# Profiling Android Vulnerabilities

Matthieu Jimenez, Mike Papadakis, Tegawendé F. Bissyandé and Jacques Klein
Interdisciplinary Centre for Security, Reliability and Trust (SnT)
University of Luxembourg
Email: {firstname.lastname}@uni.lu

***Abstract*—In widely used mobile operating systems a single vulnerability can threaten the security and privacy of billions of users. Therefore, identifying vulnerabilities and fortifying software systems requires constant attention and effort. However, this is costly and it is almost impossible to analyse an entire code base. Thus, it is necessary to prioritize efforts towards the most likely vulnerable areas. A first step in identifying these areas is to profile vulnerabilities based on previously reported ones. To investigate this, we performed a manual analysis of Android vulnerabilities, as reported in the National Vulnerability Database for the period 2008 to 2014. In our analysis, we identified a comprehensive list of issues leading to Android vulnerabilities. We also point out characteristics of the locations where vulnerabilities reside, the complexity of these locations and the complexity to fix the vulnerabilities. To enable future research, we make available all of our data.**

***Index Terms*—Software Security, Complexity, Android, Vulnerabilities, Common Vulnerability Exposures**

## I. Introduction

Smartphones have a wide range of applications that go beyond the simple task of making phone calls. Thus, simple defects can lead to vulnerabilities that cause high-profile security mishaps with the potential of impacting billions of users.

Current research on software vulnerabilities has largely focused on predicting which artifacts (e.g. components, classes or files) are vulnerable [1]–[3]. To perform such predictions, researchers have built classification models based on several metrics that were found empirically to estimate the probability that a vulnerability is present in a given artifact. However, very few works try to profile them and none of them apply to the specific case of mobile operating systems.

In view of this, the objective of our research is to improve the understanding of vulnerabilities, which impacting the Android operating system. To do so, we manually investigate known vulnerabilities and more specifically the code changes that were made to fix them. To this end, we focus on three properties: their root causes, their complexity and their location.

Since project leaders, developers and code reviewers are often interested in the root causes of vulnerabilities, we build a taxonomy of the issues causing vulnerabilities. In the remainder of the paper, we refer to those root causes as issues.

To study real Android vulnerabilities, we consider the NIST National Vulnerability Database (NVD). It is known as the largest public repository that encompasses details regarding Common Vulnerability Exposures (CVEs). Our experience in crawling this data has highlighted challenges for exploiting the NVD in its current form to perform a comprehensive and large-scale study of vulnerabilities. We found that NVD metadata is suitable for computing high-level statistics about the vulnerabilities but problematic in gathering and linking them with the actual code.

Our study has thus required intensive manual work to collect the missing/hidden information on Android vulnerabilities recorded in the NVD. This information includes the often missing links to the relevant patches, as well as some valuable information that are hidden in the description or in the natural language, of the vulnerability reports.

Based on our study, which includes vulnerabilities reported between 2008 and 2014, we build a taxonomy of the issues that are reported in CVEs. We also characterize the fixes made to correct the vulnerabilities by summarizing (1) their origin (2) the complexity of the code where the vulnerabilities are found (3) the complexity of the fixes themselves and (4) the different actions that were necessary to fix the vulnerabilities.

The present study makes the following contributions:

- We profile, taxonomize and present some characteristics of the Android vulnerabilities.
- We provide a collection of Android vulnerabilities that are reported in the NVD Database, which has been enriched with information mined outside NVD. This collection is available at http://www.jimenez.lu/Research/
- We provide guidelines for improving the NVD in order to enable systematic meta-analysis and other analysis studies related to vulnerabilities.

The remainder of this paper is organized as follows: Section II introduces Android and vulnerabilities. Section III motivates our work. Section IV describes the methodology used to obtain our results, presented in Section V. The approach and its possible improvements are discussed in Section VI. Finally, we examine related work in Section VII before concluding in Section VIII.

## II. Background

This section presents background material for defining the context of our study. Specifically, we give a brief description of the Android operating system in Section (II-A). We then provide details regarding Common Vulnerability Exposures (II-B) and the National Vulnerable Database (II-C). Finally, a definition of vulnerabilities is given (II-D).

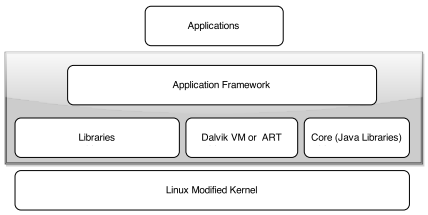



Fig. 1: The four layers of the Android operating system.

### A. Android

The Android operating system is composed of four layers, which are depicted by Figure 1. The first layer is a modified Linux kernel. The layer on top of the Linux kernel encompasses C/C++ Libraries, Java Core Libraries and the Dalvik Virtual Machine (DVM), which has recently been renamed as Android Runtime (ART). The DVM and the ART are used to run user applications and embedded applications written in the Java programming language [4]. Finally, the third layer contains built-in Android applications while the last layer is for user applications.

