Mining the Categorized Software Repositories to Improve the Analysis of Security Vulnerabilities

Alireza Sadeghi, Naeem Esfahani, and Sam Malek

Department of Computer Science
George Mason University
{asadeghi,nesfaha2,smalek}@gmu.edu

Abstract. Security has become the Achilles' heel of most modern software systems. 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, these techniques are time consuming and cannot keep up with the complexity of ever-growing software repositories (e.g., Google Play and Apple App Store). In this paper, we aim to improve the status quo and increase the efficiency of static analysis by mining relevant information from vulnerabilities found in the categorized software repositories. The approach relies on the fact that many modern software systems 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. We used open-source software repositories comprising more than 7 million lines of code to demonstrate how our approach can improve the efficiency of static analysis, and in turn, vulnerability detection.

Keywords: Security Vulnerability, Mining Software Repositories, Software Analysis

1 Introduction

According to the Symantec's Norton report [1], in 2012 the annual financial loss due to cybercrime exceeded $110 billion globally. An equally ominous report from Gartner [2] predicts 10 percent yearly growth in cybercrime-related financial loss through 2016. This growth is partly driven by the new security threats targeted at emerging platforms, such as Google Android and Apple iPhone, that provision vibrant open-access software repositories, often referred to as *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 application software, but the paradigm shift has also given rise to a whole host of security issues [1]. Numerous culprits are at play here, and some are not even technical,

such as the general lack of an overseeing authority in the case of open markets and inconsequential punishment for those caught provisioning applications with vulnerabilities or malicious capabilities.

From the standpoint of application security, 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 [3]. 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 code analysis is gaining popularity for automatically finding security problems in application software [4].

While more efficient than manual inspection, the ability to improve the efficiency of static code analysis is gaining prominence for two reasons: (1) App market operators and overseeing authorities need to employ source code analysis techniques in large scale. On September 26, 2012 Android team unveiled that Google Play hosts 675,000 apps [5]. In less than 10 months, the number of apps in Google Play hit 1,000,000, meaning that more than 1,000 apps were added per day during that time period. On top of this, thousands of apps are updated every day that also need to be analyzed. (2) Recent research [6] has shown the benefit of continuously running static analysis tools in *real-time* and as programmers are developing the code, thereby helping them catch vulnerabilities earlier in the development phase. In such settings, even a slight improvement in efficiency is highly desirable and sought-after.

An opportunity to tackle this issue is presented by the fact that software products are increasingly organized into categorized repositories, where each item is mapped to a flat or hierarchical category. Some examples are SourceForge for open source and Google Play for Android applications. Other than facilitating the users in searching and browsing, categorized repositories have shown to be good predictors of the common features found within software of a particular category [7].

In this paper, we explore the utility of categorized repositories in informing the security inspection and analysis of software applications. The fact that the majority of apps provisioned on such repositories are built using a common *application development framework (ADF)* presents us with an additional opportunity. The information encoded in the source code of software developed on top of an ADF (e.g., Android) is richer than information encoded in the source code of traditional software (e.g., one developed from scratch in Java or C++). The reason for this is that an app developed on top of an ADF leverages libraries, services, and APIs provisioned by the ADF that disclose a significant amount of information about the app's behavior/functionality. This information can be used for various purposes, including security assessment, since many of the security issues encountered in modern software are due to the wrong usage of ADF [1]. In this paper, we show how this information can be used to build a predictor for vulnerabilities one may find in the app of a particular category. This result is important, as it allows us to improve the efficiency of static analysis techniques for security assessment of software.

Running all possible static analysis rules, which encode patterns of vulnerability one may find in the code, on an application software is a time consuming and resource intensive process. For instance, in our experiments, in some cases it took up to 5.4 hours and 1.3 hours to statically analyze a single Java and Android application, respectively. Given an app belonging to a given category, we are able to use our predictor to focus the analysis on the vulnerabilities that are commonly encountered in that category. The predictor helps us pick and apply the static analysis rules that are empirically shown to be effective for the different categories of apps.

