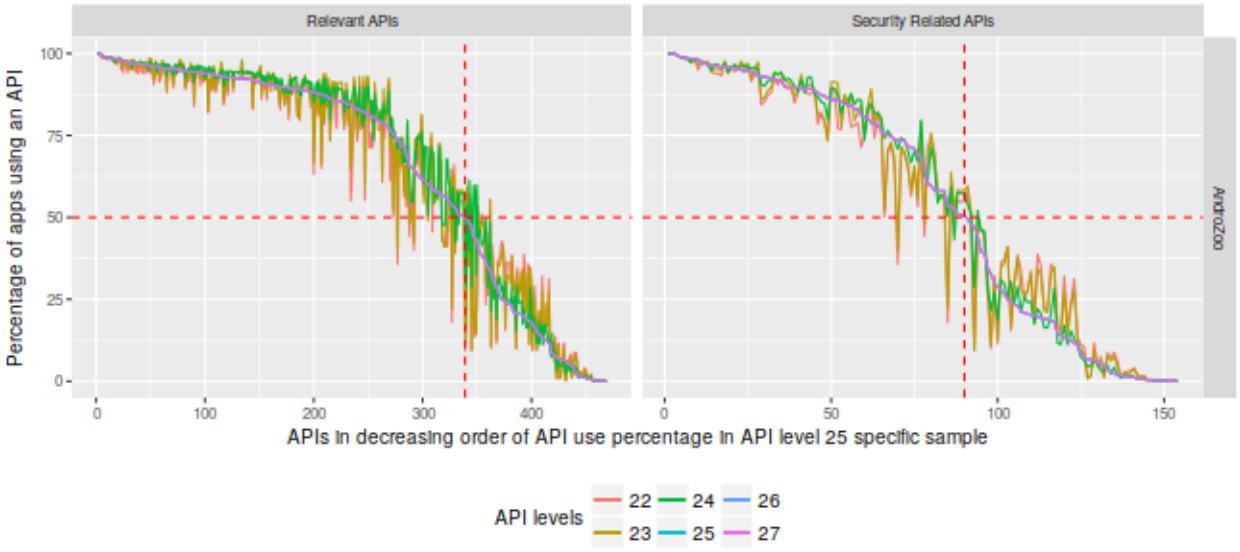

FIGURE 2: Plot showing representativeness of Ghera benchmarks

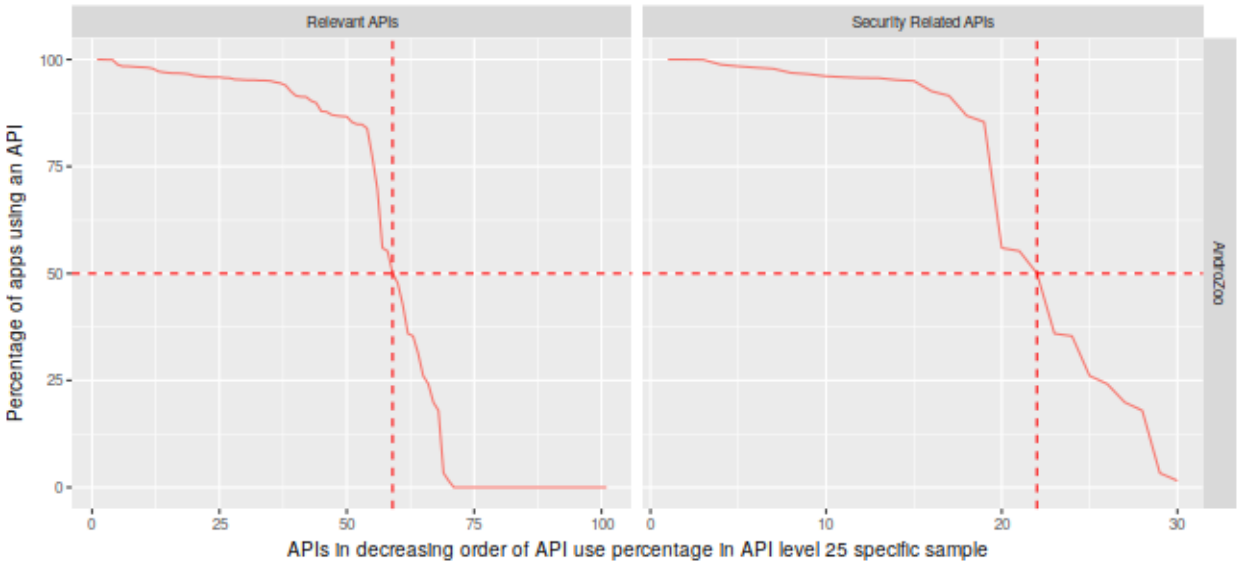

FIGURE 3: Plot showing representativeness of IccBench benchmarks

	# APIs		
Package	Total	Security-Related	# Real World Apps
android.app	2625	125	87K
android.content	2175	102	86K
android.media	2129	316	83K
android.net	1511	1029	86K
android.webkit	164	1307	84K
Total	9747	1736	

Table 3: Information about explored packages in terms of the total number of APIs and the number of security-related APIs in a package and the number of real world apps that use an API from a package.

Package	# Total APIs	# Real World Apps
android.accounts	270	83K
android.speech	171	66K
android.preference	424	64K
android.renderscript	989	42K
android.security	83	40K

Table 4: Information about unexplored packages in terms of the number APIs in a package and the number of real world apps that use an API from a package.

We selected 3 of the 1,736 APIs and studied them by examining their documentation.

- For `android.webkit.WebView.loadData` and `android.webkit.WebView.loadDataWithBase URL` methods, the official Android documentation warns against certain uses of these APIs as they may lead to vulnerabilities.
- The `android.webkit.WebSettings.setAllow ContentAccess` API is used to set a flag in custom-built browsers. If the flag is set to true, then the custom-built browser is allowed to access any Content Provider[2] [10] on the phone. If a custom-built browser loads JavaScript code from an untrusted location, then a man-in-the-middle could inject malicious JavaScript code to access any Content Provider and possibly access sensitive information. We plan to develop a benchmark based on *setAllowContentAccess* API and contribute it to Ghera.

