Mining Mobile App Markets for Prioritization of Security Assessment Effort

Alireza Sadeghi
Department of Informatics
University of California, Irvine, USA
alirezs1@uci.edu

Naeem Esfahani
Google Inc.,
Mountain View, USA
naeeem@google.com

Sam Malek
Department of Informatics
University of California, Irvine, USA
malek@uci.edu

ABSTRACT

Like any other software engineering activity, assessing the security of a software system entails prioritizing the resources and minimizing the risks. Techniques ranging from the manual inspection to automated static and dynamic analyses are commonly employed to identify security vulnerabilities prior to the release of the software. However, none of these techniques is perfect, as static analysis is prone to producing lots of false positives and negatives, while dynamic analysis and manual inspection are unwieldy, both in terms of required time and cost. This research aims to improve these techniques by mining relevant information from vulnerabilities found in the app markets. The approach relies on the fact that many modern software systems, in particular mobile software, are developed using rich *application development frameworks (ADF)*, allowing us to raise the level of abstraction for detecting vulnerabilities and thereby making it possible to classify the types of vulnerabilities that are encountered in a given category of application. By coupling this type of information with severity of the vulnerabilities, we are able to improve the efficiency of static and dynamic analyses, and target the manual effort on the riskiest vulnerabilities.

CCS CONCEPTS

• **Security and privacy → Mobile platform security**; *Access control*; • **Software and its engineering → Software testing and debugging**; *Automated static analysis*;

KEYWORDS

Security Vulnerability, Mining App Market, Software Analysis

ACM Reference format:
Alireza Sadeghi, Naeem Esfahani, and Sam Malek. 2017. Mining Mobile App Markets for Prioritization of Security Assessment Effort. In *Proceedings of 2nd International Workshop on App Market Analytics, Paderborn, Germany, September 5, 2017 (WAMA'17),* 7 pages.
https://doi.org/10.1145/3121264.3121265

1 INTRODUCTION

One of the most exciting developments in the delivery and deployment of software has been the emergence of *app markets*. By providing a medium for reaching a large consumer market at a nominal cost, app markets have leveled the software industry, allowing small entrepreneurs to compete head-to-head against prominent software development companies. The result has been a highly vibrant ecosystem of mobile application software, but the paradigm shift has also given rise to a whole host of security issues [21].

In light of the increasing security threats, the state-of-the-practice needs to move away from the reactive model of patching the security vulnerabilities to proactive model of catching them prior to product release [14]. One approach is to manually inspect the security of application software prior to its release, which is an expensive and error-prone process. Alternatively, as a step toward addressing the above issues, static analysis is gaining popularity for automatically finding security problems in application software [15]. Although this technique is more cost effective compared to manual inspection, it has its own shortcomings. Static analysis usually generates an unordered wide-ranging list of potential vulnerabilities, varying from hazardous easily-exploitable vulnerabilities to uncommon and/or non-severe risks and even false positives, i.e., detected vulnerabilities that are not really exploitable. Thus, static analysis has not been able to completely remove the need for manual inspection, and instead has helped target such effort to seemingly problematic parts of the code.

An approach to eliminate the false positives from a static analyzer is to employ dynamic analysis techniques. For instance, find test cases that execute the suspected parts of the code flagged by the static analyzer to make an unequivocal determination as to the dangers they pose [4]. However, this approach has its own technical challenges. In particular, as the software under test gets larger and more complicated, list of potential exploits becomes longer and hence, checking all possible exploits in a reasonable time becomes more difficult. As a result, some prioritization mechanisms are necessary to focus the manual inspection and dynamic analysis effort on the vulnerabilities that are likely to materialize in the form of dangerous exploits.

Categorization of apps on the app markets presents us with a unique opportunity to tackle these issues. App categories, such as *game*, *travel*, *communication*, are intended to help the users with finding relevant applications. Some examples of app markets with categories are F-Droid for open source and Google Play for Android applications. Other than facilitating the users in searching and browsing, categories of apps have shown to be good predictors of the common features found within software of a particular category [12, 17].

In this paper, we explore the utility of app markets in informing the security inspection and analysis of software applications. The fact that the majority of apps provisioned on such markets is built using a common *application development framework (ADF)* presents us with an additional opportunity. Since many of the security issues encountered in modern software are due to the wrong usage of ADF [21], we are able to develop highly effective predictors as to the types of security vulnerabilities one may find in different

WAMA'17, September 5, 2017, Paderborn, Germany

ACM ISBN 978-1-4503-5158-4/17/09…$15.00
https://doi.org/10.1145/3121264.3121265

categories of apps. In our prior research [17], we have shown the existence of strong correlations between the types of vulnerabilities and categories of apps. This paper shows how such correlations can be used to improve the efficiency of static, dynamic, and manual analysis techniques for security assessment of software.

Our research result is useful for prioritizing the order in which vulnerabilities flagged by static analyzer are manually inspected or dynamically analyzed. We use a combination of the frequency with which vulnerabilities occur in a given category of apps and the severity of such vulnerabilities to prioritize the effort that follows the static analysis phase in vetting the vulnerabilities. Thus, given a limited amount of time and resources, the approach enables the maximization of the likelihood with which the most severe exploits are detected and corroborated prior to system release.