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 in the case of Android apps by 68%, while keeping the vulnerability detection rate at 100%. The results are useful, although not as significant, for plain Java applications that do not make use of any ADF, leading to efficiency improvement of 37%, while 4% of vulnerabilities are missed.

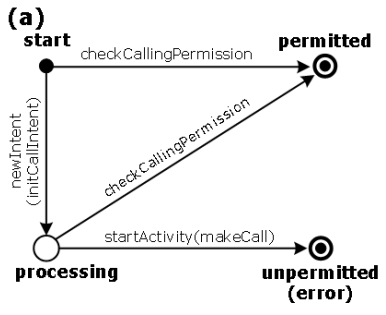
The remainder of this paper is organized as follows. Section 2 provides the required background as well as motivation for this work. Section 3 outlines the overview of our approach, while Section 4 describes the details. Sections 5 and 6 describe the research experimental setup, results, and analysis. Section 7 outlines the threats to validity of our experiments. Finally, the paper concludes with a discussion of related research and our future work.

2 Background and Motivation

Static analysis entails analysis of computer software without actually executing the software. While static analysis techniques originated in the compiler community for optimization of source code [8], they have found new applications in the past decade, as they have also shown to be effective for finding vulnerabilities in the source code [4]. Due to the complexity of statically reasoning about source code, static analysis is usually performed on an abstract model of code, such as control flow or data flow. The type of analysis done depends on the types of vulnerabilities that one aims to find. In this research, we have used three static analysis techniques for detecting vulnerabilities in Android apps: content, data flow, and control flow analysis

Content analysis deals with the pre-specified values that are known to create vulnerabilities in a program. By detecting "bad" content, the vulnerability can be discovered and prevented. For instance, a well-known attack against any phone app with communication capabilities is to trick that app to communicate (e.g., call, text message, etc.) with premium-rate numbers, which can be prevented by detecting the pattern of premium-rate numbers.

Data flow analysis considers the flow of information in the system tracking how data from input sources, such as phone identification, user input, and network interface lead to to output sinks. Data flow analysis is an effective approach for detection of information leak or malicious input injection vulnerabilities by identifying unsafe sinks and input values respectively. Android apps have plenty



(b)

```
state start (start);
state processing;
state permitted;
state unpermitted (error);
var call_intent;
start ->
    permitted {$checkCallingPermission(...)}
start ->
    processing {call_intent.$initCallIntent(...)}
processing ->
    permitted {$checkCallingPermission(...)}
processing ->
    unpermitted {$makeCall(call_intent)}
```

Fig. 1. An example of a sequence pattern used by control flow analysis to detect privilege escalation vulnerability: (a) state model representing the pattern and (b) realization of the pattern as a rule in Fortify.

of vulnerabilities that can be detected by this approach. In a recent study, Enck et al. [9] found that about 17% of the top free Android apps on the Google Play market transmit private user information, such as the phone's ID or location information over the web. They also reported that some apps log this private information to the phone's shared log stream, which is accessible to other apps running on the phone.

Control flow analysis entails processing the sequence of program execution, such as method calls. When a vulnerability can be modeled as a sequence of actions, we use control flow analysis. In this case, static analysis determines whether program execution can follow a certain sequence of actions that lead to a vulnerable status. For instance, control flow analysis can be used to detect privilege escalation in Android apps. Privilege escalation occurs when a malicious app exploits a second app to access resources that it does not have permission.

Fig. 1a depicts a control flow sequence pattern that we have developed for detecting privilege escalation vulnerability. Control flow sequence patterns model the transition of the system from a start state to two final states (one for error and one for success) through several possible intermediate states. The error state in Fig. 1a is a result of a flow in the program, where *startActivity* occurs without checking the permission of the calling application (by calling *checkCallingPermission* method). Fig. 1b shows the realization of this pattern as a rule in Fortify Static Code Analysis environment [10].

