Imperial College London

BEng Individual Project

Imperial College London

Department of Computing

Changeset metrics and Code metrics for browser defect prediction

Author:
Son Dang

Supervisor:
Dr. Ben Livshits

Second Marker:
Dr. Sergio Maffeis

June 17, 2018

Abstract

Web browsers are the doors that connect users to the World Wide Web. With malwares trying to infect personal devices and data on a daily basis, securing vulnerabilities in browsers is a major step to minimizing the damage and protecting millions of users. Being able to detect and predict possible vulnerabilities can play a crucial role in responding to an attack and help save both time and manpower.

Code change measures how much of the code is added, deleted or modified. It is suggested that the larger the code and the more often it is changed, the more likely it is to be defective.

In this project, we developed a mining tool for code change statistics of defected files in the Chromium Browser. It scrapes the Chromium Bug Blog for necessary data, which is then calculated and presented in the form of a web application. By providing additional information about changeset as well as other code change statistics from fixed bugs, a new aspect of the relationship between files is revealed to developers, which may accelerate the progress of debugging vulnerabilities.

A Naive Bayes classifier was also built to further analyze the relevance of code change metrics to vulnerabilities, with the purpose of predicting defective files. Using the mined data as training set and testing set, the results show a positive correlation between code change metrics and vulnerabilities.

Acknowledgements

I would like to express my very great appreciation to

DR BEN LIVSHITS,

for his valuable suggestions and constructive guidance during the planning and development of this project,

MY FRIENDS,

for their genuine companionship and useful insights of Cyber Security throughout our discussion,

THU LE,

for her thoughtful second opinion on the project evaluation and testing,

and MY PARENTS,

for their unconditional love and support.

Contents

List of Figures

Chapter 1

Introduction

Chrome is accounted for 60.98% of browser market share over the recent year, compared to its main rivals Firefox and Internet Explorer with a total of less than 25%. Its popularity has made Chrome an attractive target to security exploitations[1]. Being built using the source code of Chromium which is an open-source project by Google, it is even more encouraging for hackers to try and debug the code for vulnerabilities. Luckily, the community of Chromium users and developers is also doing their best to report and fix bugs before they are exploited.

With the size of 1.1 GB, thousands of files and millions of lines of code, debugging Chromium is a challenging task, not to mention a large proportion of legacy code of 10 years old. As the code gets bigger in size, it is more complicated and takes relatively longer to analyze. Without the experience of fixing similar bugs in the past, developers may find themselves switching back and forth among several files for hours before understanding the functionality of the code. If only there was a quicker and more efficient way to show the relationships between files without reading a single line of code.

1.1 Objectives

It is normal for a file **A** to use a function or a class from another file **B**. Therefore, when there are changes in B due to a faulty class, **A** is more likely to be modified compared to an unrelated file **C**. Hence, when a file is reported by users to have a bug, besides reading through that file only, it is crucial to follow the function call and check other relevant files as well. Not only will the bug be fixed thoroughly, but similar bugs will be prevented from happening in the group of linked files in the future.

Without access to the Chromium website database, our first step is to mine the Chrome Stable releases changelog for all the modified files. This would provide us with files that have their code changed in the recent time period. By monitoring them over several releases, we could learn more about the relationship between the amount of code changed and the probability of being defective. It has been proven in many pieces of research that the more modification a file had, the more likely it was to involve in bugs and vulnerabilities in the future.

Our main goal is to build a classifier using machine learning, to test out the correlation between defective and code changes. We would expect the tool to have a high Recall rate and Precision rate, and return results in a short time. This could be useful for developers when they respond to a new bug and time becomes a crucial factor. The faster they could fix the bug, the less damage could be done by hackers. We would also want the tool to be scalable, as adding more and more files to the data set should not affect its performance, but make it even more accurate at predicting vulnerabilities.

1.2 Challenges

Within a large amount of data, it is difficult to choose which metrics to be used when analyzing the files. On one hand, more data means our suggestions will have greater accuracy. On the other hand, we also want it to be quick enough so that developers can refer to it before they start debugging.

A classifier may not work properly due to the lack of direct links between files and bugs. A commit fixing a bug may include 20 files, with only 1 to 2 files actually containing the vulnerable piece of code. With hopes of a large training set, we may be able to work out the right setting for our classifier.

A small dataset might be biased. As observed in the Chrome releases, some commits contain hundreds of files compared to an average value of around 5. By adding them to the data set, they can affect the results by increasing the mean of the number of change set significantly. However, they can also be a useful indicator of non-defective files, which might improve the accuracy of the classifier instead of worsening it.

1.3 Contributions

- A python app that is able to scrape information from Chromium Bug Blog and Chromium Gerrit in both HTML and JavaScript form, extract all information from Bug ids, list of files involved and their code change metrics.
- A classifier using Naive Bayes to predict files that contain vulnerabilities.
- An implementation of a Django web application using Model-View-Control, front-end HTML, backend PostgreSQL.

Chapter 2

Background

In this chapter, we will present the researches and ideas related to the project. It is important to understand the different ways to link files and bugs that will support vulnerabilities predictions. We will also consider which code change metrics are more useful than others to be able to choose the most suitable one. Finally, we will explore the existing works of similar tools for other browsers such as Firefox, and study what can be used to improve our tool on Chromium.

2.1 Different Code metrics

2.1.1 Added, Deleted

This metric is measured by the number of lines of code added and deleted after a commit. Code changes constantly as the project grows, more features are added, old code is replaced by new code[2]. This opens up opportunities for new vulnerabilities to form, which makes it reasonable to try to identify them. The idea is that the number of lines of code changed has a positive correlation with the probability of defectiveness [3][4].

This is one of the more simple yet efficient ways to show how files are linked. Many attempts have been made to mine the added and deleted lines of codes from the repository of the source code, from Firefox Web Browser[5], Apache HTTP Web Server to MySQL Database Server[6]. The common feature of these project is the large size of source code, which makes them quite time-consuming if a more complicated metric is applied. The data of added and deleted lines for every file is readily recorded in each git commit of the project. Therefore, it is efficient for developers who want to form data sets of their preferred subdirectories to just run a simple **git diff** command.