Since Android is Open-Source[1], practitioners and researchers can adapt it for their specific needs. In this study, we focus on vulnerabilities related to the Android system, i.e., the second (libraries and DVM) and third (built-in applications) layers (highlighted in gray on Figure 1). The vulnerabilities affecting the two other layers, i.e., the kernel or the applications, were not considered since these vulnerabilities are respectively reported as Linux or application specific.

### B. Common Vulnerability Exposures (CVE)

CVE is a system that provides a reference for all publicly reported vulnerabilities. This system is managed by the National Institute of Standard and Technology (NIST). To easily share data related to vulnerabilities, each one of the identified and accepted vulnerabilities receives a unique identifier. To get a CVE identifier, a vulnerability must be reported to a certified authority that will approve and emit the number.

### C. National Vulnerability Database (NVD)

The National Vulnerability Database is the U.S. government database for all reported CVEs. Besides the various information that is required by the CVE system, NVD contains additional information like its Common Vulnerability Score, its type of vulnerability (Common Weakness Enumeration CWE [7]) and sometimes links to existing patches. In March 2016, over 75.000 vulnerabilities were reported in the NVD, making it the largest public database of vulnerabilities.

### D. Vulnerabilities

Software vulnerabilities range from simple bugs to insufficient counter measures, which provide malicious users the opportunity to attack a system or an application. The literature provides several definitions for vulnerabilities, e.g., they can be referred as bugs, security bugs [8] or software "weaknesses". In this article, we use the terminology of the CVE system to define vulnerabilities:

"*An information security "vulnerability" is a mistake in software that can be directly used by a hacker to gain access to a system or network.*" [9]

In the present paper, we use the CVE system as a dictionary for the Android vulnerabilities. Thus, all the vulnerabilities used in the present study have been accepted and received a CVE number, meaning that an independent authority has certified that these vulnerabilities are real. We therefore consider them as real vulnerabilities.

## III. Motivation

Android is currently the most widely used operating system for hand-held devices, such as smartphones, and is trending in other embedded system products, such as TV sets, watches and Internet boxes. In the first quarter of 2015 334.4 million Android devices were sold, which represents 78% of the market [10]. These sales also exceed, by far, the number of personal computers that were sold in the entire year 2014 (308 million [11]). A direct consequence is that a single vulnerability of the Android system, or within one of its embedded libraries, can impact a huge number of users.

A way to avoid this situation is to find vulnerabilities before release time. To do so, a strong focus on reviewing and fortifying the existing code needs to be made by developer teams. However it is not possible to look at the entire code base especially when taking into consideration that finding vulnerabilities requires a specific mindset [12]. A prioritization during the reviewing process is therefore necessary. If we assume that vulnerability prediction techniques are good indicators of vulnerable code [13], these techniques are costly and not well suited for the Android environment and its specificities. Indeed the Android code base is heavily fragmented and split in over 1,000 git repositories, which have some redundancy [6]. This hinders both the applicability and effectiveness of the techniques as they require to be launched on every repository independently. Also since the code base is fragmented, very few elements can be learned by the techniques. Thus, to reduce the scope of the research, a first and really instructive step consists of profiling vulnerabilities so that we increase our understanding of them. In other words, we need to provide answers specific to the application context, i.e., Android, and to the following questions: *What are they? Are they hard to fix? Where are they coming from?*.

By investigating these questions, we can help developers and code reviewers to identify them, and focus their efforts on parts that are matching our shaped profile.

## IV. Methodology

Our analysis is based on all vulnerabilities reported between 2008 and 2014 in the NVD database and are related to Android. In total we have 42 vulnerabilities. Our approach for

[1]Android source code is available on Github [5] and on GoogleSource [6].

this analysis can be summarized in the following steps. We first manually mine the NVD database to find the actual reported issues and their corresponding patches. We then classify the vulnerabilities. Finally, we analyze the vulnerabilities.

The described process is designed to answer the three following Research Questions (RQs):

- RQ1: What kind of issues cause vulnerabilities? Can we categorize them?
- RQ2: In which components the Android vulnerabilities are located?
- RQ3: How vulnerabilities are fixed?

### A. Vulnerability & Issue Mining

The first step of our approach is to retrieve Android Vulnerabilities from the CVE-NVD database. To this end, we manually mine the database to look for the vulnerabilities related to Android. Unfortunately and despite that CVE-NVD provides a lot of useful information, this is not enough to answer our RQs. In particular, we are lacking information for the following three elements: Vulnerable components, Bug reports, Patches. As a result, we have to explore all the external resources provided via links within CVE.

While browsing all the links present in our set of CVE, we found out that about 20% of those links were dead and 72% in the case of links pointing to patches or bug reports. After some experiments, we realized that this issue is not restricted to Android. Indeed, only 30% of the link declared in all CVE are still valid and that this number was even lower when only studying patch links.