Fortify is a powerful static analysis tool, however, there are plenty of analysis tools that could reveal different kinds of vulnerabilities. Ware and Fox [11], used eight different static analysis tools (namely Checkstyle [12], Eclipse TPTP [13], FindBugs [14], Fortify [10], Jlint [15], Lint4j [16], PMD [17], and QJ-Pro [18]) to identify the vulnerabilities in Java projects. Then they compared the result of using these tools side-by-side. Among 50 distinct vulnerabilities detected by combination of eight tools, no individual tool could detect more than 27 distinct items. This implies that using a single tool increases the chance of missing vulnerabilities (i.e., having false negatives). Therefore, one should apply various tools with many detection rules. However, this affects the analysis time and ham-

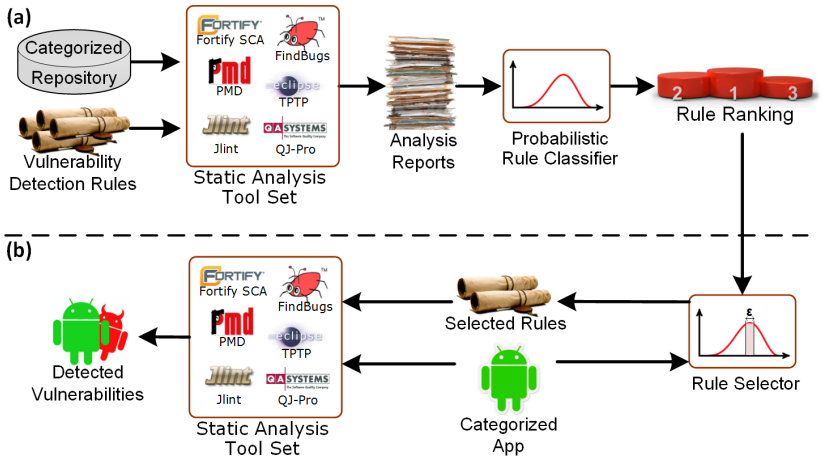


Fig. 2. Overview of the approach: (a) rank rules and (b) efficient vulnerability analysis.

pers the efficiency. This is exactly the challenge that we are aiming to resolve in this paper. In the next section, we provide an overview of our approach, which prioritizes the rules based on the likelihood of detecting vulnerabilities. Applying rules based on their priorities improves the efficiency of static analysis.

3 Approach Overview

Fig. 2 depicts an overview of our approach. *Categorized Repository* of software applications is the first input to our framework. In this repository, each application is labeled with a predefined class or category. Here, we assume such categorized applications can be gathered from on-line repositories (e.g., F-Droid and SourceForge) without any classification effort. Otherwise, machine learning techniques could be used to find the category of each application [7].

The second input to our framework is *Vulnerability Detection Rules*, which define the interesting patterns in the source code. Since our research focus is on security issues, we are interested in the rules that define patterns of vulnerability in the code.

Static Analysis Tool Set inspects the code repository and looks for any instance that matches the patterns defined in the rules. The result is an *Analysis Report*. *Analysis Report* consists of all locations in the code that are detected as potential vulnerabilities. Static analysis recurs for each application in the repository and generates the corresponding report.

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* ranks each vulnerability based on its frequency in the *Analysis Report*. In this regard, *Probabilistic Rule Classifier* applies conditional probability to find the likelihood of occurrence of each vulnerability in each category. The

result of this is *Rule Ranking*. In this ranking, a frequency score is assigned to each security rule for a given category. Higher score means that it is more likely for the corresponding rule to detect a vulnerability in that category.

Fig. 2b depicts the application of *Rule Ranking* in improving the analysis of vulnerabilities. *Rule Selector* uses the category of a given *Categorized App* and picks the most efficient rules from *Vulnerability Detection Rules* for that category based on *Rule Ranking*. *Static Analysis Tool Set* uses *Selected Rules* to efficiently analyze the *Categorized App* and detect its possible vulnerabilities, which are reported as *Detected Vulnerabilities*.