2.1.2 Changeset

The changeset is measured by recording the number of files that are committed together. Files committed together are related, and more likely to be changed together in the future. This can be explained as they share of classes and function calls among others. The change in the behaviour of file **A** can directly affect the behaviour of file B who uses a modified class of A. Therefore, in order to maintain the functionality of B, it will also need to be debugged. The complexity of the code will increase the changeset value since more and more files are connected.

There are some advanced forms of the changeset metric. The first one is time-interval changeset. All files committed within a small time-interval are considered to be related. This can be relevant since multiple commits within 15 minutes in the same branch are likely to be about the same feature. The second one is to also check the committer. Due to specialization in a group project, a person will work better in the area of their speciality, for example, Front End, Back End, DevOps, Infrastructure. Moreover, it is observed that when dealing with a bug in Chromium browser, only one developer is assigned to fix the code, and others will review the solution before pushing to the master branch. When combined with the first one, we can really see the connection of files committed within a time interval, by the same committer.[7, 6]

2.2 Vulnerability Prediction

Naive Bayes

The purpose of code change metrics is to form data set for predictions of vulnerabilities. Compared to other metrics, code change metrics are not only simple to collect, easy to implement, fast but also yield successful results. To further stretch the cost-time-sensitive aspect, simple machine learning classifiers are often applied such as Naive Bayes, Decision Tree or Logistic Regression.

Naive Bayes is based on applying Bayes's theorem with strong(naive) independence assumptions between the features.[8]

- P(c|x) is the posterior probability of class (c, target) given predictor (x, attributes).
- P(c) is the prior probability of class.
- P(x|c) is the likelihood which is the probability of predictor given class.

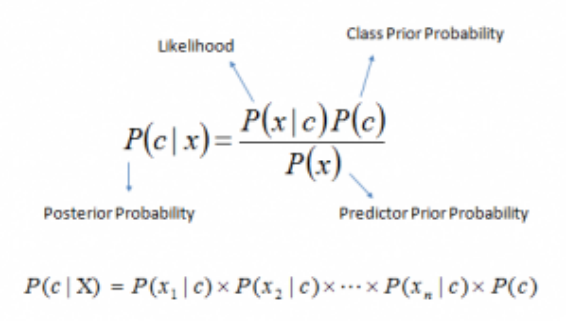

Figure 2.1: Bayes rule

- P(x) is the prior probability of predictor.

Gaussian Bayes Classifier

A training set is used to summarize data about all attributes considered in testing: number of lines added and deleted and number of the changeset. Normally, when dealing with discrete features such as counting number of lines, multinomial and Bernoulli distributions are the best fit. However, since we will also use average lines of code as attributes, a continuous model is more reasonable. As a result, we choose the Gaussian distribution, even though the values do not strictly follow a normal distribution. We will calculate the mean and standard deviation of the distribution related to each attribute. The classifier then takes a testing set as an input. All files in the testing set will be considered to reveal the probability of whether they are more likely to be defective or not.[8]

$$p(x = v \mid C_k) = \frac{1}{\sqrt{2\pi\sigma_k^2}} e^{-\frac{(v-\mu_k)^2}{2\sigma_k^2}}$$

Figure 2.2: Gaussian Distribution

- μ_k is the mean of the values in x which belonged to class C_k.
- θ_k^2 is the variance of the values in x which belonged to class C_k.
- v is the observation value.
- p(x=v | C_k) is the probability distribution of v given a class C_k

Binary classification

We will use several different classification measures to examine the prediction results based on the values of True Positive(TP), True Negative(TN), False Positive(FP) and False Negative(FN). TP is the number of correctly identified defective files. TN is the number of correctly identified non-defective files. FP is the number of non-defective files identified as defective. FN is the number of defective files identified as non-defective.

Precision reflects the percentage of defective files (TP) among the files that are marked as defective (TP+FP) from the prediction.[9]

$$\text{Precision} = \frac{TP}{TP+FP}$$

The recall is usually calculated together with precision in the field of information retrieval. The recall represents the percentage of defective files that are identified (TP) among all defective files (TP+FN).[9]

$$\text{Recall} = \frac{TP}{TP+FN}$$

Precision and recall have always been a trade-off for each other. For example, if we mark all of our testing files as defective, we will have a perfect recall rate of 100%, since all of the real defective files are identified (maximum TP) and FN = 0. However, depending on the defective/non-defective ratio, we can only achieve low precision, since we also mark all of non-defective as defective (maximum FP).

In order to gain precision, we have to increase the standard of picking faulty files. This will consequently decrease the number of files identified as defective. For example, if we set up the highest threshold, only choose one file out of hundreds in the testing set as vulnerable, we would have a precision of 100%, with TP = 1 and FP = 0. However, this would lead to an extremely low recall rate, with only 1 file detected.[10]

As a result, a third value called F1 is calculated as a single measurement that takes both recall and precision into account.

$$\text{F1} = 2 * \frac{Recall*Precision}{Recal+Precision}$$

Instead of being just an average value, the F1 score is a harmonic mean. Therefore, it prevents values from being too low or too high, and encourage the balance of both. For example, a Recall rate of 1, and Precision of near 0 will result in an average of 0.5, but an F1 of 0. Similarly, a Recall rate of 0 and Precision of 1 will result in an F1 of 0. In reality, we would not want any measurement to be 0 but prefer a more balanced outcome. Therefore, by maximizing the value of F1, we can achieve optimal recall and precision values.

