Are Free Android App Security Analysis Tools Effective in Detecting Known Vulnerabilities?

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

June 30, 2018

Abstract

Increasing interest to secure Android ecosystem has spawned numerous efforts to assist app developers in building secure apps. These efforts have developed tools and techniques capable of detecting vulnerabilities and malicious behaviors in apps. However, there has been no evaluation of the effectiveness of these tools and techniques to detect known vulnerabilities. Absence of such evaluations poses as a hurdle when app developers try to choose security analysis tools to secure their apps.

In this regard, we evaluated the effectiveness of vulnerability (and malicious behavior) detection tools for Android apps. We considered 64 security analysis tools and empirically evaluated 19 of them — 14 vulnerability detection tools and 5 malicious behavior detection tools — against 42 known vulnerabilities captured by benchmarks in Ghera repository. Of the many observations from the evaluation, the key observation is *existing security analysis tools for Android apps are very limited in their ability to detect known vulnerabilities: all of the evaluated vulnerability detection tools together could only detect 30 of the 42 known vulnerabilities.*

Clearly, serious effort is required if security analysis tools are to help developers build secure apps. We hope the observations from this evaluation will help app developers choose appropriate security analysis tools and persuade tool developers and researchers to identify and address limitations in their tools and techniques. We also hope this evaluation will catalyze or spark a conversation in the Android security analysis community to require more rigorous and explicit evaluation of security analysis tools and techniques.

1 Introduction

1.1 Motivation

Mobile devices have become an integral part of living in present day society. They have access to huge amount of private and sensitive data about their users and, consequently, enable various services for their users such as banking, social networking, and even two-step authentication. Hence, securing mobile devices and apps that run on them is paramount.

In this context, with more than 2 billion Android devices in the world, securing Android devices, platform, and apps is crucial [64]. The task of securing Android devices and the Android platform is well tackled by few teams with access to relatively large pool of resources at companies such as Google and Samsung. The task of securing Android apps is split between app stores and app developers. While app stores focus on keeping malicious apps out of the ecosystem, app developers focus on eliminating vulnerabilities from apps to protect app users against exploits. Despite the best efforts of app stores, malicious apps can enter the ecosystem e.g., app installation from untrusted sources, inability to detect malicious behavior in apps, access of malicious websites. Hence, *there is a need for app developers to secure their apps.*

When developing apps, developers juggle with a multitude of aspects including app security. Most app development teams often cannot tend equally well to all such aspects as they are often strapped for resources. Hence, there is an acute need for automatic tools and techniques that can detect vulnerabilities in apps and, when possible, suggest fixes for identified vulnerabilities. While software development community

has recently realized the importance of security, developer awareness about how security issues transpire and how to avoid them is still lacking [48]. Hence, vulnerability detection tools need to be applicable off the shelf with no or minimal configuration.

In this context, numerous efforts have proposed techniques and developed tools to detect different vulnerabilities (and malicious behavior) in Android apps. Given the number of proposed techniques and available tools, there have been recent efforts to assess the capabilities of these tools and techniques [59, 61]. However, these assessments are subject to one or more of the following limiting factors:

1. Consider techniques only as reported in the literature, i.e., without executing associated tools.
2. Exercise a small number of tools.
3. Consider only academic tools.
4. Consider only tools that employ specific kind of underlying techniques, e.g., program analysis.
5. Rely on technique-specific micro benchmarks, e.g., benchmarks targeting the use of taint-flow analysis to detect information leaks.
6. Use random real world apps that are not guaranteed to be vulnerable.

The evaluations performed in efforts that propose new tools and techniques also suffer from such limitations. Specifically, such evaluations focus on proving the effectiveness of proposed tools and techniques in detecting specific vulnerabilities. While such focus is necessary and good, it is not sufficient as the evaluations do not consider the effectiveness of the tools and techniques in the context of previously known vulnerabilities. Hence, the results are limiting in their ability to help app developers choose appropriate tools and techniques.

In short, to the best of our knowledge, *there has been no evaluation of the effectiveness of Android app vulnerability detection tools to detect known vulnerabilities without being limited by any of the above factors.*

In addition to our growing dependence on mobile apps, the prevalence of Android platform, the importance of securing mobile apps, and the need for automatic easy-to-use off-the-shelf tools to build secure mobile apps, here are few more compelling reasons to evaluate the effectiveness of tools in detecting known vulnerabilities in Android apps.

1. To develop secure apps, app developers need to choose and use tools that are best equipped to detect the class of vulnerabilities that they believe (based on their intimate knowledge about their apps) will likely plague their apps, e.g., based on the APIs used in their apps. To make good choices, app developers need information about the effectiveness of tools in detecting various classes of vulnerabilities. Information about other aspects of tools such as performance, usability, and complexity can also be helpful in such decisions.
2. With the information that a tool cannot detect specific class of vulnerabilities, app developers can either choose to use a combination of tools to cover all or most of the vulnerabilities of interest or incorporate extra measures in their development process to help weed out vulnerabilities that cannot be detected by the chosen set of tools.
3. App developers would want to detect and prevent known vulnerabilities in their apps as the vulnerabilities, their impact, and their fixes are known a priori.
4. An evaluation of effectiveness will likely expose limitations/gaps in the current set of tools and techniques. This information can aid tool developers to improve their tools. This information can help researchers direct their efforts to identify the cause of these limitations and explore either ways to address the limitations or alternative approaches to prevent corresponding vulnerabilities, e.g., by extending platform capabilities.

1.2 Contributions

Motivated by above observations, we conducted an experiment to evaluate the effectiveness of vulnerability and malicious behavior detection tools for Android apps. We considered 64 tools and empirically evaluated 19 of them. We used benchmarks from Ghera repository [55] as they captured 42 known vulnerabilities and were known to be tool/technique agnostic, authentic, feature specific, minimal, version specific, comprehensive, and dual (i.e., contain both vulnerable and malicious apps).

To ensure the findings from the above evaluation were applicable in the context of real world apps, we evaluated if Ghera benchmarks were representative of real world apps, i.e., do the benchmarks capture vulnerabilities as they occur in real world apps?

In this paper, we describe these evaluations and report our observations from them. Besides reporting about the effectiveness of tools and the representativeness of Ghera benchmarks, we also report about patterns that were prevalent in the evaluated tools.

The remainder of the paper is organized as follows. Section 2 describes Ghera repository and our rationale for using it to evaluate tools. Section 3 describes the experiment to measure the representativeness of Ghera benchmarks along with our observations from the experiment. Section 4 describes the experiment to evaluate the effectiveness of vulnerability and malicious behavior detection tools for Android apps. Section 5 discusses prior evaluations of Android security analysis tools and how our evaluation relates to them. Section 6 provides information to access the automation scripts we used to perform the evaluation and the artifacts generated in the evaluation. Section 7 mentions possible extensions to this effort. Section 8 summarizes our observations from this evaluation.

2 Ghera Vulnerability Benchmarks

For this evaluation, we considered the Android app vulnerabilities cataloged in *Ghera*, a growing repository of benchmarks that captures known vulnerabilities in Android apps [55]. Most of the captured vulnerabilities have been reported and documented in prior work.

Ghera contains two kinds of benchmarks: *lean* and *fat*. *Lean benchmarks* are stripped down apps that exhibit vulnerabilities and exploits with almost no other interesting behaviors. *Fat benchmarks* are real world apps that exhibit specific known vulnerabilities. In the rest of this paper, we will focus on lean benchmarks and refer to them as benchmarks.

Each benchmark capturing a specific vulnerability X contains three apps (where applicable): a *benign (vulnerable)* app with vulnerability X,[1] a *malicious* app capable of exploiting vulnerability X in the benign app, and a *secure* app without vulnerability X and, hence, not exploitable by the malicious app. Malicious apps are absent in benchmarks when malicious behavior occurs in outside the Android environment, e.g., web app. Secure apps are absent in benchmarks when the fix is not programmatic, e.g., fix the build process. Each benchmark is accompanied by instructions to demonstrate the captured vulnerability and the corresponding exploit by building and executing the associated apps. Consequently, the presence and absence of vulnerabilities and exploits in these benchmarks is verifiable.

At the time of this evaluation, Ghera contained 42 benchmarks grouped into the following seven categories based on the nature of the APIs (including features of XML-based configuration) involved in the creation of captured vulnerabilities. (Category labels appear in square brackets.)

1. *Crypto* category contains 4 vulnerabilities involving cryptography API. [Crypto]
2. *ICC* category contains 16 vulnerabilities involving inter-component communication (ICC) API. [ICC]
3. *Networking* category contains 2 vulnerabilities involving networking (non-web) API. [Net]
4. *Permission* category contains 1 vulnerability involving permission API. [Perm]
5. *Storage* category contains 6 vulnerabilities involving data storage and SQL database APIs. [Store]
6. *System* category contains 4 vulnerabilities involving system API dealing with processes. [Sys]

[1] We use the terms benign and vulnerable interchangeably.

7. *Web* category contains 9 vulnerabilities involving web API. [Web]

Section A briefly catalogs these vulnerabilities and their canonical references.

2.1 Why Use Ghera?

For this tools evaluation to be useful to tool users, tool developers, and researchers, the evaluation should be based on vulnerabilities that are *valid* (i.e., will result in a weakness in an app), *general* (i.e., do not depend on uncommon constraints such as rooted device or admin access), *exploitable* (i.e., can be used to inflict harm), and *current* (i.e., occur in existing apps and can occur in new apps).

The vulnerabilities captured in Ghera benchmarks have been either previously reported in literature or documented in Android documentation; hence, they are *valid*. These vulnerabilities can be verified by executing the benign and malicious apps in Ghera benchmarks on vanilla Android devices and emulators; hence, they are *general* and *exploitable*. These vulnerabilities are *current* as they are based on Android API levels 19 thru 25, which enable more than 90% of Android devices in the world and are targeted by both existing and new apps.

Due to these characteristics and the salient characteristics of Ghera — *tool and technique agnostic, authentic, feature specific, contextual (lean), version specific, duality* and *comprehensive* — described in [55], Ghera is well-suited for this evaluation.

3 Representativeness of Ghera Benchmarks

Along with the above requirements, the manifestation of a vulnerability considered in the evaluation should be *representative* of the real world manifestations of the vulnerability.