4 Probabilistic Rule Classification and Selection

As depicted in Fig. 2, the result of running Static Analysis Tool Set is the Analysis Report. This report contains the application's source code locations that match the predefined vulnerability patterns specified in Vulnerability Detection Rules. The tool set 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 analysis by removing the irrelevant rules (i.e., $\overline{M} = R - M$). We call this rule reduction.

We can extend our definition by considering the categorical information. Applications categorized in the same class have some common features implemented by similar source code patterns and API calls to common libraries [7]. Consequently, it is more likely for a set of applications in a given category $c \in C$ (where C is the set of all categories) to have common vulnerabilities. We use this insight and extend our definition as follows: M_c is the set of rules that are matched at least once inside an application with category c.

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 vulnerability in that category.

By applying Bayes Theorem [19] to the Analysis Reports (recall Fig. 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 in static analysis. 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 indicate 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 analysis is done faster, but the coverage decreases, increasing the chances of missing a vulnerability in the code. We will discuss the selection of threshold in Section 6.

5 Experiment Setup

The first step for using our approach is to populate Categorized Repository and Vulnerability Detection Rules (depicted as the two inputs in Fig. 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 *App* and R for our evaluation purposes and set up the experiments. We evaluated our approach on applications developed using Java and Android (as a representative ADF). [1]

We considered applications 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 (e.g., F-Droid and Google Play) 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, the best candidates for Java and Android are Source Forge and F-Droid, respectively. The additional benefit of using Source Forge and F-Droid together is that they have categorized the applications very similarly, allowing us to compare the results from these repositories. Table 1 shows the number of applications gathered from each category. We depict the set of applications in the same category c as App_c. Categories with two labels (one in parentheses) indicate alternative category names used in the two repositories. For instance, the *Multimedia* category in Android, is called *Audio/Video* category in Java.

[1] Research artifacts and experimental data are available at `http://www.sdalab.com/projects/infovul`.

Table 1. Number of application in each category.

C_ID	Category	Java	Android
c1	Business-Enterprise	47	-
c2	Communications (Phone)	55	14
c3	Development	54	17
c4	Game	30	51
c5	Graphics	29	-
c6	Home-Education	51	21
c7	Internet	30	53
c8	Multimedia (Audio-Video)	51	51
c9	Navigation	-	32
c10	Office	-	101
c11	Reading	-	17
c12	Science-Engineering	42	-
c13	Security-Utilities	17	-
c14	System	35	95
c15	Wallpaper	-	8
Total Number of Applications		441	460

Table 2. Experiment environment statistics.

Experiment Stats	Java	Android
Total Lines of Code	6,166,755	1,360,881
Number of Categories	11	11
Number of Exclusive Rules	156	50
Total Number of Vulnerabilities	38,312	2,633

We used HP Fortify [10] as the main static analysis tool (recall *Static Analysis Tool Set* from Fig. 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 Java we utilized built-in rules provided by Fortify, while for Android we used rules provided by Enck et al. [9]. However, as we mentioned in Section 2, a single tool has a high chance of missing some of the vulnerabilities. Hence, we also used FindBugs [14], Jlint [15], and PMD [17] to reinforce the static analysis and reduce the false negative rate.