Our experimental results for Android apps that make extensive use of a particular ADF have been very positive. Our approach improved the efficiency of static analysis of Android apps by 68%, while keeping the vulnerability detection rate at 100%. Moreover, our suggested ranking mechanism, which prioritizes vulnerabilities based on risk, could inform the manual inspection of vulnerabilities. For instance, our experiments show that when due to time limitations the security inspector is forced to only investigate half of the potential security threats, our ranking system suggests vulnerabilities that constitute 80% of total risk, while uniformed inspection would have covered only 50% of total risk.

The remainder of this paper is organized as follows. Section 2 provides examples of Android vulnerabilities and possible techniques for detecting them. Section 3 outlines the overview of our approach, while Sections 4 to 6 describe the details. Sections 7 and 8 describe the research experimental setup, results, and analysis. Finally, the paper concludes with a discussion of related research and our future work.

2 MOTIVATION

To motivate the research and illustrate our approach, we provide examples of two vulnerability patterns having to do with Inter-Process Communication (IPC) among Android apps. Android provides a flexible model of IPC mechanism using application-level message known as Intent.

The first vulnerability occurs on line 5 of Figure 1. Without checking to ensure *areaCode* is not *null*, *equals* method is used to compare its value, resulting in a null pointer exception when *areaCode* is *null*. A malware can make this app crash by intentionally not including the area code payload. This is a null dereference vulnerability [6, 7] that an adversary could discover by simply reverse engineering any of the apps on the market, and exploiting it to launch a denial of service attack.

```
1 public class MessageSender extends Service {
2   public void onStartCommand (Intent intent, int flags, int startId) {
3     String areaCode = intent.getStringExtra ("AREA_CODE");
4     String number = intent.getStringExtra ("PHONE_NUM");
5     if (areaCode.equals("911") || areaCode.equals ("703")) //null pointer
          dereference vulnerability
6       //if (hasPermission())
7       SmsManager.getDefault().sendTextMessage(number, null, msg, null, null);
8   }
9   boolean hasPermission () {
10    if(checkCallingPermission(SEMD_SMS) == PERMISSION_GRANTED)
11      return true;
12    return false;
13  }
14 }
```

Figure 1: Vulnerable app: receives an Intent and sends a text message to the provided number. Bold statements represent potentially exploitable vulnerabilities.

The second, perhaps more interesting, vulnerability occurs on lines 7 of Figure 1, where *MessageSender* service sends a text message. This is a sensitive Android action that requires a special access permission to the system's SMS service. Although *MessageSender* has that permission, it also needs to ensure that the sender of the original Intent message has the required permission to use the SMS service. An example of such a check is shown in *hasPermission* method of Figure 1, but in this particular example it does not get called (line 6 is commented) to illustrate the vulnerability. If another, potentially malware, app does not have the permission to send text message, it is able to make this app perform that action on its behalf. This is a privilege escalation vulnerability and has been shown to be quite common in the apps on the market [6].

The above two vulnerabilities are just examples we have selected from a large collection of well-known Android vulnerabilities [6, 7]. A common technique to detect such vulnerabilities is to employ static analysis tools. An example of a *rule* (i.e., a vulnerable code pattern) a static analysis tool may be configured, to check for in the source code, is operations performed on objects that may be *null* (similar to the one shown on line 5 of Figure 1).

An important shortcoming of most static analysis tools is that they produce a high-rate of false positives, i.e., alarms for unexploitable vulnerabilities. For instance, with inclusion of Line 6 in the code of Figure 1 the privilege escalation vulnerability on line 7 is not exploitable. But many static analysis tools, particularly branch-insensitive analyses, would still flag line 7 as a vulnerability. Finding the root cause of an error and determining whether it is truly a vulnerability is an extremely time consuming and unwieldy task. Additionally, many practitioners often avoid further investigation of reports generated by static analysis tools, as determining real vulnerabilities requires significant security expertise.

Our research aims to address the aforementioned challenges by eliminating the unnecessary checks and prioritizing the efforts involved in static, dynamic, and manual analysis of the software.

3 APPROACH OVERVIEW

Figure 2 depicts an overview of our approach. Our approach takes *App Market* as an input, where each application is labeled with a predefined class or category. Even if the apps are not categorized by the input repository, a machine learning techniques could be used to cluster similar apps within the same category. The other input

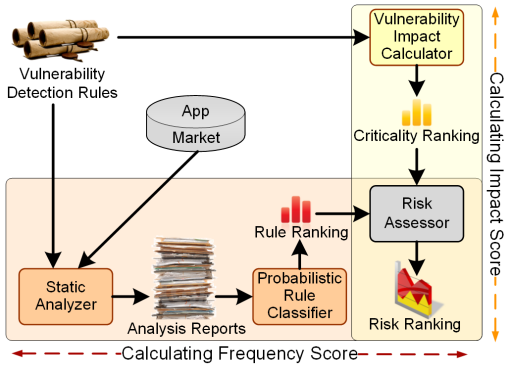

Figure 2: Overview of the approach.

to the framework is *Vulnerability Detection Rules* (VDRs), which is a list of known vulnerability patterns in Android.