Another measurement considered in this project is False alarm (FA). FA is the percentage of falsely identified defective files (FP) among the total non-defective files (TN+FP).

$$\text{False alarm} = \frac{FP}{TN+FP}$$

2.3 Related Work

Mining software repositories for traceability links

Huzefa et al. provided traceability links between not only code files but also documents such as README.md or ChangeLog using **sqminer**. Many similar approaches were mentioned, for example Canfora used an information-retrieval method to index the changed files in the CVS repositories with the textual description of past bug reports in the Bugzilla repository and the CVS commit messages; Sliwerski used a combination of information in the CVS log file and Bugzilla to study fix-inducing changes; Cubranic described a tool, namely Hipikat, to recommend artifacts from the project memory that might be relevant to a task at hand. During their research, sequential-pattern mining was their choice of mining algorithm. It used the frequently occurring subsequences

to uncover planned changes of files that were committed in order. For example, small increments of code changes would typically be implemented to add a new feature, which would be followed by modification in README.md. Additionally, it was common for a bug fix to lead to new other bugs, which were only identified after the fix was applied and tested in the master branch. Therefore, by discovering the ordered pattern of the files changesets, we would predict the set of files that were likely to be modified in the next commit given the initially changed files in the previous commit. Overall, the combined heuristic of CommitterDay outperformed the others, with consistent precision and recall rate of 100% [7]

A New Approach for Predicting Security Vulnerability Severity in Attack Prone Software Using Architecture and Repository Mined Change Metrics

Daniel et al. found a new approach: to quantify the relationship between selected metrics and the severity of vulnerabilities. The purpose of the research was to efficiently allocate security developers expertise to artifacts that were most likely to carry vulnerabilities. The research target the Firefox Web Browser, Apache HTTP Web Server and the MySQL Database Server, all with large source-code similar to Chromium. The severity score was defined according to the Common Vulnerabilities and Exposures (CVE), together with the mined bug reports for the purpose of predicting defects among code. However, due to the inconsistent mapping of vulnerabilities and files among CVE, release notes and Issues Tracking System (ITS), predicting severity for each file was not possible at the time of this research. A new metric called change burst was introduced to represent the amount of code change over a period of time. To control the change burst, the gap and burst size were adjusted. The burst size was the minimum number of changes needed to qualify as a burst. The gap indicated the period of time in days between changes that could be counted as belonging to the same burst. To sum up, Firefox and Apache case studies appeared to have significantly better results. Predictions were less precise for the MySQL case study.[6]

A Comparative analysis of the efficiency of change metrics and static code attributes for defect prediction

Moser et al. used all three popular classifiers: Naive Bayes, Decision Tree and Logistic Regression for defect prediction. A total of 18 code metrics were chosen, including average code churn and average change-set, to examine which models were the best for defect prediction. Three different releases of Eclipse were chosen in the case study. For the first and third releases, change models resulted in significantly higher precision and recall rate and lower false positive rate. The second release observed an exceptional difference in the combined model and the others. In conclusion, the results were proven to be successful with an overall of more than 75% of correctly classified files and a recall rate of more than 80%, with Decision Tree outperforming the other classifiers.[11]

Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities

Shin et al. found that since only a small percentage of the code contains defects, it was crucial for developers to be able to identify the locations of vulnerable codes. It would allow experts to quickly focus on relevant pieces of code, which resulted in fast response time when a bug was found. Code complexity and developer activity were accessed in addition to code change metrics. Code complexity was measured by the number of lines of code, number of functions and their parameters. Regarding developer activity, the hypothesis stated that a centralized developer, with more connection to others, would have a better understanding of securing files and codes. Hence, vulnerabilities were more likely to happen to non-central developers. There were 28 total metrics applied to 34 releases of the Mozilla Firefox browser, with over 2 millions of lines of code. Overall, classical code change metrics such as code churn performed better than developer activity.

Static code metrics vs. process metrics for software fault prediction using Bayesian network learners

Biljana et al. compared the discriminating power of static code metrics and dynamic code metrics for software fault prediction. Developed metrics were examined since previous pieces of research showed inconclusive results about their impacts to defect prediction. 17 models were used on 2 different Naive Bayes Classifier to experiment the research question. Several process metrics were also added for their outstanding discriminating power: age of module and number of changes made to files. An unusual observation was made when a model using only 4 metrics could predict with higher accuracy than a combined model of both static and non-static metrics. In the end, none of the metrics was proven to be substantially better than other. This suggested a further research would be required to clarify between the twos.[12]

An Empirical Model to Predict Security Vulnerabilities using Code Complexity Metrics

Yonghee Shin et al. investigated the use of complexity code metrics to predict the exact position of security bugs. While static code metrics could detect certain patterns of vulnerabilities, complexity code metrics could be useful to identify the defective functions among the codes. McCabe's cyclomatic complexity indicated the number of decision statements plus one. The higher the count was, the more complex of the logic would be. Nesting complexity and path counts were also considered in code complexity. The Mozilla JavaScript Engine was chosen as the case study due to the publicity of code and bug reports. From the acquired results, the overall accuracy was very high. However, a high False Negative rate indicated that a large percentage of defective files/functions was not identified. A large proportion of faulty-but-not-vulnerable functions were detected, which suggested differentiating highly nested functions and other functions for more precise vulnerability prediction.[13]

Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista

Thomas et al. used classical code metrics like code churn, complexity, coverage and dependence measures to predict security vulnerabilities. They also explored the correlation of these metrics and bugs to compare how accurate they are when applied in vulnerability prediction. Dependencies were measured by the number of function calls for both caller and callee, number of exports and imports of classes. A new metric called Organizational measure was implemented, which covered the number of engineers ever worked with a file, frequency of modification in an organization level. Window Vista was chosen for the case study. In their experiment, a result of high precision but a low recall was achieved, meaning not a higher proportion of defective was captured, but any identified one had a high chance of being faulty. Nevertheless, when the dependencies were used as metrics, the classifier predicted vulnerabilities with lower precision but notably better recall rate. As suggested for future work, the functionality of a file could help to improve the prediction ability. However, more data would be required, since software metrics could not differentiate code functionality.[14]

Chapter 3

Vulnerability Classifier

Since most of the previous researches used Firefox as their choice of browser, there was no previously collected data available. Without access to the chromium Database, we had to mine our own dataset. We scraped the available information from the Google Git. Our aim was to mine data between pre-release and post-release of the stable version of the project[15]. Any file involved in a bug post-release would be identified as defective because it was not fixed in the stable release.