We ran Static Analysis Tool Set with the inputs discussed above to detect vulnerabilities in the experiments and prepare Analysis Report. The results are summarized in Table 2. We then fed the Analysis Report to Probabilistic Rule Classifier to calculate the Rule Ranking for each category. Before delving into the effects of this ranking on the static analysis, which is presented in Section 6, we provide some additional insights about the generated Analysis Report in this section. To that end, we extracted the detected vulnerabilities in each category from Analysis Reports to profile the applications in our study. However, since the number of applications under study in each category (i.e., $|App_c|$) are different, the raw measurements are misleading. Therefore, we define the *Density of*

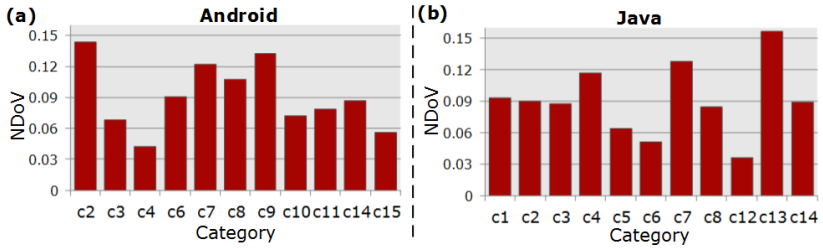


Fig. 3. Normalized Density of Vulnerabilities ($NDoV$) based on application categories for different domains: (a) Android and (b) Java.

Vulnerability (DoV_c) metric for a given category c as follows:

$$DoV_c = \frac{\sum_{r \in R} \sum_{a \in App_c} |V_{r,a}|}{|App_c|} \tag{3}$$

Here, $V_{r,a}$ is the set of vulnerabilities in the application a, which are detected by applying rule r. Since the total number of vulnerabilities may be different for Android and Java, the DoV_c is not comparable between the two domains. Therefore, we define *Normalized Density of Vulnerability* ($NDoV_c$) for a given category c as follows:

$$NDoV_c = \frac{DoV_c}{\sum_{c \in C} DoV_c} \tag{4}$$

Fig. 3, which is practically the probability distribution of the vulnerabilities in the two experiments, presents $NDoV_c$ values for Android and Java domains. In Android domain, *Communication* (c2) and *Game* (c4) are the most vulnerable and safest categories respectively. This result is reasonable, as the applications in the *Communication* category call many security-relevant APIs (e.g., *Telephony*), while applications in the *Game* category mostly call benign APIs (e.g., *Graphic*). We observe similar trends for Java domain, where *Security-Utilities* (c13) and *Science-Engineering* (c12) have the highest and lowest vulnerability rates, respectively. The results show that the different categories have starkly different levels of vulnerability.

6 Evaluation

The ultimate contribution of our research is to improve the efficiency of software security analysis. Therefore, evaluation of our approach entails measuring the efficiency improvement from using the suggested ranking. Additionally, we want to investigate our hypothesis that the abstractions afforded by ADF indeed enable further efficiency gains.

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 have already described how we exclude rules (recall E_c

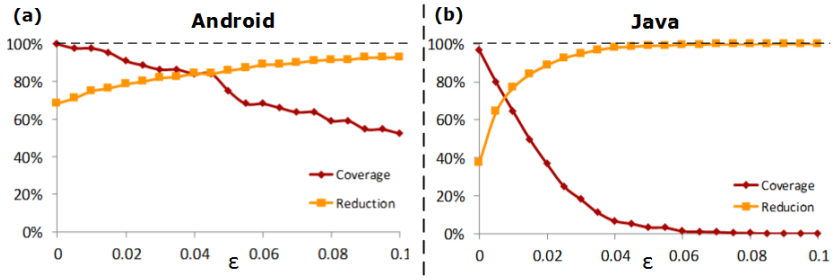


Fig. 4. The overall reduction vs. the overall coverage of the remaining rules for: (a) Android and (b) Java.

from Section 4) and assess rule reduction for a given category c (recall Equation 2). Coverage of the remaining rules (i.e., $\overline{E}_c = R - E_c$) for a given category c represents the percentage of the vulnerabilities detected in that category by only using the rules that are not excluded and is defined as follows:

$$Coverage_c = \frac{\sum_{r \in \overline{E}_c} \sum_{a \in App_c} |V_{r,a}|}{\sum_{r \in R} \sum_{a \in App_c} |V_{r,a}|} \times 100 \tag{5}$$

Fig. 4 shows the overall reduction and coverage of all categories for various ϵ values in Android and Java domains. We calculated these values using 10-Fold Cross Validation technique [20]. 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 from Table 2 that our data set comprised of 441 Java and 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 Fig. 4, in Android domain, 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. However, in Java domain, the results are significantly different as reduction with $\epsilon = 0$ is 37%. Additionally, the remaining 63% of rules can only provide 96% coverage. In other words, they are not as powerful as all of the rules in detecting Java vulnerabilities.