For each application on the repository, *Static Analyzer* component inspects the code for patterns that match VDRs. The output of this analysis is an *Analysis Report*, which consists of all the vulnerability patterns detected in the code along with the corresponding locations. The generated list of latent vulnerabilities for a categorized repository of applications serves as our training data set. Given this data, the *Probabilistic Rule Classifier* component ranks each vulnerability pattern based on its frequency of occurrence in the *Analysis Report*. To that end, *Probabilistic Rule Classifier* applies conditional probability technique to find the likelihood of occurrence of each vulnerability pattern on each category to produce *Rule Ranking*. The higher is the position of a vulnerability pattern in the ranking of a category (higher frequency score), it is more likely for its corresponding VDR to detect that vulnerability in the given category.

The other thrust of our approach is the *Vulnerability Impact Calculator*, which ranks each VDR based on its threats to various security aspects of the system (i.e, Confidentiality, Integrity, and Availability). The result is *Criticality Ranking*, in which an impact score is assigned to each *Vulnerability Detection Rule* based on the severity of damage caused by exploitation of the corresponding vulnerabilities, regardless of the application categories. Thereby, the higher impact score implies the severity of consequences an exploit causes.

Finally, *Risk Assessor* combines the two previous scores (i.e., frequency score and impact score) and provides a new two-dimensional score. The *Risk Ranking* provides an assessment of the VDRs in each software category based on the severity and likelihood of vulnerabilities it can detect.

Our proposed approach enhances the state of practice in static, dynamic, and manual analysis of the app's security as follows.

- *Rule Ranking* information is used to determine the set of relevant VDRs for a given category of application, thus increasing the speed at which static analyzer can execute and produce feedback.
- *Criticality Ranking* information is used to determine which vulnerabilities detected by static analysis should be dynamically analyzed first.
- *Risk Ranking* information is used by a security analyst to determine the risks that are likely to be posed by an application of a given category, thus informing the manual inspection process

4 PROBABILISTIC RULE CLASSIFICATION

As depicted in Figure 2, the output of *Static Analyzer* component is the *Analysis Report*. This report contains the app's source code locations that match the predefined vulnerability patterns specified in *Vulnerability Detection Rules*. *Static Analyzer* tries all the rules and finds all matches in the source code. However, some of the rules may not match at all. We depict the set of all *Vulnerability Detection Rules* as R and the set of rules where at least one match has been found for them as M. If we know these rules upfront, we can improve the efficiency of *Static Analyzer* by removing the irrelevant rules (i.e., $\overline{M} = R - M$). We refer to this task as *rule reduction*.

This definition can be further extended by considering the categorical information as follows: M_c is the set of rules that are matched to at least one vulnerability in the apps with category c. The intuition behind this formulation is that applications categorized in the same class share features implemented by similar source code patterns and API usage pattern [8]. Thereby, it is more likely for a set of applications in the same category $c \in C$ (where C is the set of all categories) to have common vulnerabilities.

It takes only one false positive to include the corresponding rule r in M_c. As the number of projects in the category and the number of files in the projects increases, it becomes more likely for all the rules to be included in M_c due to false positives, hence M_c converges to R. In other words, for each rule some kind of matching (which may be a false positive) is found. This is the problem with simply checking the membership of rule r in M_c as the binary measure of relevance of rule r to category c. We need a measure that expresses the likelihood of rule r being relevant to a given category c. This is the classical definition of conditional probability of $P(r|c)$. Calculating this value helps us to confine the static analysis rules for each application category to the rules that detect widespread vulnerabilities in that category.

By applying *Bayes Theorem* [5] to the *Analysis Reports* (recall Figure 2), we can calculate $P(r|c)$, indicating the probability of a given rule matching an application from a category:

$$P(r|c) = \frac{P(c|r) \times P(r)}{P(c)} \tag{1}$$

Here, $P(c)$ is the probability of an application belonging to a category c, calculated via dividing the number of applications belonging to category c by the total number of applications under study. $P(r)$ is the probability of a rule r matching, calculated via dividing the number of matches for rule r by the total number of matches for all rules on all applications. Finally, $P(c|r)$ is the probability that a given application category c have the rule r matching, calculated via dividing the total number of times applications of category c were matched with rule r by the total number of matches for applications of that category.

As we described earlier $P(r|c)$ is used by the Rule Selector to reduce the number of rules used by a static analyzer. We can exclude a rule r from the static analysis of an application belonging to category c, when $P(r|c) \leq \epsilon$, where ϵ is a user-defined threshold indicating the desired level of rule reduction. We depict the set of excluded rules for category c as E_c, and in turn, assess the reduction in the number of rules for category c as following:

$$Reduction_c = (|E_c|/|R|) \times 100 \tag{2}$$

The value selected for the threshold presents a trade-off between the reduction of rules (i.e., the improvement in efficiency) and the coverage of static analysis. As more rules are removed, the static analyzer executes faster, but the coverage decreases, increasing the chances of missing a vulnerability in the code. We will discuss the selection of threshold in Section 8.