Since 1,736 of the 9,747 examined APIs were security-related (and mere examination of documentation helped us uncover a potentially new benchmark), security-related APIs are highly likely to be present in the remaining 13 partially explored packages and few of them are likely to be associated with vulnerabilities not captured by any of the considered benchmark suites. Consequently, *existing benchmarks may be missing out on capturing vulnerabilities related to APIs in the partially explored packages.*

Unexplored packages contain APIs that correspond to features such as user account management, use of camera, NFC (Near Frequency Communication), and speech. Table 4 shows the distribution of the top five unexplored packages based on the number of real world apps that use an API in a package. The data in the table clearly shows a large number of real world apps use APIs from unexplored packages. Therefore, *these packages may serve as a good source of security-related APIs that could lead to the discovery of new vulnerability benchmarks and even new vulnerabilities.*

6 Answering RQ3

How do the benchmark suites compare against each other in terms of representativeness and applicability to tool evaluations?

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

For each pairwise comparison, Table 5 shows the number of relevant APIs common and unique to the compared benchmarks. Table 6 shows the similar data for security-related APIs.

[2]Content Provider is an Android component that stores and manages app data.

Benchmark Suite Pair (X-Y)	Common APIs	APIs unique to X	APIs unique to Y
DroidBench-Ghera	225	344	244
DroidBench-ICCBench	55	514	46
IccBench-Ghera	40	61	429

Table 5: Information about Relevant APIs from Pairwise Comparison

Benchmark Suite Pair (X-Y)	Common APIs	APIs unique to X	APIs unique to Y
DroidBench-Ghera	62	20	92
DroidBench-ICCBench	25	57	5
IccBench-Ghera	27	3	127