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. Moreover, as we expected, the use of ADF has a positive influence on the effectiveness of our approach. This is because the rules specified in terms of ADF are at a level of abstraction that lend themselves naturally to positive or negative correlation with a particular application category. The results show that in the domains where ADFs are heavily used, our approach could be used to significantly improve the performance of static analysis by removing the irrelevant rules.

The divergence of coverage and reduction in Java domain is very high compared to Android domain. The loss of coverage with $\epsilon = 0.04$ for Android is 16%, while for Java the loss is 93%. This clearly shows that our approach is most effective in domains where an ADF is used for the implementation of application software. As a result, in the remainder of evaluation, we focus on the results obtained in Android domain.

Table 3 provides the detailed results of experiments for Android apps when $\epsilon = 0$. The number of excluded rules (i.e., $|E_c|$) in the categories varies between 24 to 43 rules out of total 50 vulnerability detection rules. In other words, $Reduction_c$ is between 48% to 86% for different categories that leads to the average reduction of 68%, as shown before in Fig. 4a.

Table 3 presents the average and 95% confidence interval for the analysis time of an Android app in each category. Here, *Uninformed* column corresponds to the time spent for source code analysis if all vulnerability detection rules (i.e., R) are applied, while *Informed* is when unnecessary rules (i.e., E_c) are excluded from the analysis procedure. The last column in Table 3 shows the significant time savings by pruning the useless rules from source code analysis. From the last row of the table, we can see that our approach on average achieves a 67% speed up compared to the uninformed approach of simply analyzing the source code for all types of vulnerability.

Table 3. Analysis time for Android apps for $\epsilon = 0$.

Android Category	Rule Exclusion $\|E_c\|$	Rule Exclusion $Reduction_c$	Analysis Time (mins) Uninformed	Analysis Time (mins) Informed	Saved Analysis Time (mins)
Development	43	86	43.48±8.3	6.09±1.16	37.39±7.14
Education	35	70	59.29±17.76	17.79±5.33	41.5±12.43
Game	39	78	48.92±4.94	10.76±1.09	38.16±3.85
Internet	33	66	49.83±6.65	16.94±2.26	32.89±4.39
Multimedia	29	58	60.47±11.22	25.4±4.71	35.07±6.51
Navigation	31	62	44.61±8.25	16.95±3.14	27.66±5.12
Office	30	60	44.25±4.11	17.7±1.64	26.55±2.46
Phone	30	60	70.36±22.47	28.14±8.99	42.21±13.48
Reading	40	80	62.4±18.25	12.48±3.65	49.92±14.6
System	24	48	47.17±11.93	24.53±6.2	22.64±5.73
Wallpaper	42	84	44.27±12.04	7.08±1.93	37.19±10.12
All categories	34	68	52.28±11.45	16.54±3.62	35.74±7.83

7 Threats To Validity

With regard to the internal threats, there is one issue. Since we needed access to open source applications for our experiments, the training set was limited to 460 open-source Android apps, which is small in comparison to 700,000 (not necessarily open source) apps currently available on Google Play. However, since we have covered almost all available categories that exist in Google Play with a similar app distribution, our experimental app set could be considered as an admissible representation of the global Android app market. Extending our study to a larger set of applications is likely to improve the accuracy of our approach.