5 VULNERABILITY IMPACT CALCULATION

An issue with most static analyses is that many of the flagged vulnerabilities are false positives. By definition it is impossible to find an input that can exploit a given false positive. *Exploit generation* [4] is the process of determining whether a vulnerability is exploitable or not. *Triaging* [23] is also an alternative approach to eliminate false positive by auditing source code manually. However, often the large number of flagged vulnerabilities combined with limited available resources prevent applying exploit generation or triaging

on every vulnerability. In our experiments, we found up to 37 vulnerabilities in a single Android application. Clearly, a prioritization mechanism would be useful as it would allow the dynamic or manual analysis to focus on the most critical vulnerabilities first. This is where *Criticality Ranking* comes into play.

The criticality of a vulnerability is assessed based on two factors: *exploitability* and *influence*. Exploitability measures how much work is required or what resources are needed to exploit a vulnerability. Influence determines the amount of loss or the total cost of damage caused by a vulnerability exploitation. Common Vulnerability Scoring System (CVSS) [10], which is endorsed by the National Institute of Standards and Technology (NIST) for use on a wide range of systems, is a standard method that covers assessment of both factors.[1].

CVSS measures each factor with three metrics, which are called vectors. Exploitability vectors are *Access Vector*, *Access Complexity*, and *Authentication*. *Access Vector* indicates the location from which an attack is possible (e.g., network, adjacent network, local). *Access Complexity* measures the complexity of effort required to attack (e.g., high, medium, low). *Authentication* is calculated based on the number of attempts required for an attacker to accomplish an attack (e.g., multiple, single, none). Influence vectors measure the effects of exploited vulnerability on *Confidentiality*, *Integrity*, and *Availability*. *Confidentiality* measures prevention of information access to unauthorized users. *Integrity* indicates accuracy of data through its life cycle. *Availability* represents the ability to access the information when it is needed.

In addition to the above intrinsic *Base* scores, which are time and user-environments invariants, each vulnerability is correlated to some extrinsic metrics, namely *Temporal* and *Environmental* scores. Temporal scores address the properties of vulnerabilities that vary over time. Confirmation of the technical details of a vulnerability, the remediation status of the vulnerability, and the availability of exploit code are the metrics that measures temporal scores. On the other hand, the user environment and the organization's IT infrastructure have a great impact on the damage caused by an exploited vulnerability. Environmental scores assess this by considering collateral damage (including loss of life, physical assets, productivity or revenue) and percentage of all systems of the organization that could be affected by the vulnerability.

Based on the scores given to these components, CVSS calculator allocates a score in the scale of 0-10 to each vulnerability. A higher score means the vulnerability is more critical and its exploitation is more costly.

By this definition, *Criticality Ranking* is orthogonal to *Rule Ranking* defined in Section 4. The former expresses the actual cost of exploitation for a given vulnerability, while the latter expresses the potential for the presence of a given vulnerability in a certain category. In an effective exploit generation system, flagged vulnerabilities are exercised in their criticality order to find the more critical exploits before the less critical ones. In other words, the limited resources are invested on more critical vulnerabilities first.

6 RISK ASSESSMENT

As we discussed in Section 5, *Rule Ranking* and *Criticality Ranking* are orthogonal. In the previous sections, we described what is the meaning and use of each ranking. In this section, we describe how we use techniques from *quantitative risk assessment* [13] to combine these two rankings.

[1]In CVSS terminology, influence is referred to as "impact". We did not use the same term to avoid confusion with another concept defined in Section 6.

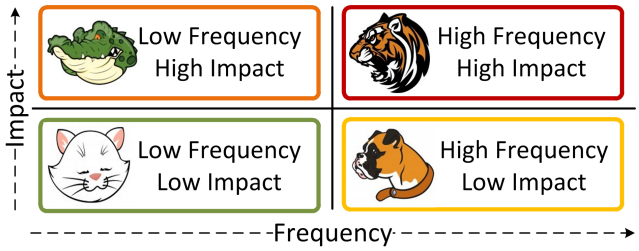

Figure 3: Tusler's risk classification scheme.

According to the quantitative risk assessment literature [13], *risk* has two dimensions: *Probability* and *Impact*. *Probability* represents the likelihood of the risk, while *Impact* represents the effect of the risk [13]. In our work, these dimensions are measured by the *Frequency Score* and *Impact Score*, which are values in the scale of 1-5.

In our research, vulnerability is considered as the risk to the security of the application. In Section 4 we defined *Rule Ranking* to be the likelihood of vulnerability in a given category. Therefore, *Rule Ranking* maps to *Probability* in that category. Similarly and by definition, *Criticality Ranking* maps to *Impact*. However, before using these rankings, we transform them into scores in the scale of 1-5, where the lowest and highest ranks obtain scores of 1 and 5, respectively.