A vulnerability can *manifest/occur* in different ways in apps due to various aspects such as producers and consumers of data, nature of data, APIs involved in handling and processing data, control/data flow paths connecting various code fragments involved in the vulnerability, and platform features involved in the vulnerability. As a simple example, consider a vulnerability that leads to information leak: sensitive data is written into an insecure location. This vulnerability can manifest in multiple ways. Specifically, at the least, each combination of different ways of writing data into a location (e.g., using different I/O APIs) and different insecure locations (e.g., insecure file, untrusted socket) can lead to *a unique manifestation of the vulnerability*.

In terms of representativeness, there is no evidence benchmarks in Ghera capture vulnerabilities as they occur in real world apps; hence, we needed to establish the representativeness of these benchmarks.

3.1 How to Measure Representativeness?

Since Ghera benchmarks capture specific manifestations of known vulnerabilities, we wanted to identify these manifestations in real world apps to establish the representativeness of the benchmarks. However, there was no definitive list of versions of apps that exhibit known vulnerabilities. So, we explored CVE [56], an open database of vulnerabilities discovered in real world Android apps, to identify vulnerable versions of apps. We found that most CVE vulnerability reports failed to provide sufficient information about the validity, exploit-ability, and manifestation of vulnerabilities in the reported apps. Next, we considered the option of manually examining apps mentioned in CVE reports for vulnerabilities. This option was not viable because CVE vulnerability reports do not include copies of reported apps. Also, while app version information from CVE could be used to download apps for manual examination, only the latest version of apps were available from most Android app stores and app vendors.

Finally, we decided to use usage information of Android APIs involved in manifestations of vulnerabilities as a proxy to establish the representativeness of Ghera benchmarks. The rationale for this decision is the likelihood of a vulnerability occurring in real world apps will be directly proportional to the number of real world apps using the Android APIs involved in the vulnerability. So, *as a weak yet general measure of representativeness, we identified Android APIs used in Ghera benchmarks and measured how often these APIs were used in real world apps.*

3.2 Experiment

3.2.1 Source of Real World Apps

We used AndroZoo as the source of real world Android apps. AndroZoo is a growing collection of Android apps gathered from several sources including the official Google Play store [28]. In May 2018, AndroZoo contained more than 5.8 million different APKs (app bundles).

Every APK (app bundle) contains an XML-based manifest file and a DEX file that contains the code and data (i.e., resources, assets) corresponding to the app. By design, each Android app is self contained. So, the DEX file contains all code that is necessary to execute the app but not provided by the underlying Android Framework or Runtime. Often, this includes code for Android support library.

AndroZoo maintains a list of all of the gathered APKs. This list documents various features of APKs such as SHA256 hash of an APK (required to download the APK from AndroZoo), size of an APK, and the date (*dex_date*) associated with the contained DEX file.[2] However, this list does not contain information about API levels (Android versions) that are targeted by the APKs; this information can be recovered from the APKs after downloading them.

3.2.2 App Sampling

Each version of Android is associated with an API level, e.g., Android versions 5.0 and 5.1 are associated with API levels 21 and 22, respectively. Every Android app is associated with a minimum API level (version) of Android required to use the app and a target API level (version) of Android that is ideal to use the app; this information is available in the app's manifest file.

At the time of this tools evaluation, Ghera benchmarks targeted API levels 19 thru 25 excluding 20.[3][4] So, we decided to select only apps that targeted API level 19 or higher and required minimum API level 14 or higher.[5] Since minimum and target API levels of apps were not available in the AndroZoo APK list, we decided to select apps based on their release dates. As API level 19 was released in November 2014, we decided to select only apps that were released after 2014. Since release dates of APKs were not available from AndroZoo APK list, we decided to use dex_date as a weak proxy for release date of apps.

Based on the above decisions, we analyzed the list of APKs available from AndroZoo to select the APKs to download. We found a total of 2.3 million APKs with dex_date between 2015 and 2018 (both inclusive). In these APKs, there were 790K, 1346K, 156K, and 17K APKs with dex_date from years 2015, 2016, 2017, and 2018, respectively. From these APKs, we drew an unequal probability sample without replacement and with probabilities 0.1, 0.2, 0.4, and 0.8 of selecting an APK from years 2015 thru 2018, respectively. We used *unequal probability sampling* to give preference to latest APKs as the selected apps would likely target recent API levels and to adjust for the non-uniformity of APK distribution across years. To create a sample with at least 100K real world Android apps that targeted the chosen API levels, we tried to download 339K APKs and ended up downloading 292K unique APKs. Finally, we used `apkanalyzer` tool from Android Studio to identify and discard downloaded apps (APKs) with target API level less than 19 or minimum API level less than 14. *This resulted in a sample of 111K real world APKs that targeted API levels 19 thru 25 (excluding 20) or higher.*

3.2.3 API-based App Profiling

Android apps access various capabilities of the Android platform via features of XML-based manifest files and Android programming APIs. We refer to the published Android programming APIs and the elements and attributes (features) of manifest files collectively as APIs.

For each app (APK), we collected its API profile based on the APIs that were used by or defined in the app and were deemed as *relevant* to this evaluation as follows.

[2] dex_date may not correspond to the release date of the app.

[3] API level 20 was excluded because it was API level 19 with wearable extensions.

[4] In the rest of this manuscript, "API levels 19 thru 25" means API levels 19 thru 25 excluding 20.

[5] We chose API level 14 as the cut-off for minimum API level as the number of apps targeting API level 19 peaked at minimum API level 14.

1. From the list of elements and attributes that can be present in a manifest file, informed by Ghera benchmarks, we conservatively identified the values of 7 attributes (e.g., *intent-filter/category@name*), the presence of 6 elements (e.g., *uses-permission*), and the presence of 26 attributes (e.g., *path-permission@writePermission*) as APIs relevant to this evaluation. For each app, we recorded which of these APIs were used in the app's manifest.

2. For an app, we considered all published (public and protected) methods along with all methods that were used but not defined in the app. Former methods accounted for callback APIs and latter methods accounted service offering APIs. We also considered all fields used in the app. From these considered APIs, we discarded obfuscated APIs, i.e., with single character name. To make apps comparable in the presence of definitions and uses of overridden Java methods (APIs), if a method was overridden, then we considered the fully qualified name (FQN) of the overridden method in place of the FQN of the overriding method (using Class Hierarchy analysis). Since we wanted to measure representativeness in terms of Android APIs, we discarded APIs whose FQN did not have any of these prefixes: *java, org, android,* and *com.android*. For each app, we recorded the remaining APIs.

3. Numerous APIs are commonly used in almost all Android APKs, e.g., *android.graphics.**. To avoid their influence on the result, we decided to ignore such APIs that are not relevant to app security. So, we considered the benign app of the template benchmark in Ghera repository; this app is a basic Android app with one activity containing couple of widgets and no functionality. Out of the 1502 APIs used in this app, we manually identified 1134 APIs as commonly used in Android apps and not relevant to app security. For each app, we removed these APIs from its list of recorded APIs and considered the remaining APIs as relevant APIs.

To collect API profiles of apps in Ghera, we performed the above steps starting with the APKs available in Ghera because extraneous APIs had been eliminated from these APKs by using `proguard` tool.

While collecting API-based profile of apps in AndroZoo sample, we discarded 2% of the APKs due to errors in APKs (e.g., missing required attributes) and tooling issues. *Finally, we ended up with a sample of 109K real world APKs (apps) from AndroZoo.*

3.2.4 Measuring Representativeness

We identified the set of relevant APIs associated with benign apps in Ghera using the steps described in the previous section. Of the resulting 601 unique relevant APIs, we identified 117 as security related APIs. For both these sets of APIs, we measured representativeness in two ways.

1. *Using API Use Percentage.* For each API, we calculated the percentage of sample apps that used the API.

   To observe how representativeness changes across API levels, we created API level specific app samples. The app sample specific to API level k contained every sample app whose minimum API level was less than or equal to k and target API level was greater than or equal to k. In each API level specific sample, for each API, we calculated the percentage of apps that used the API.

   The rationale for this measurement is, *if Ghera benchmarks are representative of real world apps in terms of using an API, then a large number of real world apps would use the API.*

2. *Comparing Sampling Proportions.* For each API, we calculated the sampling proportion of sample apps that used the API. To calculate the sampling proportion, we considered 80% of the sample apps, partitioned it into sub-samples containing 40 units each, and calculated the mean of the proportions in each sub-sample. We also calculated the sampling proportion of benign apps in Ghera that used the API. We then compared the sampling proportions of an API with confidence level = 0.95, p-value $\leq$ 0.01, and the null hypothesis being the proportion of benign apps in Ghera using the API is less than or equal to the proportion of real world apps using the API.

   We performed this proportion test both across API levels and at specific API levels.

   The rationale for this measurement is, *if Ghera benchmarks are representative of real world apps in terms of using an API, then the proportion of Ghera benchmarks using the API should be less than or equal to the proportion of real world apps using the API.*

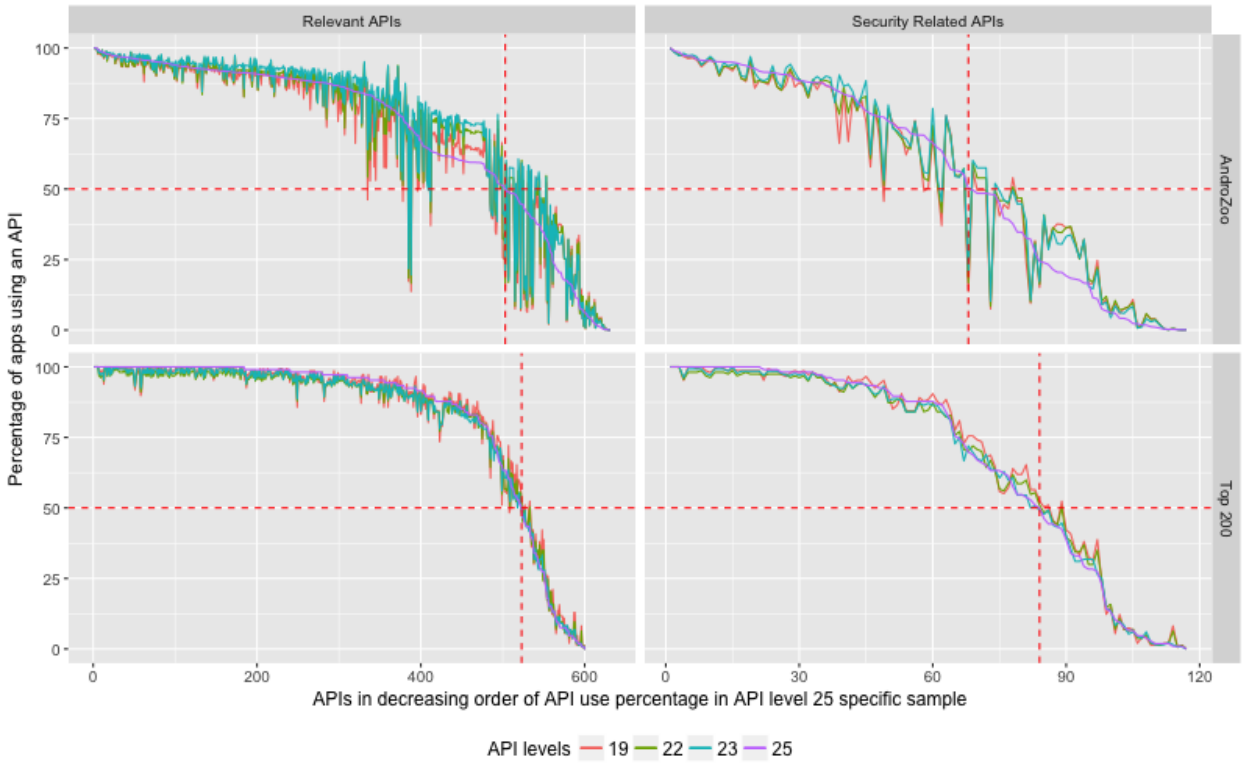

Figure 1: Percentage of apps that use APIs used in benign apps in Ghera.