This complicates the data mining process as it implies finding the new location of the resource. We solved this issue by performing extensive manual search of the related repositories, i.e., browsing of the git commit history, looking for keywords present in the vulnerability description. This allowed us to retrieve patches for 31 vulnerabilities.

### B. Taxonomy of the Issues Related to Android Vulnerabilities

Once all the sought patches and bug reports have been retrieved, we categorize the related issues. Here, an issue denotes a problem impacting one or more files within a commit (patch). Since categorizing an issue requires its understanding, we only consider vulnerabilities for which we are able to find related patch(es).

The aim of categorization is to obtain a taxonomy of the issues related to Android vulnerabilities. Many researches have been conducted to categorize bugs, e.g., [14], [15]. However, these categories were quite impractical since most of the issues we found, were either A) didn't fit into a single category or B) were fitting into many. To solve this, we use mind mapping as a way to extend the existing bug taxonomies to fit to our needs. Mind mapping is a technique suggested by Vijayaraghavan et al. [16] that constructs a taxonomy by incrementally considering one issue after the other. This process allows the taxonomy to evolve over time, i.e., to become more complete with each newly considered issue.

To categorize appropriately the considered issues, we apply the following methodology. We first analyze the commits and/or patches along with their related source code. We then read the bug and vulnerability reports and their available comments. Next, we cross the information retrieved and try to answer the following question: What was the problem with the code, what kind of mistake led to the vulnerability and at which state of the project it was made. Finally, we decide which category fits best for the studied issue. Note that when an issue doesn't fit into an existing category, a new category is created. After categorizing all the issues, we proceed with the analysis of the components and changes.

### C. Analysis

After categorizing all the issues, we proceed with the analysis of the components and changes.

*1) Component Analysis:* We analyze the vulnerable components to identify information related to Android vulnerabilities. To this end, we focus on two aspects of the vulnerable component that we detail shortly. The first one is its purpose and the second one is its complexity compared to the other components.

*a) Purpose of the Component.:* This information indicates which part of the Android system tends to have the most vulnerabilities. To this end, we design top level categories of purposes and classify the vulnerable components, according to them.

*b) Complexity Analysis of a Component.:* Chowdhury and Zulkernine [17] in their study show that there is a correlation between code complexity and the appearance of vulnerabilities. Thus, we measure the complexity of the functions that were changed to remove the vulnerability and compare it with the complexity of the other functions located in the same source code file trying to see if we find the same correlation with Android vulnerabilities. The metric used for this analysis is the McCabe Cyclomatic complexity [18].

*2) Change Analysis:* Once the issue and the component analyzed, we investigate the complexity of the vulnerability itself. We first analyze the patches and synthesize their contents through keywords. We then use the following metrics to measure the complexity of the vulnerabilities:

a) *Number of commits:* Number of commits required to correct the vulnerability.
b) *Number of modified files:* Number of modified, deleted or created files.
c) *Number of Changed Lines of Executed Code (CLoEC):* Number of modified, added or deleted lines of code. Empty lines and bracket lines are ignored and commands spanning over multiple lines are measured as a single line. This metric can also be referred as Code Churn in the literature with the difference that we don't take into account modification on commented lines.
d) *Cyclomatic Complexity of Change (CyCC) [19]* The CyCC metric measures the number of linearly independent sequences of changed statements from entry to exit in a changed program. It represents the least complex changes

needed to remove the studied defect [19]. It is computed the same way that the Cyclomatic Complexity except that it uses the Change Sequence Graph (CSG [20]) instead of the Control Flow graph. The CSG is a control flow Graph that only contains basic blocks that were changed. A Cycc superior to 2 indicates that the change was complex as it involves the addition or modification of more than two independent linearly path.

We chose the CLoEC metric since it represents a raw measure of the number of code changes. In contrast, CyCC metric quantifies the complexity of changes by considering only the independent code places that the developer has to consider. Thus, both CLoEC and CyCC quantify the difficulty of fixing a vulnerability.

## V. Experimental Results

This section presents the results obtained while applying the methodology presented in Section IV. As stated before, from a total number of 42 vulnerabilities, we managed to retrieve the related patches for 31 of them. We first identified the causes of the vulnerabilities and introduced a taxonomy (V-A). Then, we report the information regarding the nature of the vulnerable components (V-B). Finally, in Section V-C, we detail our findings related to the changes necessary to patch Android vulnerabilities.

### A. RQ1: Categorizing the Causes of Android Vulnerabilities

Our analysis revealed a total of 43 different issues, one or two per vulnerability report. They were processed according to the methodology presented in Section IV-B to build the mind map. This taxonomy is presented in Figure 2. The mind map is composed of three main nodes *Design*, *Code* and *Test*. They refer to the moment of the probable phase of development when the issue occurred. The second level of nodes corresponds to the cause of the issue. We identified 8 nodes, i.e., Resource Management, Data, Semantic, Initialization Bug, Forget to remove debug features, Flow, Unauthorized Access, and Insecure Protocol. The third level, which appears only in the Code part, allows a better categorization of the issues. The third level is composed of 9 nodes. i.e., Buffer overflow, Incorrect pointer dereference, Stack consumption, Input not verified, Serialization issue, Unprotected use of a function, Missing / Incorrect implementation of a feature, Object not rightly created, and Wrong initialization of data.