As a result of considering vulnerability as risk, software security inspection turns into risk management. The security analyst tries to detect and mitigate the risks (i.e., vulnerabilities). Just like any other risk management task, security inspection is subject to time and budget limitations and a security analyst should focus on riskiest vulnerabilities first. Figure 3 depicts Tusler's risk classification scheme, which is originally proposed for project management [13]. We use this classification to classify the security vulnerabilities. Based on Tusler's classification, there are four risk classes for vulnerabilities:

- *Tiger*: Frequent and dangerous vulnerabilities, which have high *Probability* and high *Impact*.
- *Alligator*: Rare and dangerous vulnerabilities, which have low *Probability* and high *Impact*.
- *Puppy*: Frequent and harmful vulnerabilities, which have high *Probability* and low *Impact*.
- *Kitten*: Rare and harmful vulnerabilities, which have low *Probability* and low *Impact*.

The goal of a security analyst is to cover as many high risked vulnerabilities as possible, while considering time and budget limitations. It is clear that vulnerabilities categorized as *Tiger* and *Kitten* require highest and lowest attention, respectively. However, the ordering for *Alligator* and *Puppy* vulnerabilities depends on the security expert's preferences. We define *Risk Score* as a function of *Frequency Score*, S_p, and *Impact Score*, S_i, as follows:

$$RiskScore = (S_p{}^{w_p} \times S_i{}^{w_i})^{(\frac{1}{w_p+w_i})} \quad (3)$$

Where w_p and w_i are weights of frequency and impact scores, respectively. According to this equation, risk score is calculated as *geometric weighted average* of frequency and impact scores, which is

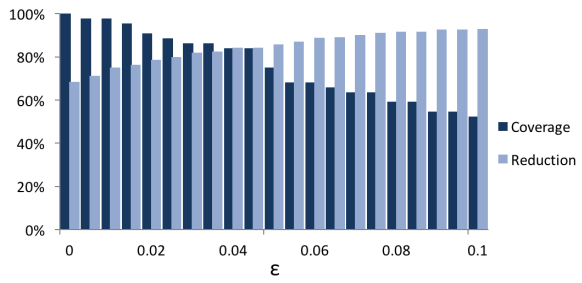

Figure 4: The overall reduction vs. the overall coverage of the remaining rules.

an extension of simple risk assessment technique, where the quantitative risk is obtained simply as product of frequency (probability) and impact factors [24]. These weights help us to customize the value of risk score based on importance of frequency and impact factors, and rank vulnerabilities based on the risk they pose. This results in *Risk Ranking*, which helps the security experts to focus on high risk vulnerabilities. In a sense, *Risk Ranking* is informed by the two dimensions that we discussed earlier: *Rule Ranking* and *Criticality Ranking*.

7 EXPERIMENT SETUP

The first step for using our approach is to populate a categorized repository (as a local *App Market*) and *Vulnerability Detection Rules*, which are depicted as the two inputs in Figure 2, with a set of application (denoted as set *App*) and a set of rules (recall R from Section 4) respectively. In this section, we describe how we collected the applications, *App*, and the set of rules, R, for our evaluation purposes and set up the experiments.

We considered Android apps with two characteristics in the evaluation process: categorized and open-source. The first characteristic is the basis of our hypothesis and almost all App repositories support it. The second characteristic is based on the requirements of some static analysis tools (e.g., Fortify) and manual inspection. Among the available repositories, F-Droid [1] supports both requirements. Hence, we collected 460 apps belonging to 11 categories from F-Droid.

We used *HP Fortify Static Code Analyzer* [2] as our static analysis tool (recall *Static Analyzer* from Figure 2). While Fortify provides a set of built-in rules for various programming languages, it also supports customized rules, which are composed by third-parties for specific purposes. For this research, we developed a set of custom rules for Android based on the rules provided by Enck et al. [7].

8 EVALUATION

The ultimate contribution of our research is to improve the efficiency of various software security analyses. Therefore, evaluation of our approach entails measuring the efficiency improvement from using the suggested rankings.

8.1 Rule Ranking

As you may recall from Section 4, the value of ϵ presents a trade-off between the reduction of rules and the coverage of static analysis. If ϵ is too low, reduction, and in turn, improvement in efficiency would be insignificant. On the other hand, if ϵ is too high, the chance of missing detectable vulnerabilities in static analysis increases. We already defined how we assess rule reduction for a given category c

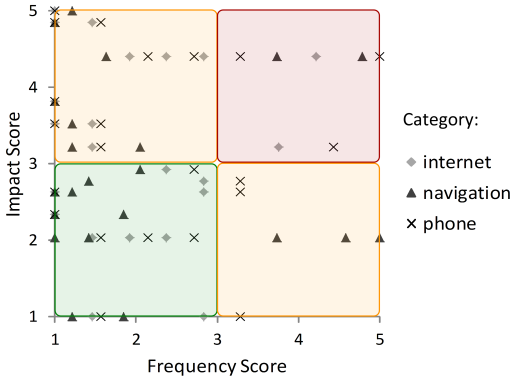

Figure 5: Scatter graph of Android vulnerabilities in a 2-dimensional risk scoring system.

in Equation 2. Similarly, we assess the coverage for a given category c as follows:

$$Coverage_c = \frac{\sum_{r \in \overline{E}_c} \sum_{a \in App_c} |V_{r,a}|}{\sum_{r \in R} \sum_{a \in App_c} |V_{r,a}|} \times 100 \quad (4)$$

Here, $V_{r,a}$ is the set of vulnerabilities in the application a, which are detected by applying rule r.