With Top 200 Apps. We gathered the top 200 apps from Google Pay store on April 18, 2018, and repeated the above measurements both across API levels and at specific API levels. Only 167 of the top 200 apps made it thru the app sampling and API-based app profiling process due to API level restrictions, errors in APKs, and tooling issues. Hence, we considered 40 sub-samples (containing 40 apps each) when measuring representativeness by comparing sampling proportions.

3.3 Observations

3.3.1 Based on API Use Percentage

The color graphs in Figure 1 show the percentage of sample real world Android apps using the APIs used in benign apps in Ghera. Y axis denotes percentage of apps using an API. X axis denotes APIs in decreasing order of percentage of their usage in API level 25 specific sample. The graphs on the left are based on relevant APIs used in benign apps in Ghera while the graphs on the right are based only on security related APIs. The graphs on the top are based on the sample of apps from AndroZoo while the graphs on the bottom are based on the sample of top 200 apps from Google Play store. To avoid clutter, we have not plotted data for API levels 21 and 24 as they closely related to API levels 22 and 25, respectively.

Since API level 25 is the latest API level considered in this evaluation, consider the data for API level 25 specific samples. In AndroZoo sample, 83% (503 out of 601) of relevant APIs used in benign apps in Ghera were each used by more than 50% (54K) of real world apps. For top 200 apps sample, this number increases to 87% (523 out of 601). When considering only security related APIs, 58% (68 out of 117) of APIs used in benign apps in Ghera were each used by more than 50% of real world apps in AndroZoo sample. For top 200 apps sample, this number increases to 71% (84 out of 117).

Barring few APIs in case of AndroZoo sample, the above observations hold true for all API levels considered from 19 thru 25 in both AndroZoo sample and top 200 apps sample.

Upon considering this measurement for malicious apps and secure apps in Ghera, we observed the per-

centage of APIs that were used in Ghera and were used by more than 50% of real world apps to be similar or higher than the above percentages in both AndroZoo sample and top 200 apps sample.

Above observations suggest *a large number of real world apps use a large number of APIs used in Ghera benchmarks. Consequently, we can conclude Ghera benchmarks are representative of real world apps.*

Side Note Observe that many security related APIs are used by a large percentage of top 200 apps. This is no surprise as these apps are likely to be widely used and are built to keep their user's information secure. In comparison, the number of security related APIs being used in real world apps in general (AndroZoo sample) is pretty high — 68 security related APIs are each used by more than 54K apps from AndroZoo sample in all considered API levels. This suggests *a large number of real world apps use security related APIs, knowingly or unknowingly and correctly or incorrectly.* Hence, there is a huge opportunity to help identify and fix incorrect use of security related APIs.

3.3.2 Based on Sampling Proportions

For APIs used in benign apps in Ghera, columns 4, 5, 8, and 9 in Table 1 report the number of APIs for which the null hypothesis could not be rejected. This data suggests, for at least 76% of relevant APIs, the proportion of Ghera apps that used an API could be less than or equal to the proportion of real world apps in AndroZoo sample that use the same API. This is true across all API levels and at specific API levels. This is also true for at least 96% of security related APIs. In case of top 200 apps sample, this is true for at least 92% of relevant APIs and 99% of security related APIs.

Upon considering this measurement for malicious apps and secure apps in Ghera, we made similar observations in both AndroZoo sample and top 200 apps sample.

Considering Ghera benchmarks as a custom sample in which the proportion of benchmarks that used a specific set of APIs (relevant or security related) was expected to be high, the above observations suggest such proportions are higher for these APIs in real world apps. Consequently, we can conclude Ghera benchmarks are representative of real world apps.

3.4 Threats to Validity

This evaluation of representativeness is based on a weak measure of manifestation of vulnerabilities — use of APIs used in vulnerabilities. Hence, this evaluation could have ignored the influence of aspects such as API usage context, nature of involved data, and data/control flow path connecting various API uses. Such influences can be verified and addressed by measuring representativeness while considering these aspects.

While a large sample of real world apps were considered, the distribution of apps across targeted API levels was skewed — there were fewer apps targeting recent API levels. Hence, recent API level specific samples may not have exhibited the variations observed in larger API specific samples, e.g., API level 19 (see Figure 1). This possibility can be verified by repeating this experiment with API level specific app samples of comparable sizes.

The version of Ghera benchmarks considered in this evaluation was developed when API level 25 was the latest Android API level. So, it is possible that tooling and library support available for API level 25 could have influenced the structure of Ghera benchmarks and, consequently, the observations in Section 3.2. This possibility can be verified by repeating this experiment in the future with tooling and library support for newer API levels.

We have taken extreme care to avoid errors while collecting and analyzing data. Even so, there could be errors in our evaluation in the form of commission errors (e.g., misinterpretation of data), omission errors (e.g., missed data points), and automation errors. This threat to validity can be addressed by examining both our raw and processed data sets (see Section 6), analyzing the automation scripts, and repeating the experiment.

	AndroZoo Apps				Top 200 Apps			
			# Representative APIs (%)				# Representative APIs (%)	
API Level	Sample Size	# of Sub-Samples	Relevant	Security Related	Sample Size	# of Sub-Samples	Relevant	Security Related
19-25	109028	2180	459 (76)	113 (96)	168	40	555 (92)	116 (99)
19	108925	2178	459 (76)	113 (96)	147	40	562 (93)	116 (99)
21	96364	1927	464 (77)	113 (96)	161	40	557 (92)	116 (99)
22	80664	1613	471 (78)	113 (96)	157	40	562 (93)	116 (99)
23	56212	1124	479 (79)	114 (97)	132	40	553 (92)	116 (99)
24	7616	152	483 (80)	114 (97)	112	40	566 (94)	116 (99)
25	2901	58	473 (78)	114 (97)	106	40	567 (94)	116 (99)

Table 1: Representativeness based on sample proportions test. Of the 601 selected (all) APIs, 117 APIs were security related.

4 Tools Evaluation

4.1 Android App Security Analysis Solutions

While creating Ghera repository in 2017, we became aware of solutions for Android app security. From June 2017, we started collecting information about such solutions. Our primary sources of information were research papers [61, 59] and repositories and blog posts collating information about Android security solutions [1, 22].

From these sources, we considered 64 solutions that were related to Android security.[6] We classified them along the following dimensions.

1. *Tools vs Frameworks: Tools* detect a fixed set of security issues. While they can be applied immediately, they are limited to detecting a fixed set of issues. On the other hand, *frameworks* facilitate creation of tools that can detect specific security issues. While they are not immediately applicable to detect vulnerabilities and involve effort to create tools, they enable detection of relatively open set of issues.

2. *Free vs Commercial:* Solutions are available either freely or for a fee.

3. *Maintained vs Unmaintained:* Solutions are either actively maintained or unmaintained. Typically, unmaintained solutions do not support currently supported versions of Android. This is also true of some maintained solutions

4. *Vulnerability Detection vs Malicious Behavior Detection:* Solutions either detect vulnerabilities in an app or flag signs of malicious behavior in an app. The former is typically used by app developers while the latter is used by app stores and end users.

5. *Static Analysis vs Dynamic Analysis:* Solutions that rely on *static analysis* analyze either source code or Dex bytecode of an app and provide verdicts about possible security issues in the app. Since the underlying analysis abstracts the execution environment and semantics along with the interactions with users and other apps, such solutions can detect issues that occur in a variety of settings. However, such solutions can consider invalid settings (due to too permissive abstractions) and, hence, have high false positive rate.

   In contrast, solutions that rely on *dynamic analysis* execute apps and check for security issues at runtime. Consequently, they have very low false positive rate. However, they are often prone to high false negative rate because they may fail to explore specific settings required to trigger security issues in an app.

6. *Local vs Remote:* Solutions are available as executables or as sources from which executables can be built. These solutions are installed and executed locally by the developer. They are also executed within the IDE or as part of the build process.

   Solutions are also available remotely as web services (or via web portals). Developers use these services by submitting the APKs of their apps for analysis and later accessing the analysis reports. Unlike local solutions, developers are not privy to what happens to their apps when using remote solutions.

4.2 Tools Selection

4.2.1 Shallow Selection

To select tools for this evaluation, we first screened the considered 64 solutions by reading their documentation and any available resources. We rejected 5 solutions because they were not well documented, e.g., no documentation, lack of instructions to build and use tools. This was necessary to eliminate human bias resulting from the effort involved in discovering how to build and use a solution, e.g., DroidLegacy [39], BlueSeal [62]. We rejected AppGuard [24] because its documentation was not in English. We rejected 6 solutions such as Aquifer [57], Aurasium [67], and FlaskDroid [34] as they were not intended to detect vulnerabilities, e.g., enforce security policy.

[6] The complete list of tools that were considered for this evaluation is available at https://bitbucket.org/secure-it-i/may2018.