The scraping process will be presented in section 3.1. Next, we will evaluate the data in section 3.2 and examine its features. Finally, details about the Classifier will be demonstrated in section 3.3.

3.1 Web scraping

Since all the data was stored in the form of a website, scraping it from the page Html code was the obvious choice. We chose BeautifulSoup[16] as our helper package, as it provided us with useful Html reader and tag extracting functions. We also looked at snippets available on-line: Stackoverflow[17], W3School[18] to help build the data scraper.

The Chrome release[19] held the information about all releases of the Chrome browser. It was simple to scan through the blog and retrieve the URLs to the change log for the needed stable releases. Each change log contained every commit ever made to the project since the last stable release, as shown in Figure 3.1. The commit included information about which bug it belonged too and which files were changed.

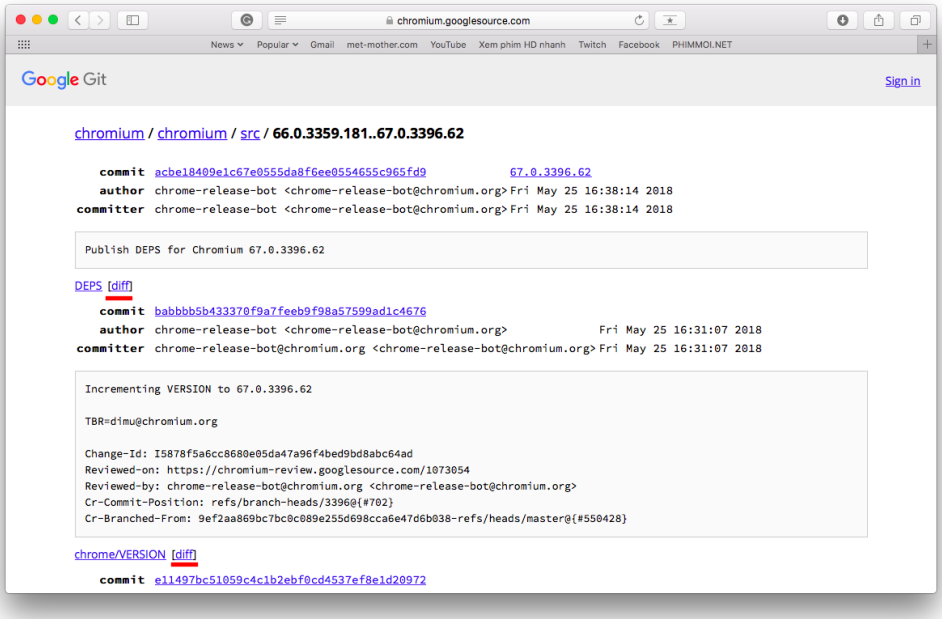

Figure 3.1: Change Log of a release

In order to scrape the data, we first looked at the source code of the page in Html form (Figure 3.5). The commit message is stored in the pre-formatted text tag <**pre**> with class name **u-pre u-monospace MetadataMessage**, where we can extract the bug ID related to the commit. Using BeautifulSoup4, we retrieved a list of lines in the message and were able to check each line for the "bug" occurrence.

```
commits = html.findAll('pre', {'class' : 'u-pre u-monospace MetadataMessage'})
```

Figure 3.2: Extracting commit

For each commit, there was a list of all the files that are modified in order to fix the bug or add new a feature. The files are stored in the unordered list tag <**ul**> with class name **DiffTree**. The extraction of the files was done the same as the above process (Figure 3.3).

```
diffs = html.findAll('ul', {'class' : 'DiffTree'})
```

Figure 3.3: Extracting changed files

After the file list was saved, we needed the code metrics information, which was the number of lines added, deleted and the changed set value. These attributes could be gathered from the **[diff]** links next to each file path (Figure 3.1). We sent a new HTTP request for each file in a commit in

order to capture the added and deleted values between the "@@" symbols. This limited the size of our dataset, as we will discuss later in chapter 3.2.

Up to this point, we had a dataset containing the file path, number of lines added, number of lines deleted, change set of any given releases. However, we could not run a classifier without knowing if any of these files are defective after the next release or not. Choosing the right release was also a problem. Some releases only held around 50 commits, therefore a lot of files were not even checked or mentioned. Marking those files as non-defective would make our predictions pointless. As a result, we chose the latest stable update on May 29th. The fact that it included more than 10000 commits made it qualified since it could be considered as a major security patch for the Chrome browser. Using the list of files acquired from the Git scraping, we could search for their vulnerability status in the May 29th changelog. If they appeared in any commit with a **BugID**, they would be marked as defective.