Table I shows the frequency of the categorized issues extracted from the mind map. We can clearly see that most of the issues originate from the coding part (about 70%). This is not very surprising as other studies on faults and failures, e.g., Hamill et al. [21], showed that failures are heavily associated with coding faults. However, the number of issues related to design is surprisingly high (about 28 %). This might be attributed to the number of issues related to permission handling. A deeper analysis shows that a great number of issues are due to a missing or incorrect implementation of features. This kind of issues can occur when developers misunderstand or misinterpret requirements of the system. Another reason can be the oversight of taking into account a specific case.

TABLE I: Distribution of the 43 issues in the taxonomy.

| Origin | Kind | Number | Total (%) |
|---|---|---|---|
| Design | Flow | 4 | 12 (27.90%) |
| | Insecure protocol | 1 | |
| | Unauthorized access | 7 | |
| Code | *Resource Management* | | 30 (69.77%) |
| | Buffer overflow | 4 | |
| | Incorrect pointer dereference | 1 | |
| | Stack consumption | 1 | |
| | *Data* | | |
| | Input not verified | 7 | |
| | Serialization issue | 1 | |
| | *Semantic* | | |
| | Unprotected use of a function | 3 | |
| | Missing/incorrect implem. of a feature | 11 | |
| | *Initialization* | | |
| | Object not rightly created | 1 | |
| | Wrong initialization of data | 1 | |
| Test | Forgot to remove debug feature | 1 | 1 (2.32%) |

Our taxonomy allows us to distinguish 13 kinds of issues that cause Android vulnerabilities. We believe that those categories can be helpful during a code review to determine what to look for. However, this new taxonomy raises the question of what are the actual differences between the vulnerability causes and their consequences? In other words, we want to check whether the categories of causes, i.e., as identified by us, differ from the categories of the consequences, i.e., as reported by the Common Weakness Enumeration (CWE [7]). To answer this question we compare our categories with the CWE one. In fact, only two of our categories are overlapping with those of CWE: Input validation (CWE category) with input not verified (our taxonomy) and Buffer errors (CWE category) with buffer overflow (our taxonomy).

Except these two categories, no other category overlaps. Yet, this overlapping seems logical as in this case the CWE categories of vulnerabilities are in fact describing the origin of the vulnerabilities.

From the above-mentioned results, it becomes clear that our *issue taxonomy* provides quite different information than the categorization of CWE. This information could be used as check list for developers before committing to the repositories. Additionally, it is noted that our taxonomy is complete with respect to our data and that it can be extended based on additional data for future investigations. This is actually the advantage of using mind maps, i.e., it is incremental and thus, it remains open for future evolution.

### B. RQ2: Vulnerable Components

Our second research question is about the vulnerable components. This piece of information is indeed of interest for prioritizing purposes.

*1) Role of the vulnerable component:* After analyzing all the vulnerabilities, we distinguished 9 kinds of top level components, i.e, Driver, Library, Messaging, Networking, Access Control, Browsing, Cryptography, Dalvik and Debug. The result of this analysis are presented in Fig. 3.