Tool [Reference]	Commit Id / Version	Updated (Published)	Static / Dynamic	Local / Remote	API (Vulnerability) Categories							Set Up Time (sec)
					Crypto	ICC	Net	Perm	Store	Sys	Web	
Amandroid [66]	3.1.2	2017 (2014)	S	L	✓	✓						3600
AndroBugs [2]	7fd3a2c	2015 (2015)	S	L	✓	✓		✓	✓	✓	✓	600
AppCritique [7]	?	? (?)	?	R	✓	✓		✓	✓	✓	✓	
COVERT [32]	2.3	2015 (2015)	S	L	✓	✓	✓	✓	✓	✓	✓	2700
DialDroid [33]	25daa37	2018 (2016)	S	L		✓						3600
DevKnox [40]	2.4.0	2017 (2016)	S	L	✓	✓	✓	✓	✓	✓	✓	600
FixDroid [58]	1.2.1	2017 (2017)	S	L	✓			✓	✓	✓	✓	600
FlowDroid [30]	2.5.1	2018 (2013)	S	L		✓	✓		✓			9000
HornDroid [35]	aa92e46	2018 (2017)	S	L	✓	✓	✓		✓		✓	600
JAADS [19]	0.1	2017 (2017)	S	L		✓		✓	✓	✓		900
MalloDroid [44]	78f4e52	2013 (2012)	S	L							✓	600
Marvin-SA[a] [50]	6498add	2016 (2016)	S	L	✓	✓	✓	✓	✓	✓	✓	600
MobSF [27]	b0efdc5	2018 (2015)	SD	L	✓	✓		✓	✓	✓	✓	1200
QARK [53]	1dd2fea	2017 (2015)	S	L	✓	✓	✓	✓	✓	✓	✓	600
AndroTotal [54]	?	? (2013)	SD	R	?	?	?	?	?	?	?	
HickWall [68]	?	? (2014)	SD	R	?	?	?	?	?	?	?	
Maldrolyzer [21]	0919d46	2015 (2015)	S	L	?	?	?	?	?	?	?	600
NVISO ApkScan [23]	?	? (?)	SD	R	?	?	?	?	?	?	?	
VirusTotal [26]	?	? (?)	S	R	?	?	?	?	?	?	?	

[a]We refer to Marvin Static Analyzer as Marvin-SA.

Table 2: Information about evaluated tools. Top part describes vulnerability detection tools. Bottom part describes malicious behavior detection tools. "?" denotes unknown information. "✓" denotes the tool is applicable to the vulnerability category in Ghera. Empty cell denotes not applicable cases.

Of the remaining 52 solutions, we selected solutions based on the first three dimensions mentioned in Section 4.1.

In terms of tools vs frameworks, we were interested in solutions that could readily detect vulnerabilities or malicious behaviors with no or minimal adaptation, e.g., build an extension to detect a specific vulnerability. The rationale was to eliminate human bias and errors involved in identifying, creating, and using the appropriate adaptations. Also, we wanted to mimic a simple developer workflow — procure/build the tool, follow its documentation, and apply it to programs. Consequently, we rejected 16 tools that only enabled security analysis, e.g., Drozer [17], ConDroid [29]. When a framework provided pre-packaged extensions to detect vulnerabilities or malicious behavior, we selected such frameworks and considered each such extension as a distinct tool, e.g., we selected Amandroid framework as it comes with seven pre-packaged vulnerability detection extensions (i.e., data leakage, intent injection, comm leakage, password tracking, OAuth tracking, SSL misuse, and crypto misuse) that can be used as tools [66].

In terms of free vs commercial, we rejected AppRay as it was a commercial solution [8]. While AppCritique was a commercial solution, a feature limited version of it was available for free. We considered the free version and did not reject AppCritique.

In terms of maintained vs unmaintained, we focused on selecting only maintained tools. So, we rejected AndroWarn and ScanDroid tools as they were not updated after 2013 [38, 45] . In a similar vein, since we were committed to currently supported Android API levels, we rejected TaintDroid as it supported only API levels 18 or below [42].

Next, we focused on tools that could be applied as is without extensive configuration (or inputs). The rationale was to eliminate human bias and errors involved in identifying and using appropriate configurations. So, we rejected tools that required additional inputs to detect vulnerabilities. Specifically, we rejected Sparta as it required analyzed apps to be annotated with security types [43].

Finally, we focused on the applicability of tools to Ghera benchmarks. We considered only tools that claimed to detect vulnerabilities stemming from APIs covered by at least one category in Ghera benchmarks. For such tools, based on our knowledge of Ghera benchmarks and shallow exploration of the tools, we assessed if the tools were indeed applicable to the benchmarks in the covered categories. This included checking if the APIs used in Ghera benchmarks were mentioned in any lists of APIs bundled with tools, e.g., list of information source and sink APIs bundled with HornDroid and FlowDroid. In this regard, we rejected 3 tools. We rejected PScout [31] because PScout focused on vulnerabilities related to over/under use of permissions and the only permission related benchmark in Ghera was not related to over/under use of permissions. We rejected StaDyna because StaDyna detected malicious behavior stemming from dynamic code loading and Ghera did not have benchmarks that used dynamic code loading [69]. We also rejected Amandroid's OAuth tracking extension ($Amandroid_5$) and LetterBomb [46] as they were not applicable to any Ghera benchmark.[7]

4.2.2 Deep Selection

Of the remaining 28 tools, for tools that could be executed locally, we downloaded the latest official release of the tool, e.g., Amandroid.

If such a release was not available, then we downloaded the most recent version of the tool (executable or source code) from the master branch of its repository, e.g., AndroBugs. We then followed the instructions in the tool's documentation to build and set up the tool. If we encountered issues during this phase, then we tried to fix the issues. These issues stemmed for reasons such as tools depending on older versions of other tools (e.g., HornDroid failed against real world apps as it was using an older version of `apktool`, a decompiler for Android apps), incorrect documentation (e.g., documented path to the DialDroid executable was incorrect), and incomplete documentation (e.g., IccTA's documentation did not mention the versions of required dependencies [52]). We stopped trying to fix an issue and rejected a tool if we could not figure out a fix by interpreting the error messages and by exploring existing publicly available bug reports. This resulted in rejecting DidFail [51].

Of the remaining tools, we tested 18 local tools using apps from I1, I2, W8, and W9 Ghera benchmarks and Offer Up, Instagram, Microsoft Outlook, and My Fitness Pal's Calorie Counter apps from Google Play

[7] While Ghera did not have benchmarks that were applicable to some of the rejected tools at the time of this evaluation, it currently has such benchmarks that can be considered to extend this evaluation in the future.

store. We executed each tool with each of the above apps as input on a 16 core Linux machine with 64GB RAM and with 15 minute time out period. If a tool failed to execute successfully on all of these apps, then we rejected the tool. Specifically, we rejected IccTA and SMV Hunter because they failed to process the test apps by throwing exceptions [52, 63]. We rejected CuckooDroid and DroidSafe because they ran out of time or memory while processing the test apps [47, 9].

For 9 tools that were available only remotely, we tested them by submitting the above test apps for analysis. If a tool's web service was unavailable, failed to process all of the test apps, or did not provide feedback within 30–60 minutes, then we rejected it. Consequently, we rejected 4 remote tools, e.g., TraceDroid [25].

4.2.3 Selected Tools

Table 2 reports information about the set of 19 tools that were selected for this evaluation along with their canonical references. For each of these tools, if available, the table reports the version (or the commit id) selected for evaluation, date of its initial publication and latest update, whether it uses static analysis (S) or dynamic analysis (D) or both (SD), whether it runs locally (L) or remotely (R), and the Ghera categories that it is applicable to. For each of the selected tools, the last column in the table reports the time spent to set up tools on a Linux machine.

Unlike vulnerability detection tools, malicious behavior detection tools do not publicly disclose the malicious behaviors they can detect. Hence, we do not know the categories of Ghera benchmarks to which these tools apply.

4.3 Experiment

Every selected vulnerability detection tool (including pre-packaged extensions considered as tools) was applied in its default configuration to the *benign* app and the *secure* app (separately) of every applicable Ghera benchmark (given in column 9 in Table 3).

Every selected malicious behavior detection tool was applied in its default configuration to the *malicious* app of every applicable Ghera benchmark. Unlike in case of vulnerability detection tools, malicious behavior detection tools were evaluated on only 33 Ghera benchmarks because 9 benchmarks — 1 in Networking category and 8 in Web category — did not have malicious apps as they relied on non-Android apps to mount man-in-the-middle exploits. Further, each tool was evaluated on all 33 benchmarks as there was no information about the kind of exploits detected by the tool.

The tools were executed on a 16 core Linux machine with 64GB RAM and with 15 minutes time out. For each execution, we recorded the execution time and any output reports, error traces, and stack traces. We then examined the output to determine the verdict and its veracity pertaining to a vulnerability v.

Variations

FixDroid was not evaluated on secure apps in Ghera because Android Studio version 3.0.1 was required to build the secure apps in Ghera and FixDroid was available as a plugin only to Android Studio version 2.3.3. Further, since benchmarks C4, I13, I15, and S4 were added after Ghera was migrated to Android Studio version 3.0.1, FixDroid was not evaluated on these benchmarks; hence, FixDroid was evaluated on only 38 Ghera benchmarks.

COVERT and DialDroid detect vulnerabilities stemming from inter-app communications, e.g., collusion, compositional vulnerabilities. So, to evaluate these tools, we applied each tool in its default configuration to every 33 Ghera benchmarks with malicious apps by providing the benign app and the malicious app together as input.

JAADS operates in two modes: fast mode in which only intra-procedural analysis is performed and full mode in which both intra- and inter-procedural analyses are performed. Since the modes can be selected easily, we evaluated JAADS in both modes.

QARK analyzes both source code and APK of an app. It decompiles APK into source form. Since the structure of reverse engineered source code may differ from original source code and we did not know if this could affect the accuracy of QARK's verdicts, we evaluated QARK with both APKs and source code.

4.4 Observations & Questions

4.4.1 Tools Selection

Of the considered 64 solutions, 17 tools (including Amandroid) were intended to enable security analysis of Android apps. This raises two related questions worth pursuing: *how expressive, effective, and easy-to-use are these tools?* and *are Android app developers and security analysts willing to invest effort in using tools that enable security analysis?*

We rejected only 32% of tools (9 out of 28) considered in deep selection. Even considering the number of instances when evaluated tools failed to process certain benchmarks, such low rejection rate is rather impressive. This suggests *more tool developers are putting in effort to release robust security analysis tools.* This number can be further improved by distributing executables (where applicable), providing complete and accurate build instructions (e.g., versions of required dependencies) for local tools, and providing estimated turn around times for remote tools.