Figure 3.4: Diff value in the [diff] link

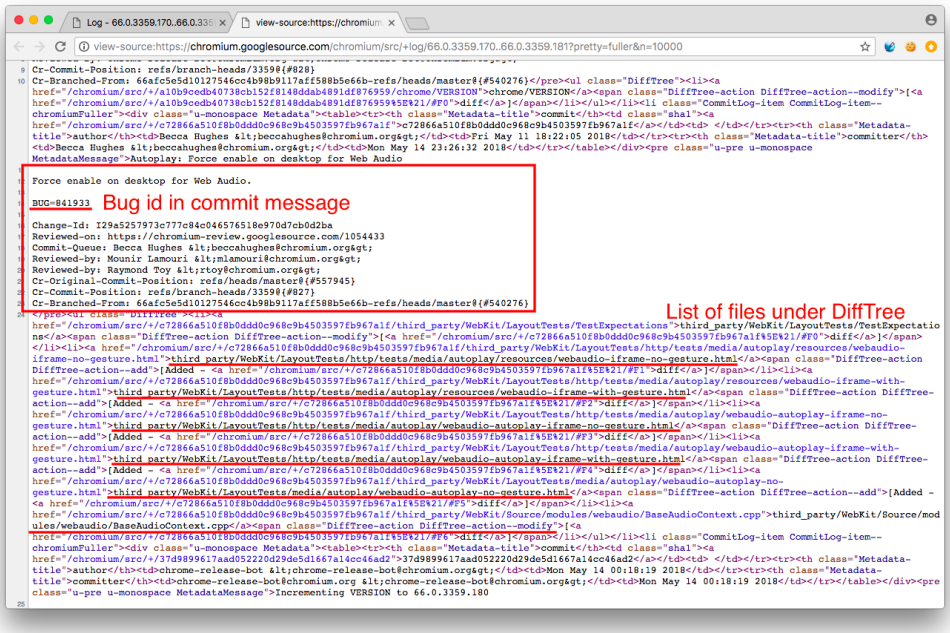

Figure 3.5: Source code of Change log

3.2 Data set

Our dataset consisted of 1757 unique files, collected in 3 different releases prior to the major security patch on May 29th. In those 1757 files, 1197 files were defective. This was an unusual dataset compared to other researches since they were conducted using Firefox releases. Each release had more than 10000 files when only around 100 security bugs were found on CVE - Common Vulnerabilities and Exposures. This contributed to a defective ratio of around 1%. The reason being was that each release was a few months apart and the bugs considered were only severe security-related bugs in CVE. Since the ratio was so low, the values of code change metrics between defective and non-defective appeared to be more distinctive.

Our dataset, on the other hand, considered all bugs fixed between two releases. The period of time between releases was also much shorter, ranging from 1 to 2 weeks. As a result, changes in a short time could indicate more bugs fixing and less feature adding. This explained the more-than-half defective ratio rate. However, we believed that it could better represent the recent changes to the code and be more useful for developers on a day-to-day basis. Files modified in recent smaller releases were more likely to involve in bugs in the near future.

Even though the available metrics were the number of lines added, the number of lines deleted and changeset values, we thought that by adding more metrics we could improve the result of the classifier. Therefore a new value called **involved** was collected, which recorded the number of time a file was involved in a commit. This enabled us to calculate the average added lines, average deleted lines and average changeset, which doubles the number of metrics and provides us with more results to evaluate.

Within the dataset, there were a few notable commits. These commits included more than 100 files each, which was significant compared to the size of our dataset. Hence, there were hundreds of files with the same number of changeset, which could have a great effect on the mean and variation of these attributes. Moreover, one particular commit appeared to be the update in a set of language files. Therefore, not only the changeset but also the number of added and deleted lines were the same on these files. In the beginning, we considered the idea of excluding these commits and marked them as outliers, as suggested when using Gaussian normal distribution. However, we came to realize that this was a feature of non-defective files. If there was a change in feature functionality or new library package, the files that used those features would be changed together in one commit, with the same added and deleted values. On the other hand, bug fixing commits tended to involve fewer files, and the changes in lines of code were more diverse.

Figure 3.6: Example of files in the dataset

3.3 Naive Bayes Classifier

The Classifier was written in Python conveniently to match with the web scraping as well as the web application system. In the process of programming the classifier, snippets regarding machine learning the python available online: Stackoverflow [17], W3School [18] and Machine Learning Mastery also assisted us[20].

3.3.1 Training

The training process of the Naive Bayes classifier was rather straightforward. The first step was to apply the 10-fold cross-validation technique (Figure 3.7) to the dataset. We used the **random** method provided by Python Library to randomly select files into 10 smaller sets. This allowed us to have a completely different set of files each time the Classifier was run. Given that our dataset was small, choosing every single file at random might not face the time-consuming issue. Moreover, by using 10-fold cross validation, we made sure that all files participated in both training and testing, and each file was used as test subject exactly one.

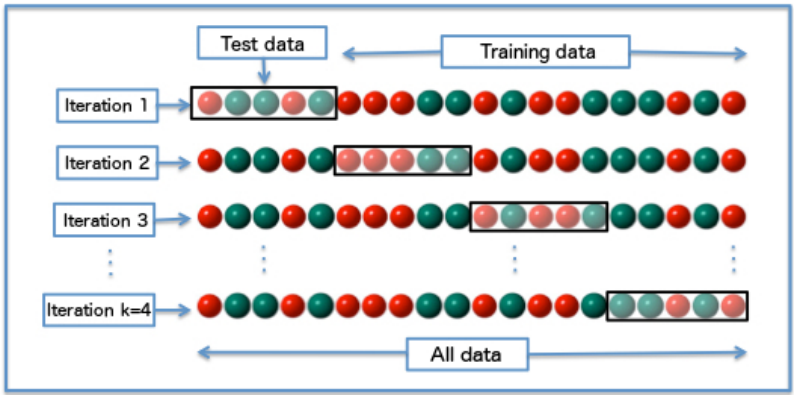

Figure 3.7: K-fold cross validation

For each rotation of the 10-fold, we had to split the training set into defective and non-defective sets. The reason being was so that we could conduct a summary of the mean and variance of each attribute for both classes. The summary would then be used in the testing process to discriminate between defective and non-defective files.

3.3.2 Predicting

In order to predict a file being defective or not, we would calculate the probability of it in each class, and choose whichever was higher. First, for each attribute **k** of a file, we calculated the probability p_k of that value being defective and non-defective, using the means and variances from the summary above. Then, by multiplying all p_k together, we ended up with a final probability of **x** for each class. Comparing these two probabilities would result in the prediction.

3.3.3 Classification

With the predictions of all the files in the testing set, we could examine the statistic features of those values using True Positive, True Negative, False Positive, False Negative. Since we used the 10-fold class validation, we had 10 different values for each of Recall, Precision, False alarm and F1. This would allow us to study the stability of the classifier.

3.4 Results

Using Naive Bayes Classifier, we could check the effectiveness of each metrics individually and different combinations to figure out the most optimal setting. Moreover, as we examined the data, creating a threshold value to control the balance of recall and precision could improve the overall result

Code change metrics

The first setting we tried was the number of lines added as metrics. As shown in Figure 3.8, the classifier had perfect recall rate and false alarm rate of 100% in 8 cases. This could be explained as the tool marked all files as defective (TN, FN = 0) when we debugged the results. Even though a high recall rate was encouraged, the false alarm rate was too high to consider this setting to be useful. It was clear that the number of added lines did not reflect the difference between defective and non-defective files for our data set.