An external threat is related to our non-overlapping decisive app categorization method. In this research, we have assumed each app belongs to a single prespecified category. This is a reasonable assumption as many app markets (e.g., F-Droid as well as Google Play) assign an app to one category. But when software repositories allow an application to belong to multiple categories, an app may possess the features, and thus vulnerabilities of more than one category. Our approach in its current form is not applicable to such settings. For this we would need to precede our approach with a preprocessing step in which we first determine the category that best matches the characteristics of an application, or alternatively provide a probabilistic measure of confidence with which the application belongs to a particular category.

8 Related Work

Prior research could be classified into two thrusts: (1) security vulnerability prediction and (2) Android security threats and analysis techniques. 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. While most prior approaches on vulnerability prediction are platform-independent and try to predict the occurrence of vulnerability regardless of the application domain [21–23], some have focused on a specific platform or domain, such as Android [24] or Microsoft Windows vulnerabilities [25].

An Important distinction between our work and the prior research is the features of application software that are selected for prediction. Some vulnerability prediction approaches are based on various software metrics. For example Scandariato and Walden [24] have considered a variety of source code metrics, including size, complexity and object-oriented metrics, Shin et al. [23] have applied complexity, code churn, and developer activity metrics, and Zimmermann et al. [25] have used the same metrics together with coverage and dependency measures. Some other vulnerability prediction approaches have considered the raw source code and applied text retrieval techniques to extract the features. For example, Hovsepyan et al. [21] have transferred Java files into feature vectors, where each feature represents a word of the source code, while Neuhaus et

al. [22] have not included all of the words in the source code in the analysis, and instead established a correlation between vulnerabilities, imports, and function calls.

In our research, 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. We believe our approach complements the prior research, as it presents an alternative method of detecting and classifying presence of vulnerabilities.

The second thrust of research has studied Android security threats and analysis techniques at different levels of system stack, from operating system level [26, 27] to application level [9, 27–29]. However, in most cases, Android architecture and its security model have been the main focus of the study [26–28], as opposed to the vulnerabilities that arise in the application logic. Shabtai et al. [26] have clustered security threats based on their risk, described the available security mechanism(s) to address each threat, and assessed the mitigation level of described solutions. Enck et al. [27] have enumerated security enforcement of Android at two levels: system level and inter-component communication level. They have developed a tool, named Kirin, to check the compliance of described security mechanism with Android apps. Enck et al. [9] have also investigated vulnerabilities of 1,100 Android apps by using static analysis. In this regard, they have provided a set of vulnerability detection rules, which we have used in our research.

A body of prior research has tried to automate the security testing of Android apps. Mahmood et al. [28] suggested a whitebox approach for testing Android apps on the cloud, where test cases are generated through lightweight program analysis. In another research, Gilbert at el. [29] suggested AppInspector, which tracks and logs sensitive information flows throughout app's possible execution paths and identifies security or privacy violations. Unlike our work, all prior research has implicitly assumed that various vulnerabilities have the same likelihood, and consequently tackled them with equal priority. Our research complements prior research by prioritizing the order in which vulnerabilities are analyzed and tested.

9 Conclusion

The ability to streamline the security analysis and assessment of software is gaining prominence, partly due to the evolving nature of the way in which software is provisioned to the users. We identified two new sources of information that when mined properly present us with a unique opportunity to improve the state-of-the-art (1) meta-data available in the form of application category on such repositories, and (2) vulnerabilities specific to the wrong usage of *application development framework (ADF)*.

In summary, the contributions of our work are as follows: (1) We were able to derive a strong correlation between software categories and security vulnerabilities, in turn allowing us to eliminate the vulnerabilities that are irrelevant for a given category. Most notably, we showed that we can achieve 68% reduction in the vulnerability detection rules, while maintaining 100% coverage of the detectable vulnerabilities in Android. (2) We developed a probabilistic method of ranking the rules to improve the efficiency and enable prioritization of static analysis for finding security vulnerabilities. (3) We empirically demonstrated the benefits of ADF in the security vulnerability assessment process. An app developed on top of an ADF leverages libraries, services, and APIs provisioned by the ADF that disclose a significant amount of information about the app's behavior/functionality. We showed how this information can be used to predict vulnerabilities one may find in the app of a particular category.