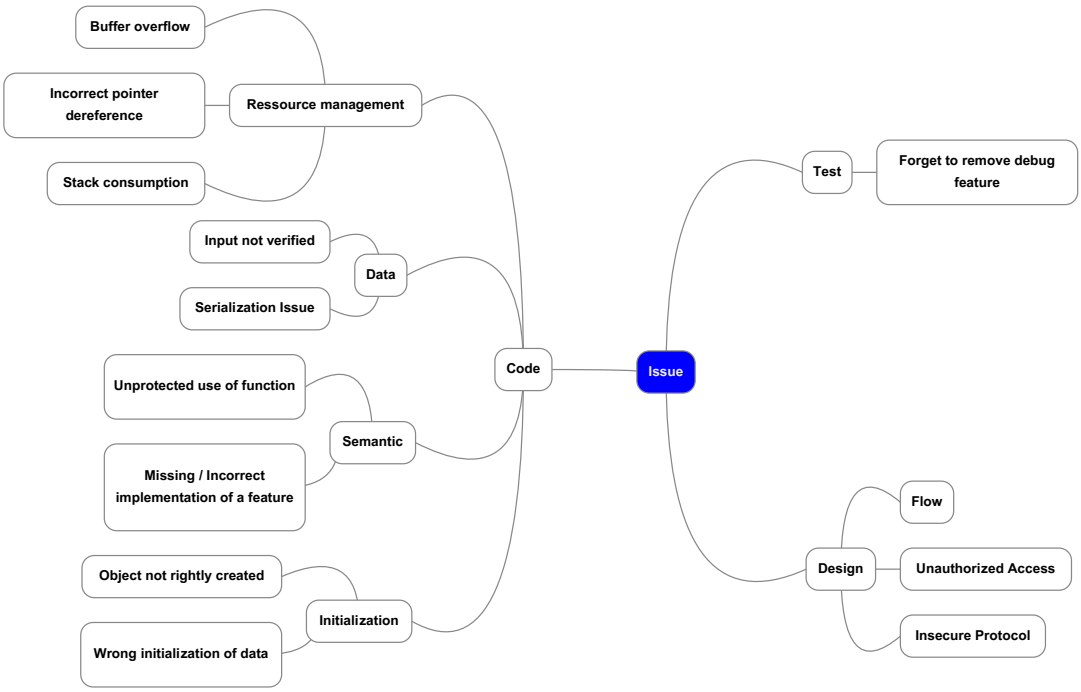



Fig. 2: The issue taxonomy: mind map of issues related to Android vulnerabilities.

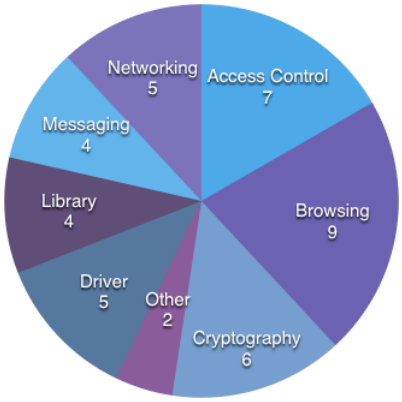



Fig. 3: Vulnerable components.

We observed that 9 of the vulnerabilities were originating from components related to web browsing. The components in charge of the control of access were responsible for 7 of them. This was expected as we found a lot of issues related to the handling of permission.

Another important kind of component was the cryptography related ones, with 6 vulnerabilities. One interesting fact about those vulnerabilities is that they are directly linked to a lack of understanding on how cryptography works and how it should be implemented.

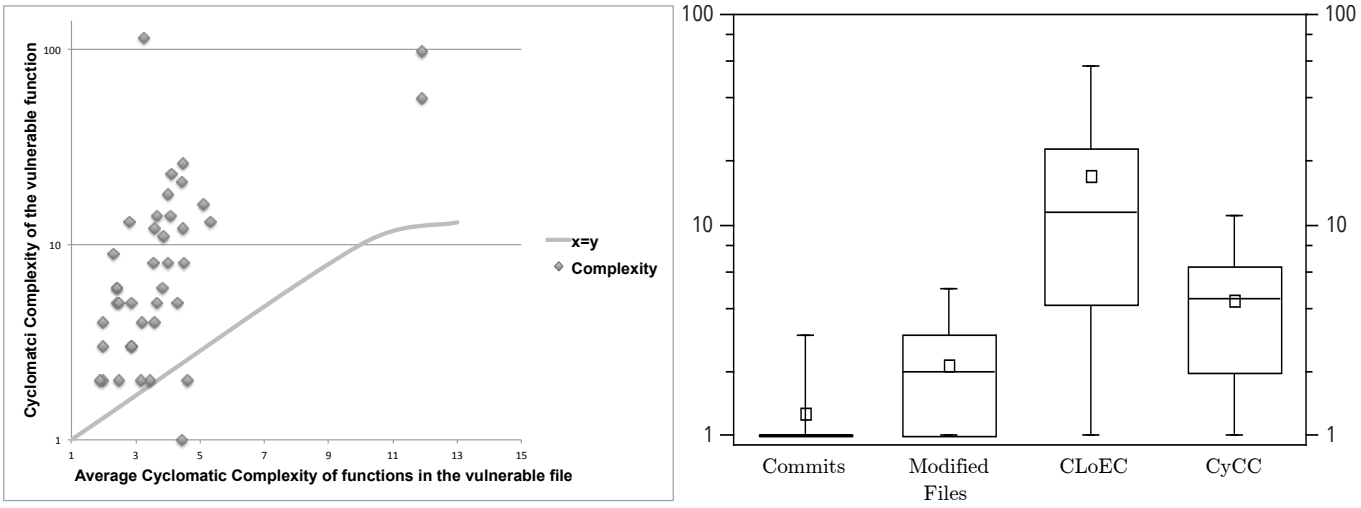
The remaining categories were Driver, Network, Library, Messaging, Dalvik and Debug with respectively 5, 5, 4, 4, 1 and 1 vulnerabilities. We noticed that the 3 most represented ones account for more than 50% of all the vulnerabilities. These components can have a major impact on the system, i.e., to gain control or leak some sensitive information.

Thus focusing on repositories and components that have this kind of role will reduce the amount of code to review

*2) Complexity of the vulnerable components:* In the second part of our analysis, we computed the cyclomatic complexity of all vulnerable functions and compared it to the average complexity of all the other functions that were present in the same file. In total, we studied 40 different vulnerable functions. They correspond to the code that we retrieved and was written in Java, C or C++. Indeed, it is not possible to compute the complexity for vulnerabilities caused by errors within XML or RC files.