After the above evaluation, We expected that using the number of deleted lines might result similarly. The recall reduced slightly by around 10%. Precision and the F1 score remained unchanged with an average of 72% and 80% respectively. However, the false alarm rate decreased significantly to around 70%. Overall, both classes of defective and non-defective had similar distributions.

As we combined the two metrics, the results were improved. The false alarm was cut by a half, meaning more files are marked as non-defective correctly. However, the rate of Recall also dropped by 20% to an average of 70%. Since Precision did not change, F1 decreased slightly by 10%. To conclude, the combination of both added and deleted were much preferred out of the three models.

Recall	False alarm	Precision	F1
100.0	100.0	72.0	83.72
100.0	100.0	65.14	78.89
95.04	100.0	68.05	79.31
100.0	100.0	64.0	78.05
100.0	100.0	62.29	76.76
100.0	100.0	69.14	81.76
100.0	100.0	68.57	81.36
100.0	100.0	70.29	82.55
95.08	90.57	70.73	81.12
100.0	100.0	70.86	82.94

Figure 3.8: Metrics used: Added only

Recall	False alarm	Precision	F1
89.34	64.15	76.22	82.26
88.1	71.43	76.03	81.62
86.96	68.33	70.92	78.12
86.36	63.08	69.85	77.24
85.83	62.5	78.42	81.95
100.0	100.0	68.0	80.95
100.0	100.0	66.29	79.73
91.6	73.21	72.67	81.04
89.43	71.15	74.83	81.48
90.43	65.0	72.73	80.62

Figure 3.9: Metrics used: Deleted only

Recall	False alarm	Precision	F1
68.46	44.44	81.65	74.48
71.3	43.33	75.93	73.54
65.0	36.36	79.59	71.56
81.82	66.67	73.33	77.34
81.3	38.46	83.33	82.3
56.76	23.44	80.77	66.67
60.5	33.93	79.12	68.57
64.71	37.5	78.57	70.97
65.22	33.33	78.95	71.43
60.17	49.12	71.72	65.44

Figure 3.10: Metrics used: Added and Deleted

Average Code Change metrics

We divided the number of lines changed by the number of commits each file involved in to get the averages.

First, we examined the average number of added lines. Compared to the non-average values, this was a much more practical metric. Although the Recall rate was not high (50%), the Precision value was extremely good. Therefore, any file that was marked as defective would most likely to be defective. This would be a helpful metric for developers to find bugs in their projects. The False alarm was less than 10%, which showed that the number of files identified as non-defective was also highly reliable.

The average number of deleted lines performed slightly worse than the added one. Recall rate fluctuated between 38% and 44%. The False alarm was almost negligible, with none of the cases passing 2%. Precision was also better, with the lowest value being 97%. Nevertheless, the F1 score decreased moderately compared to the previous table.

When combined together, the average changes showed great performance as we expected. More than half of the defective files were identified, with a precision rate of 97%. The balance was also maintained as observed by the average of 70% of the F1 score, highest in the three tables.

Overall, the average number of lines changes outperformed the normal measurements. This could be explained as even though having similar added/deleted number of lines, the difference in the number of involvement in bugs was drastic enough help distinguish between the two classes.

Recall	False alarm	Precision	F1
46.43	3.17	96.3	62.65
42.74	3.92	96.36	59.22
44.83	1.69	98.11	61.54
44.35	3.33	96.23	60.71
41.88	1.72	98.0	58.68
50.45	9.38	90.32	64.74
45.04	0.0	100.0	62.11
52.46	7.55	94.12	67.37
43.55	3.92	96.43	60.0
47.93	1.85	98.31	64.44

Figure 3.11: Metrics used: Average Added

Recall	False alarm	Precision	F1
43.9	1.92	98.18	60.67
38.26	0.0	100.0	55.35
39.68	0.0	100.0	56.82
42.5	1.82	98.08	59.3
41.44	1.56	97.87	58.23
38.33	0.0	100.0	55.42
43.8	1.85	98.15	60.57
42.86	1.59	97.96	59.63
38.33	1.82	97.87	55.09
40.0	2.0	98.04	56.82

Figure 3.12: Metrics used: Average Deleted

Recall	False alarm	Precision	F1
54.7	0.0	100.0	70.72
60.34	3.39	97.22	74.47
48.72	1.72	98.28	65.14
53.72	0.0	100.0	69.89
49.02	6.85	90.91	63.69
45.53	1.92	98.25	62.22
54.92	3.77	97.1	70.16
54.13	3.03	96.72	69.41
51.54	4.44	97.1	67.34
53.28	2.63	98.65	69.19

Figure 3.13: Metrics used: Average Added and Deleted

Changeset

Changeset appeared to have a perfect result summary. Recall rate was always at 100%, with nearly no False alarm. Together with a high precision rate, the F1 score was approximately 99%. As explained in Chapter 3.2, a lot of non-defective files were committed together. Therefore, they boosted the mean and standard deviation of the distribution to significantly higher values, which made it easier for the classifier to identify defective files. For example, a bug-related commit only included around 5 to 20 files. However, some non-defective commits had hundreds of files, which made the mean of the non-defective distribution to rocket to around 100. These accounted for most of our non-defective files, therefore the predicting process was even more accurate. The results also supported one of our expectation: files that involved in bugs in the past releases had a higher probability of participating in a future vulnerability compared to the files in a feature-adding commit.

Recall	False alarm	Precision	F1
100.0	3.7	98.37	99.18
100.0	5.66	97.6	98.79
100.0	5.77	97.62	98.8
100.0	6.9	96.69	98.32
100.0	0.0	100.0	100.0
100.0	0.0	100.0	100.0
100.0	3.28	98.28	99.13
100.0	9.68	94.96	97.41
100.0	0.0	100.0	100.0
100.0	5.45	97.56	98.77

Figure 3.14: Metrics used: Change set

Combined metrics

We used both the code metrics and changeset to predict the defective files. Since the results of the number of lines added and deleted metrics were impractical in files discrimination, the changeset appeared to have a strong influence in the combination. Recall rate and precision were around 100%, with extremely low false alarm rate. This was easily foreseen when 2 indecisive metrics were used together with a key metric.