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.

Acknowledgements

This work was supported in part by awards W911NF-09-1-0273 from the US Army Research Office, D11AP00282 from the US Defense Advanced Research Projects Agency, and CCF-1252644 and CCF-1217503 from the US National Science Foundation.

References

1. Symantec Corp.: 2012 norton study (2012)
2. Gartner Inc.: Gartner reveals top predictions for IT organizations and users for 2012 and beyond (2011)
3. McGraw, G.: 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. (1997) 117–119
4. McGraw, G.: Automated code review tools for security. Computer **41** (2008) 108–111
5. Android: Official blog. (officialandroid.blogspot.com)
6. Muslu, K. et al.: Making offline analyses continuous. In: Int'l Symp. on the Foundations of Software Engineering, Saint Petersburg, Russia (2013) 323–333
7. Linares-Vsquez, M. et al.: On using machine learning to automatically classify software applications into domain categories. Empirical Software Engineering (2012) 1–37

8. Binkley, D.: Source code analysis: A road map. In: Int'l Conf. on Software Engineering, Minneapolis, Minnesota (2007) 104–119
9. Enck, W. et al.: A study of android application security. In: Proceedings of the 20th USENIX security symposium. Volume 2011. (2011)
10. HP Enterprise Security: (Static application security testing)
11. Ware, M.S., Fox, C.J.: Securing java code: heuristics and an evaluation of static analysis tools. In: Proceedings of the 2008 workshop on Static analysis. SAW '08, Tucson, Arizona, ACM (2008) 12–21
12. Checkstyle: Enforce coding standards. (checkstyle.sourceforge.net)
13. Eclipse: Eclipse test & performance tools platform project. (www.eclipse.org/tptp)
14. Hovemeyer, D., Pugh, W.: Finding bugs is easy. ACM Sigplan Notices **39** (2004) 92–106
15. Jlint: Find bugs in java programs. (jlint.sourceforge.net)
16. Lint4j: Lint4j overview. (www.jutils.com)
17. PMD: Source code analyzer. (pmd.sourceforge.net)
18. QJ-Pro: Code analyzer for java. (qjpro.sourceforge.net)
19. Bertsekas, D.P., Tsitsiklis, J.N.: Introduction to Probability, 2nd Edition. Athena Scientific (2008)
20. Tan, P.N. et al.: Introduction to Data Mining. 1 edn. Addison Wesley (2005)
21. Hovsepyan, A. et al.: Software vulnerability prediction using text analysis techniques. In: Proceedings of the 4th international workshop on Security measurements and metrics. (2012) 7–10
22. Neuhaus, S. et al.: Predicting vulnerable software components. In: Proceedings of the 14th ACM conference on Computer and communications security. (2007) 529–540
23. Shin, Y. et al.: Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. Software Engineering, IEEE Transactions on **37** (2011) 772–787
24. Scandariato, R., Walden, J.: Predicting vulnerable classes in an android application. In: Proceedings of the 4th international workshop on Security measurements and metrics. (2012) 11–16
25. Zimmermann, T. et al.: 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. (2010) 421–428
26. Shabtai, A. et al.: Google android: A comprehensive security assessment. Security & Privacy, IEEE **8** (2010) 35–44
27. Enck, W. et al.: Understanding android security. Security & Privacy, IEEE **7** (2009) 50–57
28. Mahmood, R. et al.: A whitebox approach for automated security testing of android applications on the cloud. In: 2012 7th International Workshop on Automation of Software Test (AST). (2012) 22 –28
29. Gilbert, P. et al.: Vision: automated security validation of mobile apps at app markets. In: Proceedings of the second international workshop on Mobile cloud computing and services. (2011) 21–26