Figure 4 shows the overall reduction and coverage of all categories for various ϵ values. We calculated these values using *10-Fold Cross Validation* technique [22]. We partitioned the set of apps under study into 10 subsets with equal size and used them to conduct 10 independent experiments. In each experiment, we treated 9 subsets as the training set and the remaining subset as the test set. Recall that our data set comprised of 460 Android applications. We calculated $Reduction_c$ and $Coverage_c$ values for each test set based on the $P(r|c)$ values learned from the corresponding training set. Then, we calculated the intermediate reduction and coverage for each experiment as the weighted average of $Reduction_c$ and $Coverage_c$ values; the weights were assigned proportional to the number of applications fallen in category c for that experiment. Finally, we calculated the overall reduction and coverage as the average of intermediate reduction and coverage values obtained from the 10 experiments.

According to Figure 4, with $\epsilon = 0$ (i.e., when only the rules with learned detection probability of 0 are excluded), reduction is 68%, while coverage is at 100%, meaning that all vulnerabilities that are detectable using all of the rules in our experiment are indeed detected. In other words, the remaining 32% of the rules are as powerful as all of the rules in detecting all of the vulnerabilities and achieving 100% coverage. These results support our hypotheses. They emphasize the effectiveness of our probabilistic ranking as we could achieve full coverage of Android vulnerabilities with 68% reduction of unnecessary rules. In our experiments, no rule was excluded from all categories. This implies that every rule is useful, but, may be unnecessary in some categories.

TABLE 1 shows the *Frequency Score* of the rules for *Internet*, *Navigation*, and *Phone*, which are representative of three Android categories in our study. Recall from 6 that *Frequency Score* is a transformation of $P(r|c)$ to the range of 1-5. As we will see in the remainder of this section, *Frequency Score* will be used in conjunction with the *Impact Score* to calculate risk.

Table 1: Ranking scores for a partial list of Android vulnerabilities.

V_ID	Android Vulnerability	Frequency Score			Impact Score	Risk Score		
		Internet	Navigation	Phone		Internet	Navigation	Phone
v1	IPC Null Check	4.21	4.79	5	4.41	4.31	4.59	4.69
v2	Background Audio/Video	1	1	1	3.81	1.95	1.95	1.95
v3	Retrieves ICC-ID/IMEI/IMSI	2.38	1.42	2.71	2.04	2.2	1.7	2.35
v4	Retrieves Installed Applications	1.46	1.21	1.57	1	1.21	1.1	1.25
v5	Retrieves Location	1.92	5	2.14	2.04	1.98	3.19	2.09
v6	Retrieves Phone Number	2.38	1	2.71	2.04	2.2	1.43	2.35
v7	Sensitive Info to Network	1	1.21	1	5	2.24	2.46	2.24
v8	IPC to Intent Address	1.46	1	1.57	4.85	2.66	2.2	2.76
v9	Unprotected Broadcast Receiver	2.38	2.05	2.71	2.93	2.64	2.45	2.82
v10	Unsafe Pending Intent	3.75	2.05	4.43	3.22	3.48	2.57	3.78
v11	Information Leak to Log	1	1.84	1	2.33	1.53	2.07	1.53
v12	Unprotected Intent Broadcast	2.84	1.42	3.29	2.78	2.81	1.99	3.02
v13	Constant Phone Number for SMS	1	1	1	2.63	1.62	1.62	1.62
v14	Hardcoded Phone Number	2.84	1.21	3.29	2.63	2.73	1.78	2.94
v15	Uses Socket Directly	1.46	1.21	1.57	3.52	2.27	2.06	2.35

8.2 Criticality Ranking

In contrast to *Frequency Score* of a rule, which is dependent on the category of a given application, *Impact Score* of a vulnerability detected by a rule is constant for all categories. To assess the *Impact Score*, we conducted a survey based on the CVSS calculator provided by *National Vulnerability Database (NVD)* [3].

To that end, we asked five security analysts with Android development experience to evaluate the severity of each vulnerability using CVSS calculator. We provided the analysts with the detailed description of each vulnerability and exploitation conditions, along with the CVSS documentations on scoring factors and tips. The average Coefficient of Variance of the estimated scores was 0.15, which is considerably low. In other words, security analysts, more or less, had similar perceptions about the severity of vulnerabilities. Therefore, without any further analysis, we assigned the average score of each vulnerability as an indicator of criticality. TABLE 1 presents the results of our survey as normalized *Impact Scores*.

As shown in TABLE 1, different vulnerabilities have starkly different impacts. This result is useful for the dynamic analysis phase of security testing, as it informs the order in which vulnerabilities detected through static analysis should be explored.