Figure 4a presents our results on a logarithmic scale. The horizontal axis corresponds to the average complexity and the vertical axis corresponds to the complexity of the vulnerable function. The gray line represents the $y = x$ function. Thus, points above this line indicates a complexity of the vulnerable function higher than the non-vulnerable ones.

From these data, we observed that almost all vulnerable functions had a higher cyclomatic complexity than the rest of the functions, on average, located in the same file. In fact, only two cases are less complex. However, these represent functions that needed to be changed in combination with others which are indeed complex. This fact confirms that vulnerabilities tend to appear in functions that have a higher complexity than the average and can be used together with the previous result as a way to warn the developer when they are about to modify complex part of a likely to be vulnerable component.




(a) Complexity of vulnerabilities.

(b) Complexity of the changes made.

Fig. 4: Complexity of the vulnerable components and of the changes made to fix the vulnerabilities.

### C. RQ3: Fix Analysis

The last part of our study of Android vulnerability is about the changes required to remove a vulnerability. These are described in terms of change types and complexity.

*1) Kind of Change:* Table II shows the changes that were required and their distribution. Among the patches that we were able to retrieve, 60% of them consisted in the addition of one or more lines of codes, 23% were modifications of existing codes and 17% were about removing code.

One of the first things that we notice is that almost one third of those changes are in fact additions of one or more conditions, meaning that the developer's mistake was mostly failing to adequately check something in the code. Removing actions are frequently used to remove some "resources" that are not needed. The modification changes were mostly simple ones like a call to a method, changing the arguments that were passed to it in order to avoid triggering a vulnerability. In only one case the issue was so complicated that it required a complete rewrite. This means that vulnerability fixing doesn't imply in most of the case large and deep refactoring of the code and can be done within the vulnerable function.

*2) Complexity of the Changes:* In the previous subsection, we saw that the vulnerability fix could be done in most of the cases within the vulnerable function. However, this doesn't mean that the fix actions are easy. Thus in this part of our analysis, we look at the complexity of the changes required to remove the vulnerabilities. However quantifying the complexity of changes is quite hard. To deal with this problem, we choose to rely on four metrics that we believe allow a good understanding of the complexity of the changes.

The box-plots of Figure 4b show the complexity results. We observe that in most of the cases only one commit is required to patch an Android vulnerability. In average, it involves the modification of 2 files and 17 lines of codes. The average changed complexity according to the CyCC metric is 4. This indicates that four linearly independent paths had to be changed. This can be considered as very complex according to the work of Böhme and al. [19]. Similarly the complexity according to CLoEC is high, i.e., 17 lines of instructions were in average modified (add/delete/modify) to patch a vulnerability. Thus if Android vulnerabilities can be solved at the function level, the efforts required to fix this function are quite complex. We observed that the CyCC was interesting for understanding the complexity, but not sufficient to be used alone. Indeed sometimes vulnerabilities require instantiation and modification that is not measured by this metric. However, we believe that its combination with the other metrics represents the required information.

## VI. Discussion

This section summarizes the findings of our study and discusses its limitations. Specifically, Section VI-A provides a set of guidelines that may enable future research and meta-analysis of vulnerabilities. Sections VI-B and VI-C discuss our actionable findings and threats to validity, respectively.

### A. Guidelines for CVE-NVD

During our study, we found that the current form of the NVD database is satisfactory for reporting problems and keeping some statistics but is not well suited for mining information.

In order to provide a framework that enables further research on the mining and analysis of vulnerabilities, we suggest the following changes. First, the inclusion of the relevant patch file (if existing). This will allow researchers and practitioners to better understand the error that led to vulnerability. Second, the addition of a categorization of the vulnerability issues. The combination of different categorization like the root cause and the consequence will allow a better comprehension of the vulnerabilities. Finally, the integration of a link checker to avoid having dead links.

TABLE II: Distribution of the code changes needed to fix vulnerabilities.

| Type | Kind | Description | Number | Total |
|---|---|---|---|---|
| ADD | Condition(s) | If then else | 19 | 36 (60.0%) |
| | Authorization | Permission in android manifest | 2 | |
| | Function + Use of it | Creation of a new function | 5 | |
| | Class + Use of It | Creation of a new class | 2 | |
| | Exception raise | Try catch or throw | 3 | |
| | Define or initialization | Variable | 5 | |
| REMOVE | Condition(s) | If then else | 2 | 10 (16.7%) |
| | Use to a function | If the function was vulnerable | 5 | |
| | File | Deprecated and vulnerable files | 2 | |
| | Authorization | Permission in the Android manifest | 1 | |
| MODIFY | Call to a method | Changing arguments | 8 | 14 (23.3%) |
| | Condition | Modifying expression | 1 | |
| | Function | Rewrite of a function | 4 | |
| | Complete rewrite | Rewrite of the all component | 1 | |

### B. Android Vulnerability Profile

The work presented in this paper can be summarized by the following Android vulnerability profile. Android vulnerabilities are mostly located in components responsible for browsing, Cryptography, Access Control or Networking. The vulnerable parts of these components are among the most complex ones. The vulnerabilities are mostly coming from the coding and among all vulnerabilities one out of four (i.e., 25%) are due to a missing or an incorrect implementation of a feature. A lot of them are also emanating from mishandling authorization or input. Fixing vulnerabilities is complex and requires changes on code parts located on (in average) 4 linearly independent path accounting for an average of 17 lines of code. However, these fix actions can be done directly within the vulnerable function.