If the sample of tools considered in this evaluation is representative of the population of Android app security analysis tools, then *almost every Android app security analysis tool relies on static analysis, i.e., 18 out of 19.*

While every vulnerability detection tool publicly discloses the category of vulnerabilities it tries to detect, none of the malicious behavior detection tools publicly disclose the malicious behaviors they try to detect. Also, while almost all vulnerability detection tools are available as locally executable tools (i.e., 13 out of 14), almost none of the malicious behavior detection tools are available as remote services, i.e., 4 out of 5. So, in terms of information about detection capabilities, the vulnerability detection tools are *open* and malicious behavior detection tools are *closed*. This difference may stem from the purpose of these tools: *vulnerability detection tools are intended to build secure apps while malicious behavior analysis tools are intended to thwart malicious apps.*

Ignoring tools with unknown update dates ("?" in column 3 of Table 2) and considering the evaluation was conducted between June 2017 and May 2018, 9 out of 13 tools are less than 1.5 years old (2017 or later) and 12 out of 13 are less than or about 3 years old (2015 or later). Hence, *the considered tools are current and the resulting observations are highly likely to be representative of the current state of the freely available Android app security analysis tools.*

4.4.2 Vulnerability Detection Tools

Table 3 summarizes the effectiveness of tools in detecting different categories of vulnerabilities.

Most of the tools (11 out of 14) were applicable to every Ghera benchmark. With the exception of MalloDroid, the rest of tools were applicable to 24 or more Ghera benchmarks. This observation is also true of Amandroid if the results of its pre-packaged extensions are considered together.

Every Ghera benchmark is associated with exactly one unique vulnerability v, and its benign app exhibits v while its secure apps does not exhibit v. So, for a tool, for each applicable benchmark, we classified the tool's verdict for the benign app as either true positive (i.e., v was detected in the benign app) or false negative (i.e., v was not detected in the benign app). We classified the tool's verdict for the secure app as either true negative (i.e., v was not detected in a secure app) or false positive (i.e., v was detected in a secure app). Columns 10, 11, and 12 in Table 3 reports true positives, false negatives, and true negatives, respectively. False positives are not reported in the table as none of the tools except DevKnox (observe the *D* under *System* benchmarks in Table 3) and *data leakage* extension of Amandroid (observe *(1)* for $Amandroid_1$ under *Storage* benchmarks in Table 3) provided false positives verdicts. Reported verdicts do not include cases in which a tool failed to process apps, e.g., the number of true negatives for DialDroid is 0 because it failed to process every secure app.

Based on the classification of the verdicts, *4 out of 14 tools detected none of the vulnerabilities captured in Ghera* (considering all extensions of Amandroid as one tool). Even in case of tools that detected some of the vulnerabilities captured in Ghera, *none of the tools detected more than 15 out of the 42 vulnerabilities.* This is also evident by the number of N's in Table 3. This suggests *current tools (in isolation) are very limited in their ability to detect known vulnerabilities captured in Ghera.*

Tool	Crypto (4)	ICC (16)	Net (2)	Perm (1)	Store (6)	Sys (4)	Web (9)	# Applicable Benchmarks	Benign		Secure	Other
									TP	FN	TN	
Amandroid$_1$		7/**0**/**9**/3	1/**0**/**1**/0		4/**0**/**1**/0 {1}		6/**0**/**3**/0	15	1	14	14	3
Amandroid$_2$		X		N	1/**0**/**2**/0 [3]	X	5/**0**/**0**/0 [4]	30	0	3	3	0
Amandroid$_3$		8/**0**/**8**/2	1/**0**/**1**/0		4/**0**/**2**/0		6/**0**/**3**/0	14	0	14	14	2
Amandroid$_4$		13/**0**/**3**/2						3	0	3	3	2
Amandroid$_6$							6/**0**/**3**/0	3	0	3	3	0
Amandroid$_7$	0/**2**/**2**/0							4	2	2	4	0
AndroBugs	N	0/**2**/**14**/3	N	0/**1**/**0**/0	N	0/**4**/**0**/0	0/**4**/**5**/1	42	11	31	42	4
AppCritique	0/**2**/**2**/0	N	N	N	0/**3**/**3**/0	N	0/**2**/**7**/0	42	7	35	42	0
COVERT	N	N	N	N	N	N		33	0	33	33	0
DialDroid	N	N	1/**0**/**1**/0	N	N	N	8/**0**/**1**/0	33	0	33	0	0
DevKnox	0/**1**/**3**/0	N	N	N	N	D	N	42	5	37	38	0
FlowDroid		N	N		N			24	0	24	24	0
HornDroid	N	0/**1**/**15**/7	N		0/**0**/**6**/1		0/**0**/**9**/1	37	1	36	37	9
JAADS$_1$	N	0/**2**/**14**/0	N	N	N	N	0/**4**/**5**/1	42	6	36	42	1
JAADS$_2$	N	0/**2**/**14**/0	N	N	N	N	0/**4**/**5**/1	42	6	36	42	1
MalloDroid			X				5/**0**/**1**/0 [3]	4	0	1	1	0
Marvin-SA	0/**1**/**3**/0	0/**5**/**11**/3	N	0/**1**/**0**/0	0/**0**/**6**/2	0/**4**/**0**/0	0/**4**/**5**/0	42	15	27	42	5
MobSF	0/**1**/**3**/0	0/**5**/**11**/0	N	0/**1**/**0**/1	0/**1**/**5**/0	0/**4**/**0**/0	0/**3**/**6**/0	42	15	27	42	1
QARK$_1$	N	0/**3**/**13**/0	N	0/**1**/**0**/0	N	0/**4**/**0**/0	0/**2**/**7**/0	42	10	32	42	0
QARK$_2$	N	0/**3**/**13**/0	N	0/**1**/**0**/0	N	0/**4**/**0**/0	0/**2**/**7**/0	42	10	32	42	0
# Undetected	1	5	2	0	2	0	2					

Table 3: Results from evaluating vulnerability detection tools. The number of benchmarks in each category is mentioned in parentheses. In each category, empty cell denotes the tool is inapplicable to any of the benchmarks, N denotes the tool flagged both benign and secure apps as not vulnerable in every benchmark, X denotes the tool failed to process any of the benchmarks, and D denotes the tool flagged both benign and secure apps as vulnerable in every benchmark. H/I/J/K denotes the tool was inapplicable to H benchmarks, flagged benign app as vulnerable and secure app as not vulnerable in I benchmarks, flagged both benign and secure app as not vulnerable in J benchmarks, and reported non-existent vulnerabilities in benign or secure apps in K benchmarks. The number of benchmarks that a tool failed to process is mentioned in square brackets. The number of benchmarks in which both benign and secure apps were flagged as vulnerable is mentioned in curly braces. "-" denotes not applicable cases.

For 11 out of 14 tools, the number of false negatives was within 30% of the number of true negatives.[8] This suggests two possibilities: *most tools prefer to report only valid vulnerabilities* or *most tools can only detect specific manifestations of vulnerabilities.* Both these possibilities have limited effectiveness in assisting developers build secure apps because validity of reported vulnerabilities takes precedence over building secure apps.

Tools make claims about specific vulnerabilities or class of vulnerabilities that they can detect. So, we examined such claims. While both COVERT and DialDroid claim to detect vulnerabilities related to communicating apps, neither detected such vulnerabilities in any of the 33 Ghera benchmarks that are contained a benign app and a malicious app. While MalloDroid focuses solely on SSL/TLS related vulnerabilities, it did not detect any of the SSL vulnerabilities captured in Ghera benchmarks. We observed similar failures with FixDroid. This suggests *there exists a gap between the claimed capabilities and the observed capabilities of tools that could lead to vulnerabilities in apps.*

Different tools use different kinds of analysis under the hood to perform security analysis. Tools such as Amandroid, FlowDroid, and HornDroid rely on deep analysis (e.g., data flow analysis) while tools such as QARK, Marvin-SA, and AndroBugs rely on shallow analysis (e.g., searching for code smells/patterns). In the evaluation, *tools that rely on deep analysis reported fewer true positives and more false negatives than tools that rely on shallow analysis.* A possible reason for this could be deep analysis tools often depend on extra information about the analyzed app (e.g., a custom list of sources and sinks to be used in data flow analysis) and we did not provide such extra information in our evaluation. However, JAADS was equally effective in both fast (intra-procedural analysis) and full (intra- and inter-procedural analyses) mode. Also, FixDroid was more effective than other deep analysis tools while operating within an IDE. This leads us to two questions: *are Android app security analysis tools that rely on deep analysis effective in detecting vulnerabilities?* and *are the deep analysis techniques used in these tools well suited to detect vulnerabilities in Android apps?* This question is pertinent because Ghera benchmarks capture known vulnerabilities and the benchmarks are small/lean in complexity, features, and size, i.e., often less than 1000 lines of developer created Java, XML, and build system code.

Switching the focus to vulnerabilities, each of the 5 vulnerabilities captured by *Permission* and *System* benchmarks were detected by some tool. However, none of the 2 vulnerabilities captured by *Networking* benchmarks category were detected by any tool. Considering all vulnerabilities, 12 out of 42 known vulnerabilities captured in Ghera were not detected by any tool. In other words, *using all tools together is not sufficient to detect the known vulnerabilities captured in Ghera.* However, 30 of the 42 vulnerabilities were detected using 15 tools with no or minimal configuration. This raises the questions: *with reasonable configuration effort, can the evaluated tools be configured to detect the undetected vulnerabilities?* and *would the situation improve if rejected vulnerability detection tools could be used?*

In the evaluation, 8 out of 14 tools reported vulnerabilities that were not the focus of Ghera benchmarks. (See last column in Table 3.) Upon manual examination of the benchmarks, we found none of these reported vulnerabilities were present in the benchmarks.

For tools that completed the analysis of apps (either normally or exceptionally), the median run time was 5 seconds with the lowest and highest run times being 2 and 63 seconds, respectively. So, *in terms of performance, all of the tools that completed analysis exhibited good run times.*

While the two modes of QARK provided identical verdicts, they exhibited the largest difference in run times: 2 seconds source code analysis mode and 63 seconds in APK analysis mode, which can be attributed to the conversion of bytecodes into source code. In this context, *all of the evaluated tools supported APK analysis.* A possible explanation for this is analyzing APKs helps tools cater to wider audience: APK developers and APK users (i.e., app stores and end users).

4.4.3 Malicious Behavior Detection Tools

Table 4 reports the results of evaluating malicious behavior detection tools. Unlike vulnerability detection tools, the results from malicious behavior detection tools are homogeneous and concerning.