The same effect happened to the combination of average code changes metrics and changeset. However, the recall and F1 score were not quite as absolute as the previous one. It showed that even though changeset was a strong indicator of defectiveness, average metrics could still in some occasions affect the prediction.

Recall	False alarm	Precision	F1
100.0	1.72	99.15	99.57
99.17	3.7	98.36	98.77
100.0	8.93	95.97	97.94
100.0	3.7	98.37	99.18
100.0	1.82	99.17	99.59
100.0	3.39	98.31	99.15
100.0	3.85	98.4	99.19
100.0	7.02	96.72	98.33
100.0	1.75	99.16	99.58
100.0	5.36	97.54	98.76

Figure 3.15: Metrics used: Added, Deleted, Change set

Recall	False alarm	Precision	F1
100.0	4.92	97.44	98.7
95.9	1.89	99.15	97.5
98.37	5.77	97.58	97.98
100.0	6.0	97.66	98.81
93.86	4.92	97.27	95.54
91.74	1.85	99.11	95.28
98.31	3.51	98.31	98.31
96.77	1.96	99.17	97.96
97.44	6.9	96.61	97.02
98.23	3.23	98.23	98.23

Figure 3.16: Metrics used: Average Added, Deleted and Change set

Chapter 4

Web application

I built the tool based on the Model View Control architecture to utilize the modularity of each application logic part and ease of collaboration. The design of the application was based on the Django Web application tutorial[21], and other snippets available on Stackoverflow[17] and W3School[18]. I also followed tutorial on Python3[22] documentation and PostgreSQL[23] documentation.

4.1 Model View Control

The disconnection of internal parts not only allowed me to separate our concerns for each component but also made parallel progressing possible. I was able to work at front end and back end at the same time, without worrying that a bug in the logic of the back end might crash the whole tool. Moreover, by using a distributed Database Server (Postgres), I could keep adding new observations as input for the Classifier without modifying the source code. This enabled the application to scale, as the cost of transferring data between the components were negligible.

Model

I used a Postgres Database server to store information of Files, Bugs and the Releases. Django provided an efficient way to manage Database by considering them as different objects. Each object consisted of all the information about them, together with the link to other object types. For example, File-Release was set as a Many-To-One relationship. As a result, I could differentiate the same file path across different releases. Information such as repeated bug involvements was also revealed by storing data in this setup.

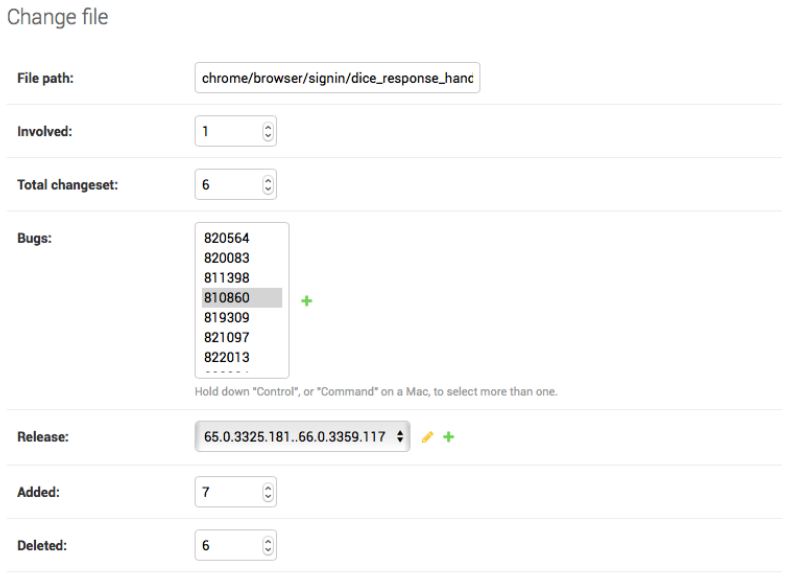

Figure 4.1: A file stored in the Database

```python
class Bug(models.Model):
    bug_id = models.CharField(max_length=10)
    #to String
    def __str__(self):
        return self.bug_id

class Release(models.Model):
    release_number = models.CharField(max_length=200)
    #to String
    def __str__(self):
        return self.release_number

class File(models.Model):
    file_path =models.CharField(max_length=200)
    involved = models.IntegerField(default=0)
    total_changeset = models.IntegerField(default=0)
    bugs = models.ManyToManyField(Bug)
    release = models.ForeignKey(Release, on_delete=models.CASCADE)
    added = models.IntegerField(default=0)
    deleted = models.IntegerField(default=0)
    #to String
    def __str__(self):
        return self.file_path
```

Figure 4.2: Models in Django

View

The front end was programmed using HTML. It printed out the result of the Classifier, and also contained the link that allowed users to control the database. When a button was clicked, it would trigger the script in the **views.py** file, and request a query to the Database Server. The Database server would then return the list of files needed for the logic.

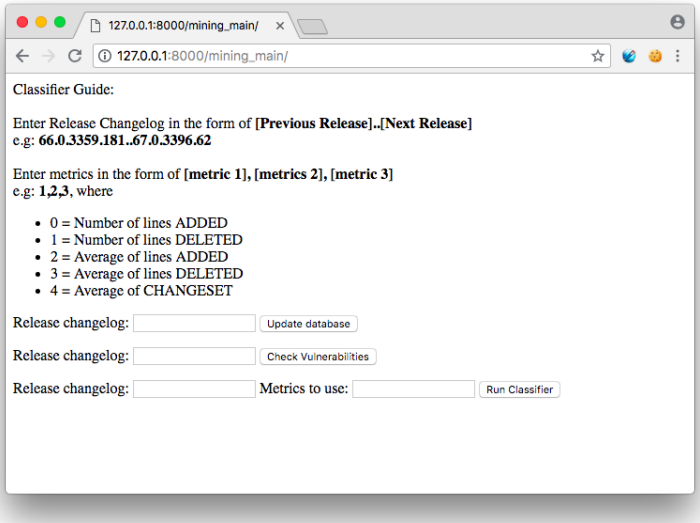

Figure 4.3: Homepage

Control

The back end of the application was also written in Python, which included the Logic of how the front end would operate and the Classifier. For example, if the index page was requested, first the list of files would be requested from the Database, ordered in the number of lines added in descending. Then an Html template would be fetched, and loaded to the front end together with the list of files.