### C. Threats to Validity

A first threat to validity is that this study was based on manual analysis. Thus, we cannot ensure that our results are fully accurate. For instance, a misinterpretation of the nature of a patch or a wrong categorization of an issue are possible mistakes that we might have made. We reduced this threat by reproducing the results with different analysis, for three times. Additionally, we made all of our data publicly available, thus, enabling replication and independent validation of our results.

Another threat to the validity lies in our use of the CyCC metrics. We use this metric as it has been used by the literature for the study of regression bugs [19]. However, it might not be appropriate for describing vulnerability complexity. To reduce this threat, we included other metrics like CLoEC.

Finally, the biggest threat to validity is the fact that we rely on the NVD-CVE set of Android vulnerabilities which have really few links to patches. We thus cannot be certain that the results would be the same with vulnerabilities for which patch are not available.

## VII. RELATED WORK

The security of the Android system has been extensively reviewed in the literature [22]–[24]. Researchers have found a number of security issues both at the operating system level and at the application level. In an effort to address those issues, Enck et al. have developed a tool for testing the security enforcement at the levels of both the system and the inter-component communication [22]. In [24] the authors have also defined a set of analysis rules that can be used to statically detect vulnerabilities in Android applications.

Privacy leaks are the most common security concerns in Android. With AsDroid [25] and AppIntent [26], researchers have built tools for statically detecting such leaks. Leaks, however, can be benign, by being designed as part of the application features, or malicious. The proposed approaches thus attempt to differentiate between them in order to report the potentially malicious leaks.

Regarding software vulnerability, a substantial part of the literature is focused on the direction of vulnerability prediction. The objective of related work in this direction is mainly to predict whether a component is likely to contain a vulnerability. Morrison et al. [27] underlines the barriers of applying vulnerability prediction models. Meneely et al. [28] and Shin et al. [13] have proposed a prediction approach using code churn, i.e., code tokens that are added, modified or deleted, in combination with both code complexity and developer activity, to identify vulnerable files. Experiments with data from Mozilla Firefox and the Linux kernel have revealed that their approach can correctly identify 80% of vulnerable files, however, with 25% false positives. Scandariato et al. [2] have used text mining metrics to successfully predict vulnerable components. In another recent study, Walden et al. [1] have presented a comparative study on the performance of software metrics and text mining for predicting vulnerabilities.

Meneely et al. [29], [30] leverage social networking metrics to predict vulnerable files. They found that, in the Linux kernel and Apache, files modified by more than nine contributors or by a new one are likely to be vulnerable. Neuhaus et al. [31] experimented on Mozilla and found that when components had only one vulnerability in the past, they were unlikely to have others in the future. Gegick et al. [32] developed a regression tree model method which combines metrics like lines of code, code churn and warnings generated by static analysis tools and all vulnerable components with an 8% false positive rate.

Walden et al. [33] focused on PHP applications. They considered three complexity metrics and five security indicators, and measured their correlation with vulnerabilities. Their results vary significantly between the studied projects, indicating that correlation between complexity and vulnerabilities might not be generalizable.

All these studies use code or repository attributes to predict vulnerable files. In contrast, the present paper aims at studying directly what code changes were made to fix known and likely severe vulnerabilities. Additionally, our focus is specific to the Android system for which there is scarce work in the literature of vulnerability prediction. Perhaps the most relevant work to ours is by Bosu et al. [34] who studied the characteristics of vulnerable code changes. However, our study differs from it in the following three ways: a) we study changes that fix a vulnerability rather than ones that introduce it, b) we study known, severe and exploitable vulnerabilities rather than those found by code reviews and c) we aim at categorizing and describing the origin of the Android vulnerabilities and not generic changes that might indicate the introduction of one.

Other closely related work is due to Morrison et al. [35] and Fonseca et al. [36]. These studies explore vulnerabilities and analyze their patches. However, both of them focus on specific vulnerabilities that are not related to Android.

## VIII. Conclusion

This paper presented an analysis of the issues, components and patches related to Android vulnerabilities. Our study was performed on the Android vulnerabilities that were reported in the CVE-NVD database. We identified the involved issues and established their taxonomy.

Our analysis led to the following findings: 1) vulnerable Android components fall within 9 different top-level categories, 2) vulnerabilities tend to appear in the most complex functions, 3) there are 13 different types of issues leading to those vulnerabilities and 4) they require complex changes in order to be removed. Finally, we provided some guidelines to improve CVEs.