Recall that 33 benchmarks in Ghera are composed of two apps: a malicious apps that exploits a benign app. So, while malicious apps in Ghera are indeed malicious, 3 out of 5 tools did not detect malicious

[8] We considered all variations of a tool as one tool, e.g., JAADS. We counted DialDroid as it failed to process every secure app in Ghera. We did not count FixDroid as it was not evaluated on secure apps in Ghera

Tool	Crypto (4)	ICC (16)	Net (1)	Perm (1)	Sys (4)	Store (6)	Web (1)	# PUP	# FN
AndroTotal	0/0/**4**	0/0/**16**	0/0/**1**	0/0/**1**	0/0/**4**	0/0/**6**	0/0/**1**	0	33
HickWall	0/0/**4**	0/0/**16**	0/0/**1**	0/0/**1**	0/0/**4**	0/0/**6**	0/0/**1**	0	33
Maldrolyzer	0/0/**4**	0/0/**16**	0/0/**1**	0/0/**1**	0/0/**4**	0/0/**6**	0/0/**1**	0	33
NVISO ApkScan	0/0/**4**	0/9/**7**	0/1/**0**	0/1/**0**	0/0/**4**	0/4/**2**	0/1/**0**	16	17
VirusTotal	0/4/**0**	0/16/**0**	0/1/**0**	0/1/**0**	0/4/**0**	0/6/**0**	0/1/**0**	33	0

Table 4: Results from evaluating malicious behavior detection tools. The number of benchmarks in each category is mentioned in parentheses. X/Y/Z denotes the tool failed to process X benchmarks, deemed Y benchmarks as PUPs, and failed to flag Z benchmarks as malicious.

behavior in any of the malicious apps. VirusTotal flagged all of the malicious apps as *potentially unwanted programs (PUPs)*, which is not the same as being malicious; it is more akin to flagging a malicious app as non-malicious. NVISO ApkScan flagged one half of the apps as PUPs and the other half as non-malicious. In short, *all of the malicious behavior detection tools failed to detect any of the malicious behaviors in Ghera benchmarks.*

Since AndroTotal, NVISO ApkScan, and VirusTotal rely on antivirus scanners to detect malicious behavior, the results suggest *the malicious behaviors captured in Ghera benchmarks will likely go undetected by antivirus scanners.* Also, since HickWall, Maldrolyzer, and NVISO ApkScan rely on static and/or dynamic analysis to detect malicious behavior, the results suggest *static and dynamic analyses used in these tools are ineffective in detecting malicious behaviors captured in Ghera benchmarks.*

4.5 Threats to Validity

While we tried to execute the tools using all possible options but with minimum or no extra configuration, we may not have considered options or combinations of options that could result in more true positives and true negatives. The same is true of extra configuration required by certain tools, e.g., providing custom list of sources and sink to FlowDroid.

Our personal preferences for IDEs (e.g., Android Studio over Eclipse) and flavors of analysis (e.g., static analysis over dynamic analysis) could have biased how we diagnosed issues encountered while building and setting up tools. This could have affected the selection (rejection) of tools and the reported set up times.

We have taken utmost care in using the tools, collecting their outputs, and analyzing their verdicts. However, our bias along with commission errors (e.g., incorrect tool use) and omission errors (e.g., missed data) could have affected the evaluation.

All of the above threats can be overcome by having this evaluation repeated by different experimenters with different biases and preferences than us and comparing their observations with our observations as documented in this manuscript and the artifacts repository (see Section 6).

Our observations are based on the evaluation of 19 security analysis tools. While this is a reasonably large set of tools, it may not be representative of the population of security analysis tools, e.g., it does not include commercial tools. So, the above observations should be considered only in the context of tools similar to the evaluated tools. More exploration should be performed before generalizing the observations to other tools.

4.6 Interaction with Tool Developers

By end of June 2017, we had completed a preliminary round of evaluation of QARK and AndroBugs. Since we were surprised by the low detection rate of both tools, we contacted both teams with our results and the corresponding responses were very different.

QARK team acknowledged our study and reverted back to us with a new version of their tool after two months. We evaluated this new version of QARK. While the previous version of the tool flagged 3 benchmarks as vulnerable, the new version flagged 10 benchmarks as vulnerable. The evaluation reported in this manuscript uses this new version of QARK.

AndroBugs team pointed out that the original version of their tool was not available on GitHub; hence, we were not using the original version of AndroBugs in our evaluation. When we requested the original version of the tool, the team did not respond.

After these interactions, we decided not to communicate with tool developers until the evaluation was complete. The rationale for this decision was to make the evaluation fair by evaluating the tools as publicly available without being influenced by this evaluation.

4.7 Why did this evaluation take one year?

In June 2017, we started gathering information about solutions related to Android app security. By mid-June, we had performed both shallow and deep selection of tools. At that time, Ghera repository had only 25 benchmarks and we evaluated the selected tools on these benchmarks.

In July 2017, we started adding 17 new benchmarks to Ghera. In addition, we upgraded Ghera benchmarks to work with Android Studio 3.0.1 (from Android Studio 2.3.3). So, in October 2017, we evaluated the tools against 25 revised benchmarks and 17 new benchmarks.

In November 2017, based on community feedback, we started adding secure apps to Ghera benchmarks along with automation support for functional testing of benchmarks. This exercise forced us to change some of the existing benchmarks. We completed this exercise in January 2018.

To ensure our findings would be fair, current, and useful, we decided to re-evaluate the tools as the benchmarks had changed considerably, i.e., a whole new set of secure apps along with few revamped benign and malicious apps. We completed this evaluation in February 2018.

While the time needed for tools evaluation — conducting the experiment and analyzing the results – was no more than two months, changes to Ghera and consequent repetitions of the evaluation prolonged the evaluation.

Between February and May 2018, we designed and performed experiments to measure the representativeness of Ghera benchmarks. Remote location of Androzoo, sequential downloading of apps from Androzoo, processing of 339K apps, and repetition of the experiment due to automation errors contributed to prolonging this exercise.

5 Related Work

Android security has generated considerable interest in the past few years. This is evident by the sheer number of research efforts exploring Android security. Sufatrio et al. summarized such efforts by creating a taxonomy of existing techniques to secure the Android ecosystem [64]. They distilled the state of the art in Android security research and identified potential future research directions. While their effort assessed existing techniques theoretically on the merit of reported details and results, we evaluated existing tools empirically by executing them against a common set of benchmarks. So, these efforts are complementary.

In 2016, Reaves et al. systematized Android security research that analyzes applications by considering Android app analysis tools that were published in 17 top venues since 2010 [59]. Also, they empirically evaluated the usability and applicability of results of 7 Android app analysis tools. In contrast, our evaluation considered 19 tools that detected vulnerabilities and malicious behaviors. They used benchmarks from DroidBench [16], 6 vulnerable real world apps, and top 10 financial apps from Google Play store. While DroidBench benchmarks and vulnerable real world apps were authentic (i.e., they did contain vulnerabilities), this was not the case with the financial apps. In contrast, all of the 42 Ghera benchmarks used in our evaluation were synthetic but authentic. Further, while DroidBench focuses on ICC related vulnerabilities and use of taint analysis for vulnerability detection, Ghera is agnostic to the underlying techniques and contains vulnerabilities related to ICC and other aspects of Android platform. While Reaves et al. focused usability of tools (i.e., how well does the tool work in practice?), our evaluation focused on more on the effectiveness of tools in detecting known vulnerabilities and malicious behavior and less on the usability of tools. Even with these differences, the effort by Reaves et al. is the most closely related to this evaluation.

Sadhegi et al. conducted an exhaustive literature review of the use of program analysis techniques to address issues related to Android security [61]. They identified trends, patterns, and gaps in existing literature along with the challenges and opportunities for future research. In comparison, our evaluation also exposes gaps in existing tools. However, it does so empirically while being agnostic to techniques underlying the tools, i.e., not limited to program analysis.

Zhou et al. conducted a systematic study of the installation, activation, and payloads of 1260 malware samples collected from August 2010 thru 2011 [70]. They characterized the behavior and evolution of malware. In contrast, our evaluation is focused on the ability of tools to detect vulnerable and malicious behaviors.

6 Evaluation Artifacts

The code and input data used in the evaluation of representativeness of Ghera benchmarks are available at https://bitbucket.org/secure-it-i/evaluate-representativeness/src/rekha-may2018 along with the output data from the evaluation and the instructions to repeat the evaluation.

A copy of specific versions of offline tools used in tools evaluation are available at https://bitbucket.org/secure-it-i/may2018 along with tool output from the evaluation. Specifically, *vulevals* and *secevals* folders contain artifacts from the evaluation of vulnerability detection tools using benign apps and secure apps from Ghera, respectively. *malevals* folder contains artifacts from the evaluation of malicious behavior detection tools using malicious apps from Ghera. The repository also contains the automation scripts used in the evaluation along with the instructions to repeat the evaluation.

7 Future Work

Here are few ways to extend this effort to help the Android developer community.

1. Evaluate paid security analysis tools by partnering with tool vendors, e.g., AppRay [8], IBM AppScan [18], Klocwork [20].
2. Evaluate freely available Android security analysis tools that have not been considered in this tools evaluation, e.g., ConDroid [29], Sparta [43], StaDyna [69].
3. Explore different modes and configurations of evaluated tools (e.g., Amandroid) to evaluate their effect on the accuracy of verdicts.
4. Extend tools evaluation to consider any new lean and fat (vulnerable real-world apps) benchmarks added to Ghera repository.
5. Create and maintain an online dashboard based on results from tools evaluation. Also, publish resulting artifacts via a public repository. This can help app developers identify security tools that are well suited to check their apps for vulnerabilities and tool developers assess how well their tools fare against both known (regression) and new vulnerabilities and exploits.

8 Summary

When we started this evaluation, we were expecting many security analysis tools would detect many of the vulnerabilities considered in the evaluation. The rationale for our expectation was the vulnerabilities were known a priori and there have been explosion of efforts in recent year to develop tools and techniques for Android app security analysis.

After our evaluation, we observed that most of the tools and techniques are able to detect only a small fraction of the vulnerabilities considered in the evaluation. Further, all tools put together were unable to detect all of the considered vulnerabilities.

These observations suggest that, if current and new security analysis tools and techniques are to be helpful in building secure Android apps, then we need to ensure these tools and techniques are effective in detecting vulnerabilities, starting with known vulnerabilities. One way to achieve this is by building an open free public corpus of known vulnerabilities in verifiable and demonstrable form and using the corpus to evaluate the effectiveness of security analysis tools and techniques.

Acknowledgement

We would like to thank various readers who read the pre-print of this manuscript and suggested corrections and edits to improve the manuscript.