8.3 Risk Ranking

Here we assume that the importance (i.e. weights in Equation 3) of two dimensions of the risk are same. Therefore, the *Risk Score* is calculated as the square root of the product of *Frequency Score* and *Impact Score*. TABLE 1 shows the risk score and its two corresponding factors for several representative Android categories. Figure 5 provides a scatter graph highlighting the risk factor associated with each part of the graph, using the Tulser classification described in Figure 3. We can see that Android vulnerabilities exhibit a wide variety of risk, from severe and frequently encountered to rare and unimportant. This result is highly useful for automated and manual analysis of software, as given an application of a given category, we can infer the most important vulnerabilities that should be checked first.

As mentioned earlier, *prioritized* security inspection is essential, especially in presence of time and budget limitations. Ranking based mechanism is an informed approach to prioritize vulnerabilities based on their importance. We evaluated and compared the effectiveness of the different ranking systems devised in our research (*Rule Ranking*, *Criticality Ranking* and *Risk Ranking*) against *No Ranking*, i.e., uninformed security inspection.

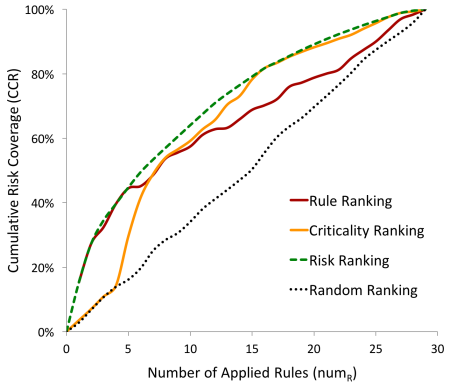

Figure 6: Cumulative Covered Risk (CCR) for 4 ranking systems.

In this regard, we first define the $CoveredRisk_r$ to evaluate the effectiveness of the rule r in detecting the latent vulnerabilities in application set *App*, as follows:

$$CoveredRisk_r = \sum_{a \in App} (|V_{r,a}| \times I_r) \tag{5}$$

Where $V_{r,a}$ is the set of vulnerabilities of type r in the application a and I_r is the impact score of vulnerability r as calculated in the previous subsection. This metric indicates how much risk is covered by vulnerability detection rule r in the set of test applications (i.e., *App*). Therefore, if there is a limitation on the number of rules we would like to statically match, the most successful ranking system is the one that picks up the more powerful rules, i.e., rules with the highest *CoveredRisk*, first.

To compare the success of the four different ranking systems, we define the *Cumulative Covered Risk*, *CCR*, as follows:

$$CCR_{rank,\theta} = \sum_{i=1}^{\theta} CoveredRisk_{r_i^{rank}} \tag{6}$$

where *rank* denotes the type of ranking (i.e., Rule, Criticality, Risk, or No Ranking), θ denotes the number of rules included in the analysis, and r_i^{rank} denotes the ith rule from the ranked list. CCR calculates the total covered risk by applying θ top ranked rules based on the specified ranking method.

Figure 6 shows the plot of *CCR* for various number of applied rules and different ranking systems in Android domain. We can see that *Risk Ranking* is always the best ranking and placed over all other methods, while *Random Ranking*, as the base line, draws a

straight line under the others. It is worth noting that *Rule Ranking* shows to be almost as effective as *Risk Ranking* when only the top most ranked rules are picked, while *Criticality Ranking* shows to be almost as good as *Risk Ranking* only toward the end when most of the rules have been picked. Thus, while *Rule Ranking* and *Criticality Ranking* are the proper ranking system when the number of applied rules is low and high, respectively, *Risk Ranking* seems to be the best overall choice.

9 RELATED WORK

Prior research could be classified into two thrusts: (1) security vulnerability prediction and (2) security assessment of Android applications. In this section, we review the prior literature in light of our approach.

The goal of the first thrust of research is to inform the process of security inspection by helping the security analyst to focus on the parts of the system that are likely to harbor vulnerabilities. An Important distinction between our work and the prior research is the features of application software that are selected for prediction. The majority of the vulnerability prediction approaches are based on software metrics, such as source code [19] and complexity metrics [20], and coverage and dependency measures [25]. In our research, however, we took advantage of categorized software repositories to predict the potential vulnerabilities of an application. In contrast to the prior work, we have used meta-data of apps (i.e., category), which is predefined and does not require any preprocessing techniques, together with the information obtained through static analysis of the code.

The other thrust of related research focuses on security assessment of Android applications. In fact, Android security has received a lot of attention in recently published literature, due mainly to the popularity of Android platform, as well as increasing reports of its vulnerabilities. Such research efforts employ various techniques, including static and dynamic analysis, machine learning, and formal verification, among the others, to identify different security threats in Android applications [16]. Among the large body of research in this area, most related work are those that assess the security *risk* of mobile apps. For instance, RiskMon [11] leverages machine learning techniques to provide a risk assessment framework helping users understand and mitigate security risks of mobile applications. RiskRanker [9] applies a two-layer risk analysis technique to identify mobile malware in a scalable way. Similar to our approach, Sarma et al. [18] use app categories for risk assessment, through combining this information together with the permissions requested by the apps. Distinct from the aforementioned approaches, the goal of our technique is to assess the risk associated with each kind security vulnerability in app categories, instead of security assessment of each individual apps. Such information, however, can be leveraged by security analysts to efficiently assess the security risk of each app.