To enable further research and reproducibility of our results, the vulnerabilities along with their detailed analysis and manually collected patches are publicly available at http://www.jimenez.lu/Research/.

## References


[1] J. Walden, J. Stuckman, and R. Scandariato, "Predicting vulnerable components: Software metrics vs text mining," in *ISSRE'14*, pp. 23–33.

[2] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *TSE*, vol. 40, no. 10, pp. 993–1006, Oct 2014.

[3] M. Gegick, P. Rotella, and L. A. Williams, "Predicting attack-prone components," in *ICST'09*, pp. 181–190.

[4] Android dalvik and art. [Online]. Available: https://source.android.com/devices/tech/dalvik/

[5] Android source code on github. [Online]. Available: https://github.com/android

[6] Android source code on googlesource. [Online]. Available: https://android.googlesource.com

[7] Cwe presentation page. [Online]. Available: http://cwe.mitre.org

[8] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1665–1705, 2014.

[9] Definition of vulnerability. [Online]. Available: https://cve.mitre.org/about/terminology.html

[10] Rfc 2616: Http/1.1 method definitions. [Online]. Available: http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

[11] Estimation of the number of pc sales. [Online]. Available: http://goo.gl/ytqXQ3

[12] G. McGraw, *Software Security: Building Security In*. Addison-Wesley Professional, 2006.

[13] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *TSE*, vol. 37, no. 6, pp. 772–787, 2011.

[14] Top levels of boris beizer's bug taxonomy. [Online]. Available: http://c2.com/cgi/wiki?SourcesOfBugs

[15] A software bug taxonomy. [Online]. Available: http://www.cs.nyu.edu/~lharris/content/bugtaxonomy.html

[16] C. K. Giri Vijayaraghavan, "Bug taxonomies: Use them to generate better tests," *Star East*, pp. 1–40, 2003. [Online]. Available: http://www.testingeducation.org/articles/bug_taxonomies_use_them_to_generate_better_tests_star_east_2003_paper.pdf

[17] I. Chowdhury and M. Zulkernine, "Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities?" in *SAC '10*, 2010, pp. 1963–1969.

[18] T. J. McCabe, "A complexity measure," in *ICSE '76*, pp. 407–.

[19] M. Böhme and A. Roychoudhury, "Corebench: Studying complexity of regression errors," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA'14, 2014, pp. 105–115.

[20] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury, "Regression tests to expose change interaction errors," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, 2013, pp. 334–344.

[21] M. Hamill and K. Goseva-Popstojanova, "Exploring fault types, detection activities, and failure severity in an evolving safety-critical software system," *Software Quality Journal*, vol. 23, no. 2, pp. 229–265, 2015.

[22] W. Enck, M. Ongtang, and P. D. McDaniel, "Understanding android security," *IEEE Security & Privacy*, vol. 7, no. 1, pp. 50–57, 2009.

[23] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer, "Google android: A comprehensive security assessment," *IEEE Security & Privacy*, vol. 8, no. 2, pp. 35–44, 2010.

[24] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *USENIX Security'11*, 2011.

[25] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "Asdroid: detecting stealthy behaviors in android applications by user interface and program behavior contradiction," in *ICSE*.

[26] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "Appintent: analyzing sensitive data transmission in android for privacy leakage detection," in *CCS'13*, 2013, pp. 1043–1054.

[27] P. Morrison, K. Herzig, B. Murphy, and L. Williams, "Challenges with applying vulnerability prediction models," in *HotSoS'15*. ACM–Association for Computing Machinery.

[28] A. Meneely and L. A. Williams, "Strengthening the empirical analysis of the relationship between linus' law and software security," in *ESEM'10*, 2010.

[29] A. Meneely and L. Williams, "Socio-technical developer networks: should we trust our measurements?" in *ICSE'11*, 2011, pp. 281–290.

[30] A. Meneely, L. Williams, W. Snipes, and J. A. Osborne, "Predicting failures with developer networks and social network analysis," in *FSE'08*, 2008, pp. 13–23.

[31] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *CCS'07*, 2007, pp. 529–540.

[32] M. Gegick, L. Williams, J. A. Osborne, and M. A. Vouk, "Prioritizing software security fortification throughcode-level metrics," in *QoP'08*, 2008, pp. 31–38.

[33] J. Walden, M. Doyle, G. A. Welch, and M. Whelan, "Security of open source web applications," in *ESEM'09*, 2009, pp. 545–553.

[34] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni, "Identifying the characteristics of vulnerable code changes: an empirical study," in *FSE'14*, 2014, pp. 257–268.

[35] A. Milenkoski, B. Payne, N. Antunes, M. Vieira, and S. Kounev, "Experience report: An analysis of hypercall handler vulnerabilities," in *ISSRE'14*, pp. 100–111.

[36] J. Fonseca, N. Seixas, M. Vieira, and H. Madeira, "Analysis of field data on web security vulnerabilities," *Dependable and Secure Computing, IEEE Transactions on*, vol. 11, no. 2, pp. 89–100, March 2014.