References

[1] A collection of android security related resources. https://github.com/ashishb/android-security-awesome. Accessed: 01-May-2018.

[2] AndroBugs Framework. https://github.com/AndroBugs/AndroBugs_Framework. Accessed: 21-Nov-2017.

[3] Android developer documentation - Binder. https://developer.android.com/reference/android/os/Binder.html#getCallingPid(). Accessed: 01-Jun-2018.

[4] Android developer documentation - Content Provider. https://developer.android.com/reference/android/content/ContentProvider.html#call(java.lang.String,%20java.lang.String,%20android.os.Bundle). Accessed: 07-Mar-2018.

[5] Android Intent with Chrome. https://developer.chrome.com/multidevice/android/intents. Accessed: 29-May-2018.

[6] Android Security Tips. https://developer.android.com/training/articles/security-tips. Accessed: 01-Jun-2017.

[7] AppCritique: Online Vulnerability Detection Tool. https://appcritique.boozallen.com/. Accessed: 21-Nov-2017.

[8] AppRay. http://app-ray.co/. Accessed: 04-Jun-2018.

[9] CuckooDroid: Automated Android Malware Analysis. https://github.com/idanr1986/cuckoo-droid. Accessed: 01-May-2018.

[10] CVE-2014-0806 : Geolocation disclosure. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0806. Accessed: 29-May-2018.

[11] CVE-2014-1566 : Leak information to SD card. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1566. Accessed: 29-May-2018.

[12] CVE-2014-1977 : Weak Permissions. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1977. Accessed: 29-May-2018.

[13] CVE-2014-5319 : Directory traversal vulnerability. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-5319. Accessed: 29-May-2018.

[14] CVE-2014-8507 : SQL Injection vulnerability. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-8507. Accessed: 29-May-2018.

[15] CVE-2014-8609 : Android Settings application privilege leakage vulnerability. http://seclists.org/fulldisclosure/2014/Nov/81. Accessed: 29-May-2018.

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

[17] Drozer. https://github.com/mwrlabs/drozer/. Accessed: 20-Apr-2018.

[18] IBM App Scan. https://www.ibm.com/us-en/marketplace/ibm-appscan-source. Accessed: 01-June-2018.

[19] Joint Advanced Application Defect Assessment for Android Application (JAADAS). https://github.com/flankerhqd/JAADAS. Accessed: 21-Nov-2017.

[20] Kolcwork. https://www.roguewave.com/products-services/klocwork/detection/android. Accessed: 01-June-2018.

[21] Maldrolyzer. https://github.com/maldroid/maldrolyzer. Accessed: 21-Nov-2017.

[22] Mobile Security Wiki. https://mobilesecuritywiki.com/. Accessed: 01-May-2018.

[23] NVISO ApkScan. https://apkscan.nviso.be/. Accessed: 01-May-2018.

[24] SRT:AppGuard. http://www.srt-appguard.com/en/. Accessed: 04-Jun-2018.

[25] Tracedroid. http://tracedroid.few.vu.nl/. Accessed: 01-May-2018.

[26] Virus Total. https://www.virustotal.com/. Accessed: 21-Nov-2017.

[27] Ajin Abraham, Dominik Schelecht, and Matan Dobrushin. Mobile Security Framework. https://github.com/MobSF/Mobile-Security-Framework-MobSF. Accessed: 21-Nov-2017.

[28] 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.

[29] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 59:1–59:11. ACM, 2012.

[30] 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.

[31] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: Analyzing the android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 217–228. ACM, 2012. http://pscout.csl.toronto.edu/, Accessed: 21-Nov-2017.

[32] 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.

[33] 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, New York, NY, USA, 2017. ACM. https://github.com/dialdroid-android, Accessed: 05-May-2018.

[34] Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *22nd USENIX Security Symposium (USENIX Security '13)*. USENIX, 2013. http://www.flaskdroid.org/index.html%3Fpage_id=2.html, Accessed: 04-Jun-2018.

[35] 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.

[36] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, pages 239–252. ACM, 2011.

[37] Erika Chin and David A. Wagner. Bifocals: Analyzing webview vulnerabilities in android applications. In *WISA*, pages 138–159. Springer, 2013.

[38] Thomas Debize. AndroWarn : Yet Another Static Code Analyzer for malicious Android applications. https://github.com/maaaaz/androwarn/. Accessed: 21-Nov-2017.

[39] Luke Deshotels, Vivek Notani, and Arun Lakhotia. Droidlegacy: Automated familial classification of android malware. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, pages 3:1–3:12. ACM, 2014.

[40] DevKnox. DevKnox - Security Plugin for Android Studio. https://devknox.io/. Accessed: 21-Nov-2017.

[41] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, pages 73–84. ACM, 2013.

[42] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, pages 393–407. USENIX Association, 2010.

[43] Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward X. Wu. Collaborative verification of information flow for a high-assurance app store. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1092–1104. ACM, 2014. https://www.cs.washington.edu/sparta, Accessed : 01-June-2018.

[44] 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.

[45] Adam Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. Scandroid: Automated security certification of android applications. 2009. https://github.com/SCanDroid/SCanDroid, Accessed: 04-Jun-2018.

[46] Joshua Garcia, Mahmoud Hammad, Negar Ghorbani, and Sam Malek. Automatic generation of inter-component communication exploits for android applications. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 661–671. ACM, 2017. http://seal.ics.uci.edu/projects/letterbomb/, Accessed: 24-Apr-2018.

[47] Michael I. Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin" Rinard. Information-flow analysis of Android applications in DroidSafe. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS)*, 2015. https://github.com/MIT-PAC/droidsafe-src, Accessed: 21-Apr-2018.

[48] Matthew Green and Matthew Smith. Developers are not the enemy!: The need for usable security apis. *IEEE Security and Privacy*, pages 40–46, 2016.

[49] Yunhan Jack Jia, Qi Alfred Chen, Yikai Lin, Chao Kong, and Zhuoqing Morley Mao. Open doors for bob and mallory: Open port usage in android apps and security implications. In *EuroS&P*, pages 190–203. IEEE, 2017.

[50] Juan Heguiabehere Joaquín Rinaudo. Marvin Static Analyzer. https://github.com/programa-stic/Marvin-static-Analyzer. Accessed: 21-Nov-2017.

[51] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*. ACM, 2014. https://www.cert.org/secure-coding/tools/didfail.cfm, Accessed: 21-Apr-2018.

[52] 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.

[53] LinkedIn. Quick Android Review Kit. https://github.com/linkedin/qark/. Accessed: 21-Nov-2017.

[54] Federico Maggi, Andrea Valdi, and Stefano Zanero. Andrototal: A flexible, scalable toolbox and service for testing mobile malware detectors. In *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, pages 49–54. ACM, 2013. http://andrototal.org/, Accessed: 21-Nov-2017.

[55] 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.

[56] Mitre Corporation. Common vulnerabilities and exposures. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=Android, 2017. Accessed: 08-Jun-2017.

[57] Adwait Nadkarni, Benjamin Andow, William Enck, and Somesh Jha. Practical DIFC enforcement on android. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1119–1136. USENIX Association, 2016. https://wspr.csc.ncsu.edu/aquifer/, Accessed : 01-June-2018.

[58] Duc Cuong Nguyen, Dominik Wermke, Yasemin Acar, Michael Backes, Charles Weir, and Sascha Fahl. A stitch in time: Supporting android developers in writingsecure code. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1065–1077. ACM, 2017. https://plugins.jetbrains.com/plugin/9497-fixdroid, Accessed: 21-Apr-2018.

[59] Bradley Reaves, Jasmine Bowers, Sigmund Albert Gorski III, Olabode Anise, Rahul Bobhate, Raymond Cho, Hiranava Das, Sharique Hussain, Hamza Karachiwala, Nolen Scaife, Byron Wright, Kevin Butler, William Enck, and Patrick Traynor. *droid: Assessment and evaluation of android application analysis tools. *ACM Comput. Surv.*, pages 55:1–55:30, 2016.

[60] Chuangang Ren, Yulong Zhang, Hui Xue, Tao Wei, and Peng Liu. Towards discovering and understanding task hijacking in android. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 945–959. USENIX Association, 2015.

[61] Alireza Sadeghi, Hamid Bagheri, Joshua Garcia, and Sam Malek. A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. *IEEE Transactions on Software Engineering*, pages 492–530, 2017.

[62] Feng Shen, Namita Vishnubhotla, Chirag Todarka, Mohit Arora, Babu Dhandapani, Eric John Lehner, Steven Y. Ko, and Lukasz Ziarek. Information flows as a permission mechanism. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 515–526. ACM, 2014. http://blueseal.cse.buffalo.edu/multiflow.html, Accessed : 01-June-2018.

[63] David Sounthiraraj, Justin Sahs, Garret Greenwood, Zhiqiang Lin, and Latifur Khan. Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps. In *In Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS)*, 2014. https://github.com/utds3lab/SMVHunter, Accessed: 10-Jun-2018.

[64] Sufatrio, Darell J. J. Tan, Tong-Wei Chua, and Vrizlynn L. L. Thing. Securing android: A survey, taxonomy, and challenges. *ACM Comput. Surv.*, pages 58:1–58:45, 2015.

[65] Vasant Tendulkar and William Enck. An application package configuration approach to mitigating android SSL vulnerabilities. *CoRR*, abs/1410.7745, 2014.

[66] 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.

[67] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the 21st USENIX Conference on Security Symposium*, pages 27–27. USENIX Association, 2012. https://github.com/xurubin/aurasium, Accessed : 01-June-2018.

[68] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. Droid-sec: Deep learning in android malware detection. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 371–372. ACM, 2014. http://www.droid-sec.com/,http://analysis.droid-sec.com/,Accessed: 21-Apr-2018.

[69] Yury Zhauniarovich, Maqsood Ahmad, Olga Gadyatskaya, Bruno Crispo, and Fabio Massacci. Stadyna: Addressing the problem of dynamic code updates in the security analysis of android applications. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 37–48, New York, NY, USA, 2015. ACM. https://github.com/zyrikby/StaDynA, Accessed: 11-Jun-2018.

[70] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pages 95–109. IEEE Computer Society, 2012.

A Catalog of Considered Vulnerabilities

In this catalog, we briefly describe the 42 vulnerabilities (along with their canonical references) captured in Ghera that were used in this evaluation. Few vulnerabilities have generic references as they were discovered by Ghera authors while reading the security guidelines available as part of Android documentation [6]. Please refer to [55] for details about the repository and the initial set of vulnerabilities.