10 CONCLUSION

In this paper, we leveraged the unique opportunity provided by app markets, i.e., availability of the apps and their meta-data, particularly app categories, to improve the state-of-the-art techniques for risk assessment of Android applications. More specifically, we presented three ways of ranking the security vulnerabilities, namely Rule Ranking, Criticality Ranking, and Risk Ranking, to improve the efficiency and enable prioritization of static, dynamic, and manual analysis, respectively.

As part of our future work, we are interested to extend the research to situations in which an app belongs to more than one category. In addition, in this research we focused on vulnerabilities, which are unintentional mistakes providing exploitable conditions that an adversary may use to attack a system. However, another important factor in security analysis is malicious capabilities, which are intentionally designed by attackers and embedded in an app. Hence, as a complement of this research, we plan to mine the categorized software repositories to improve the malware analysis techniques.

REFERENCES

[1] 2017. F-Droid: Free and Open Source App Repository. (2017). https://f-droid.org/
[2] 2017. Fortify Static Code Analyzer. (2017). https://saas.hpe.com/software/sca
[3] 2017. National Vulnerability Database CVSS Scoring. (2017). https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator
[4] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. 2011. AEG: Automatic exploit generation. In *Proceedings of the Network and Distributed System Security Symposium*.
[5] Dimitri P. Bertsekas and John N. Tsitsiklis. 2008. *Introduction to Probability, 2nd Edition.* Athena Scientific.
[6] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. Analyzing inter-application communication in Android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services (MobiSys '11)*. ACM, New York, NY, USA, 239–252.
[7] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. 2011. A study of android application security. In *Proceedings of the 20th USENIX security symposium*, Vol. 2011.
[8] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. 2014. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 1025–1035.
[9] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. 2012. Riskranker: scalable and accurate zero-day android malware detection. In *The 10th International Conference on Mobile Systems, Applications, and Services, Ambleside, United Kingdom - June 25 - 29, 2012*. Washington, DC, 281–294.
[10] CVSS-SIG group. 2007. Common Vulnerability Scoring System (CVSS-SIG). (2007). www.first.org/cvss
[11] Yiming Jing, Gail-Joon Ahn, Ziming Zhao, and Hongxin Hu. 2015. Towards Automated Risk Assessment and Mitigation of Mobile Applications. *IEEE Trans. Dependable Sec. Comput.* 12, 5 (2015), 571–584.
[12] Mario Linares-Vasquez, Collin McMillan, Denys Poshyvanyk, and Mark Grechanik. 2012. On using machine learning to automatically classify software applications into domain categories. *Empirical Software Eng.* (2012), 1–37.
[13] Jack T. Marchewka. 2009. *Information Technology Project Management.* Wiley.
[14] Gary McGraw. 1997. Testing for security during development: why we should scrap penetrate-and-patch. In *Are We Making Progress Towards Computer Assurance? Proceedings of the 12th Annual Conference on Computer Assurance, 1997. COMPASS '97*. 117–119.
[15] G. McGraw. 2008. Automated Code Review Tools for Security. *Computer* 41, 12 (2008), 108–111.
[16] Alireza Sadeghi, Hamid Bagheri, Joshua Garcia, and Sam Malek. 2016. A Taxonomy and Qualitative Comparison of Program Analysis Techniques for Security Assessment of Android Software. *IEEE Transactions on Software Eng.* (2016).
[17] Alireza Sadeghi, Naeem Esfahani, and Sam Malek. 2014. Mining the Categorized Software Repositories to Improve the Analysis of Security Vulnerabilities. In *Fundamental Approaches to Software Engineering (FASE)*. 155–169.
[18] Bhaskar Pratim Sarma, Ninghui Li, Christopher S. Gates, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. 2012. Android permissions: a perspective combining risks and benefits. In *17th ACM Symposium on Access Control Models and Technologies, Newark, NJ, USA - June 20 - 22, 2012*. 13–22.
[19] Riccardo Scandariato and James Walden. 2012. Predicting vulnerable classes in an Android application. In *Proceedings of the 4th international workshop on Security measurements and metrics*. 11–16.
[20] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A. Osborne. 2011. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Eng.* 37, 6 (2011), 772–787.
[21] Symantec Corp. 2012. 2012 Norton Study. (Sept. 2012). www.symantec.com/about/news/release/article.jsp?prid=20120905_02
[22] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. 2005. *Introduction to Data Mining* (1 ed.). Addison Wesley.
[23] James Walden and Maureen Doyle. 2012. SAVI: Static-analysis vulnerability indicator. *Security & Privacy, IEEE* 10, 3 (2012), 32–39.
[24] T.M. Williams. 1993. Risk-management infrastructures. *International Journal of Project Management* 11, 1 (1993), 5–10.
[25] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. 2010. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*. 421–428.