Table 6: Information about Security-Related APIs from Pairwise Comparison

6.1 DroidBench vs Ghera

Observation 1 As shown in Table 2, *DroidBench uses fewer APIs in total but contains more apps than Ghera* – 211 benchmarks in DroidBench use 1,134 APIs and 56 benchmarks in Ghera use 1,864 APIs. This is due to two reasons. First, Ghera benchmarks use more APIs (e.g., UI related APIs) that are commonly used by all Android apps (possibly to be more complete or similar to real world apps) while DroidBench benchmarks use fewer such APIs (possibly to be minimal). Second, DroidBench captures different manifestations of a vulnerability involving similar APIs (e.g., multiple benchmarks capture different ways of sensitive information flowing into Android system log) while every Ghera benchmark captures a unique vulnerability. This observation suggests that, *if a tool can detect vulnerability* x *that is captured by both DroidBench and Ghera, then testing the tool using DroidBench would be better as the tool will be tested against different manifestations of* x.

Observtion 2 As shown in Table 2, *Ghera uses 100 fewer relevant APIs than DroidBench.* As observed in the previous observation, Ghera benchmarks used more APIs that are commonly used by all Android apps – Ghera uses 881 UI related APIs while DroidBench uses 367 UI related APIs. Since these commonly used APIs are not considered as relevant, Ghera has fewer relevant APIs than DroidBench. So, *in terms of relevant API usage, Ghera benchmarks are more minimal than DroidBench benchmarks.* Since minimality helps ensure the benchmarks are focused (i.e., contain one vulnerability), Ghera benchmarks are likely to ease testing of tools and possibly ease comprehension of vulnerabilities.

Observation 3 As seen in Table 5, Ghera uses 244 relevant APIs that are not used in DroidBench. Of these 244 APIs, 194 (79%) APIs are related to Android features (e.g., cryptography, networking, storage, web) that are either minimally targeted or not targeted by DroidBench. Similar observation can be made from Table 6. Specifically, Ghera uses 92 security-related APIs that are not used in DroidBench. Consequently, *Ghera should be preferred over DroidBench when evaluating tools that focus on detecting vulnerabilities that stem from the use of different Android features.*

Observation 4 As seen in Table 5, DroidBench uses 344 APIs that are not used in Ghera and 168 (50%) of these APIs are related to ICC. Hence, *DroidBench should be preferred over Ghera when evaluating tools that focus on detecting information flow vulnerabilities that stem from the use of ICC.*

Observation 5 Ghera targets API levels 22-27 whereas DroidBench largely targets API levels 19 or less. Consequently, some of the vulnerabilities captured by DroidBench cannot be exploited in newer API Levels of Android, e.g., starting a service via implicit intent, sensitive information flowing into a log file. Further, Ghera benchmarks uses a large number of security-related APIs that are used by DroidBench benchmarks while also using 92 security-related APIs that are not used by DroidBench benchmarks. So, *Ghera should be preferred over DroidBench when evaluating the ability of tools to detect current vulnerabilities.* However,

for tools that are built to detect vulnerabilities in apps that target API level 19 or less, either DroidBench or an older version of Ghera that targets API levels 19-25 [15] should be considered.

6.2 DroidBench vs IccBench

Observtion 6 As seen in Table 5, out of the 101 relevant APIs used in IccBench, 55 APIs are also used in DroidBench. Of the remaining 46 APIs, 30 APIs are not used by any app in our real world app sample. Also, DroidBench uses 514 relevant APIs that are not used in IccBench. Further, as seen in Table 6, DroidBench uses 57 security-related APIs not used in IccBench while also using all but five of the security-related APIs used in IccBench. So, *in general, DroidBench should be preferred over IccBench.*

6.3 Ghera vs IccBench

Observation 7 As seen in Table 5, out of the 101 relevant APIs used in IccBench, 40 APIs are also used in Ghera. Of the remaining 60 APIs, 30 APIs are not used by any app in our real world app sample. Further, as seen in Table 6, Ghera uses 127 security-related APIs not used in IccBench while also using all but three of the security-related APIs used in IccBench. So, *in general, Ghera should be strongly preferred over IccBench.*

7 Threats to Validity

This evaluation is based on the methods and metrics proposed by Ranganath and Mitra [19]. 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 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.