Acknowledgement

The authors would like to thank Aditya Narkar and Nasik Muhammad Nafi for their help in implementing 17 new benchmarks that are being cataloged as Ghera benchmarks for the first time in this paper and were used in the evaluations described in this paper.

A.1 Crypto

Crypto APIs enable Android apps to encrypt, decrypt information, and manage cryptographic keys.

C1 The result of encrypting a message twice using Block Cipher algorithm in *ECB mode* is the message itself. So, apps using Block Cipher algorithm in ECB mode (explicitly or due to default on Android platform) can *leak information* [41].

C2 Encryption using Block Cipher algorithm in CBC mode with a *constant Initialization Vector (IV)* can be broken by recovering the constant IV using plain text attack. So, apps using such encryption can *leak information* [41].

C3 Password based encryption (PBE) uses a salt to generate password based encryption key. If the salt is constant, the encryption key can be recovered with knowledge about the password. Hence, apps *using PBE with constant salt can leak information* [41].

C4 Cipher APIs rely on unique keys to encrypt information. If such keys are embedded in the app's code, then attackers can recover such keys from the app's code. Hence, such apps are susceptible to both information leak and data injection [41].

A.2 Inter Component Communication (ICC)

Android apps are composed of four basic kinds of components: 1) *Activity* components display the user interface, 2) *Service* components perform background operations, 3) *Broadcast Receiver* components receive event notifications and act on those notifications, and 4) *Content Provider* components manage app data. Communication between components in an app and in different apps (e.g., to perform specific actions, share information) is facilitated via exchange of *Intent*s. Components specify their ability to process specific kinds of intents by using *intent-filters*.

I1 Android apps can *dynamically register broadcast receivers* at runtime. Such receivers are automatically *exported without any access restrictions* and, hence, can be accessed via ICC and exploited to perform unintended actions [36].

I2 A component can use a *pending intent* to allow another component to perform an action on its behalf. When a pending intent is empty (i.e., does not specify an action), it can be seeded (via interception) with an unintended action to be executed on behalf of originating component [15].

I3 To perform an action (e.g., send email), users choose an activity/app from a list of *activities ordered by priority*. By using appropriate priorities, activities can gain unintended privilege over other activities [36].

I4 Implicit intents are processed by any *qualifying service determined by intent filters* as opposed to a specific explicitly named service. In case of multiple qualifying services, the service with the highest priority processes the intent. By registering appropriate intent filters and by using appropriate priorities, services can gain unintended access to implicit intents [36].

I5 Pending intents can contain *implicit intents*. Hence, services processing of the implicit intent contained in a pending intent are vulnerable as in I4 [15].

I6 Apps can use `path-permissions` to control access to data exposed by content provider. These permissions control access to a folder and not to its subfolders/descendants and their contents. Incorrectly assuming the extent of these permissions can lead to information leak (read) and data injection (write) [6].

I7 Components that process *implicit intents* are by default *exported without any access restrictions*. If such a component processes intents without verifying their authenticity (e.g., source) and handles sensitive information or performs sensitive operations in response to implicit intents, then it can leak information or perform unintended actions [36].

I8 Broadcast receivers registered for *system intents* (e.g., low memory) from Android platform are by default *exported without any access restrictions*. If such a receiver services intents without verifying the authenticity of intents (e.g., requested action), then it may be vulnerable like the components in I7 [36].

I9 Broadcast receivers respond to *ordered broadcasts* in the order of priority. By using appropriate priorities, receivers can modify such broadcasts to perform unintended actions [36].

I10 *Sticky broadcast intents* are delivered to every registered receiver and saved in the system to be delivered to receivers that register in the future. When such an intent is re-broadcasted with modification, it replaces the saved intent in the system. This can lead to information leak and data injection [36].

I11 Every activity is launched in a *task*, a collection (stack) of activities. An activity can declare its *affinity* to be started in a specific task under certain conditions. When the user navigates away from an activity X via the *back* button, the activity below X on X's task is displayed. This behavior along with task affinity and specific start order — malicious activity started before benign activity — can be used to mount a phishing attack [60].

I12 When an activity from a *task in the background* (i.e., none of its activities are being displayed) is resumed, the activity at the top of the task (and not the resumed activity) is displayed. This behavior along with task affinity and specific start order — malicious activity started after benign activity — can be used to mount a phishing attack [60].

I13 When a *launcher activity* is started, its task is created and it is added as the first activity of the task. If the task of a launcher activity already exists with other activities in it, then its task is brought to the foreground but the launcher activity is not started. This behavior along with task affinity and specific start order — malicious activity started before benign activity — can be used to mount a phishing attack [60].

I14 In addition to declaring task affinity, an activity can request that it be moved to the affine task when the task is created or moved to the foreground (known as *task reparenting*). This behavior along with task affinity and specific start order — malicious activity started before benign activity — can be used to mount a denial-of-service or a phishing attack [60].

I15 Apps can request permission to *perform privileged operations* (e.g., send SMS) and offer interfaces (e.g., broadcast receiver) thru which these privileged operations can be triggered. Unless appropriately protected, these interfaces can be exploited to perform operation without sufficient permissions [36].

I16 The `call` method of Content Provider API can be used to invoke any provider-defined method. With a reference to a content provider, this method can be invoked without any restriction leading to both information leak and data injection [4].

A.3 Networking

Networking APIs allow Android apps to communicate over the network via multiple protocols.

N1 Apps can *open server sockets* to listen to connections from clients; typically, remote servers. If such sockets are not appropriately protected, then they can lead to *information leak* and *data injection* [49].

N2 Apps can communicate with remote servers via *insecure TCP/IP connections*. Such scenarios are susceptible to MitM attacks [49].

A.4 Permissions

In addition to system-defined permissions, Android apps can create and use custom permissions. These permissions can be combined with the available four protection levels (i.e., normal, dangerous, signature, signatureOrSystem) to control access to various features and services.

P1 Permissions with normal protection level are automatically granted to requesting apps during installation. Consequently, any component or its interface protected by such "normal" permissions will be accessible to every installed app [12].

A.5 Storage

Android provides two basic options to store app data.

1. *Internal Storage* is best suited to store files private to apps. Every time an app is uninstalled, its internal storage is purged. Starting with Android 7.0 (API 24), files stored in internal storage cannot be shared with and accessed by other apps [6].

2. *External Storage* is best suited to store files that are to be shared with other apps or persisted even after an app is uninstalled. While public directories are accessible by all apps, app-specific directories are accessible only by corresponding apps or other apps with appropriate permission [6].

S1 *Files stored in public directories on external storage* can be accessed by an app with appropriate permission to access external storage. This aspect can be used to tamper data via data injection [6].

S2 The same aspect from S1 can lead to information leak [6].

S3 Apps can *accept paths to files in the external storage from external sources and use them without sanitizing them*. A well-crafted file path can be used to read, write, or execute files in the app's private directory on external storage (directory traversal attack) [13].

S4 Apps can *copy data from internal storage to external storage*. This could lead to information leak if such apps accept input from untrusted sources to determine the data to be copied [11].

S5 `SQLiteDatabase.rawQuery()` method can be used by apps to serve data queries. If such uses rely on external inputs and use non-parameterized SQL queries, then they are susceptible to *sql injection* attacks [14].

S6 Content Provider API support `selectionArgs` parameter in various data access operations to separate selection criteria and selection parameters. App that do not use this parameter are be susceptible to *sql injection* attacks [14].

A.6 System

System APIs enable Android apps access low level features of the Android platform like process management, thread management, runtime permissions, etc.

Every Android app runs in its own process with a unique Process ID (PID) and a User ID (UID). All components in an app run in the same process. A permission can be granted to an app at installation time or at run time. All components inherit the permissions granted to the containing app at installation time. If a component in an app is protected by a permission, only components that have been granted this permission can communicate with the protected component.

Y1 During IPC, `checkCallingOrSelfPermission` method can be used to check if the calling/caller process or the called process has permission P. If a component with permission P uses this method to check if the calling component has permission P, then improper use of this method can *leak privilege* when the calling component does not have permission P [3].

Y2 `checkPermission` method can be used to check if the given permission is allowed for the given PID and UID pair. `getCallingPID` and `getCallingUID` methods of Binder API can be used to retrieve the PID and UID of the calling process. In certain situations, they return PID/UID of the called process. So, improper use of these methods by a called component with given permission can *leak privilege* [3].

Y3 During IPC, `enforceCallingOrSelfPermission` method can be used to check if the calling/caller process or the called process has permission P. Like in Y1, improper use of this method can *leak privilege* [3].

Y4 `enforcePermission` method can be used to check if the given permission is allowed for the given PID and UID pair. Like in Y2, improper use of this method along with `getCallingPID` and `getCallingUID` can *leak privilege* [3].

A.7 Web

Web APIs allow Android apps to interact with web servers both with and without SSL/TLS, display web content through *WebView* widget, and control navigation between web pages via *WebViewClient* class.

W1 Apps connecting to remote servers via HTTP (as opposed to HTTPS) are susceptible to *information theft* via *Man-in-the-Middle (MitM)* attacks [65].

W2 Apps can employ `HostnameVerifier` interface to perform custom checks on host name when using SSL/TLS for secure communication. If these checks are incorrect, apps can end up connecting to malicious servers and be targets of malicious actions [65].

W3 In secure communication, apps employ `TrustManager` interface to check the validity and trustworthiness of presented certificates. Like in W2, if these checks are incorrect, apps can end up trusting certificates from malicious servers and be targets of malicious actions [65].

W4 Intents can be embedded in URIs. Apps that do not handle such intents safely (e.g., check intended app) can *leak information* [5].

W5 Web pages can access information local to the device (e.g., GPS location). Apps that allow such access without explicit user permission can *leak information* [10].

W6 When `WebView` is used to display web content, JavaScript code executed as part of the web content is executed with the permissions of the host app. Without proper checks, malicious JavaScript code can get access to the app's resources e.g. private files [37].

W7 When loading content over a secure connection via `WebView`, host app is notified of SSL errors via `WebViewClient`. Apps ignoring such errors can enable MitM attacks [65].

W8 When a web resource (e.g., CSS file) is loaded in `WebView`, the load request can be validated in `shouldInterceptRequest` method of `WebViewClient`. Apps failing to validate such requests can allow loading of malicious content [37].

W9 When a web page is loaded into `WebView`, the load request can be validated in `shouldOverridUrlLoading` method of `WebViewClient`. Apps failing to validate such requests can allow loading of malicious content [37].