As observed in Table 1, the distribution of real world apps across target API levels 19-27 is skewed. This could have affected the representativeness measured for those API levels. This threat can be verified by repeating the experiment with a larger or a more recent sample of real world apps.

All DroidBench benchmarks were built with minimum API Level 8 and target API Level 19. Building the benchmarks in DroidBench with different minimum and target API Level might have affected the reported results. This threat can be verified by repeating the experiment with DroidBench benchmarks built with different minimum and target API Levels.

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

We identified a set of security-related APIs from the set of relevant APIs based on our knowledge of Android and the considered benchmark suites. This identification is subject to our bias and understanding of Android and the benchmark suites. While this subjectivity is hard to eliminate, the extent of its influence can be measured by others researchers verifying the artifacts from our evaluation or repeating the evaluation.

8 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 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/Jan2019/ for DroidBench, https://bitbucket.org/secure-it-i/evaluate-repr-ghera-jan2019/src/Jan2019/ for Ghera, and https://bitbucket.org/secure-it-i/evaluate-repr-iccbench-jan2019/src/Jan2019/ for IccBench.

9 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.

- 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.
- Explore the APIs not covered by considered benchmark suites to extend these and other benchmark suites.
- Replicate this evaluation with other benchmarks e.g., DIALDroid [6], UBCBench [18].
- Replicate this evaluation with a more recent sample of real world apps that includes more apps targeting API levels 26 and 27.

10 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, and IccBench benchmark suites and used API usage as a metric to measure representativeness. Of the considered suites, Ghera was most representative of real world apps followed by DroidBench and IccBench. The results also suggest there may be an easy and worthwhile opportunity to extend existing benchmark suites by merely exploring APIs that are used by real world apps but not by existing benchmark suites. Further, in the context of tool evaluation, the results suggest, while DroidBench and IccBench should be preferred in specific situations (e.g., tools targeting older Android versions or ICC related vulnerabilities), Ghera should be preferred over DroidBench and IccBench in general. These results can help benchmark creators improve their benchmark suites and tool developers to choose appropriate benchmark suites to test/evaluate their tools.

We hope this effort will spark the interest of software mining community to develop techniques and methodologies to help evaluate and extend benchmark repositories in communities that will most benefit from rigor, e.g., Android app security.

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] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek. Covert: Compositional analysis of android inter-app permission leakage. *IEEE Transactions on Software Engineering*, pages 866–886, 2015. https://seal.ics.uci.edu/projects/covert/index.html, Accessed: 21-May-2018.

[4] 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.

[5] Amiangshu Bosu. DIALDroidBench. https://tool865110240.wordpress.com/dialdroidbench/, 2017. Accessed: 12-Sep-2018.

[6] Amiangshu Bosu, Fang Liu, Danfeng (Daphne) Yao, and Gang Wang. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 71–85. ACM, 2017.

[7] 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.

[8] DroidBench. DroidBench: A micro-benchmark suite to assess the stability of taint-analysis tools for Android. https://github.com/secure-software-engineering/DroidBench, 2013. Accessed: 01-June-2018.

[9] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why eve and mallory love android: An analysis of android ssl (in)security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 50–61. ACM, 2012. https://github.com/sfahl/mallodroid, Accessed: 15-Apr-2018.

[10] Google Inc. Content Providers: Official Android Documentation. https://developer.android.com/guide/topics/providers/content-providers, 2019. Accessed: 22-Jan-2019.

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

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

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] Venkatesh-Prasad Ranganath and Joydeep Mitra. Are free android app security analysis tools effective in detecting known vulnerabilities? *CoRR*, 2018.

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

[21] Fengguo Wei. ICC-Bench. https://github.com/fgwei/ICC-Bench, 2017. Accessed: 12-Sep-2018.

[22] 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.