Chapter 5

Evaluation

With the objectives stated in the introduction, we will first consider how practical the tool can be when being used in different cases. This can be subjective as we do not have the opportunity to present the application to real Chrome developers to test out. Once all the uses of the tool have been stated, we can have a comparison of our work to other similar researches, what we have done better, and what we have done worse. Finally, we will list what can be changed to improve the Classifier performance.

5.1 Practical

At the time of this project, a reliable tool for predicting vulnerabilities did not exist for the Chrome browser, since most bugs were spontaneous. There was no accurate way to predict when a piece of code would be faulty in a specific file. However, if we could reduce the range of files that could be defective, and focused on debugging those files, it could help to save a lot of time as well as manpower. This was exactly what we achieved in this project.

As the Chromium project grows, a new commit is added every few seconds to the repository. Our tool is an efficient way to monitor newly modified files and see if they contain faulty code. We choose a data set of 3 most recent stable releases as our training set, which includes commits from up to a month ago. From the acquired results, it is shown that the tool, with the correct set of metrics, can produce prediction with high recall rate and precision. Therefore, we have succeeded in proving a correlation between code metrics, changeset and vulnerabilities in the Chromium Project.

We would like to demonstrate how a user can use the classifier to help to improve his work.

User story

1. Developer Albert receives a bug report from the Bug Release.
2. He narrows down a list of files that might contain the faulty code in the repository.
3. Instead of debugging right away, follow function calls between files to trace down the bug, he uses the Classifier.
4. Using the list of files he produced as the testing set, what he needs now is to pick a training set
5. A training set can be a few recent releases in the last month, and up to the whole repository git commit-tree from the beginning. The size of the training set depends on how severe the vulnerability is: whether it is a minor bug and he wants a quick check, or a thorough scan is required to identify any relevant files.
6. As he runs the Classifier, he acquires the sublist of files predicted to be defective. He can start debugging a smaller list of files with high probability to contain a vulnerability.
7. Overall, the classifier helps reduce his debugging time, and drives his focus into files that have the highest chance of being faulty

5.2 Performance

We will compare the performance of our classifier to previously built tools by other researches. Even though Firefox was their browser of choice, both Firefox and Chromium are open-source browsers. Therefore they can have similar bugs, with the same severity and code changes when fixed. Moreover, when a security bug is found to a browser, hackers tend to try to apply it to other browsers to exploit it before it is fixed.

The average recall rate of Firefox classifier is 85% [5]. Our tool measurement is 55% when we choose the average number of added lines and deleted lines as metrics. It is clear that our Classifier does not perform as well as we want when it comes to code change metrics. However, when comparing the changeset metric value, we outperform the other tool mentioned in [11]. Our average recall rate is 99% compared to 80%. As explained in section 3.2, our dataset is much smaller than the Firefox case (1757 - $\tilde{1}$0000). In their 10000 files, only 100 files are defective, when we have more than 1000. The drastic difference in defective ratio means it is much harder for their classifier to achieve a high value in precision and recall compared to ours.

5.3 Future Improvement

Dataset

As mentioned many times before, we can further test our classifier using a more balanced and larger dataset. When compared to the size of the repository, the real defective rate is much smaller, since a lot of files are only changed once a year, for example, to update to a newer library package. Therefore only taking recent releases might not be enough to track down all possible faulty files.

The reason we did not scale up our data set is that it was quite time-consuming and the occurrence of high change set commits. In order to extract the code change value, we had to send a request to a new URL and read from the result Html. Imagine there are 400 files in one commit, it would take around 20 minutes for that one single commit to go through. There are approximately 10000 commits in the big security patch in April, which might take a few days to run. There is a major security release every month, and considering Chromium has been developed for 9 years, it is possible to estimate the size of the total possible dataset. For future improvement, we can work on collecting data more efficiently.

We can also scale down the dataset to only includes bugs that have the label **security** as other researches. This indicates that we can focus on defending against hackers, fixing vulnerabilities before they are being exploited. Moreover, security-related bugs appear less frequently than others and their code metrics are more distinct. According to Chromium Bug Blog [24], there are $\tilde{7}00$ out of 12000 bugs has the label **security**.

Distribution of metrics

Using the Gaussian distribution might not be the best fit to represent the metrics we use for the classifier. The number of lines changes and the number of changesets does not follow a normal distribution. We can consider Multinomial or Bernoulli distributions or a combination of both for all the discrete values, and use the Gaussian for the calculated average metrics. It may be a good idea to allow users to select the type of distributions they want and compare performances across the same dataset to find the most suitable distribution that matches their purposes.

Threshold value

The threshold value ranging from 0 to 1 can act as a controller to balance between recall and precision. The higher the threshold is, the stricter the standard for predicting a file as defective will be. This means higher precision since we only pick the files with the highest possibilities. However, as a result, recall rate will drop, since fewer files will be identified as faulty. Depending on the purpose of the user, the threshold value can be useful, whether precision or recall is more relevant to the situation

Chapter 6

Conclusion

In the project, we have successfully built a complete Naive Bayes Classifier for the Chrome browser. The application of The code metrics and changeset metric has been proven to be practical in defect prediction. However, their impacts on the discriminating power were highly different as expected from the start. The changeset metric outperformed the code metrics in both precision and recall rate. This indicated that there was a drastic contrast in the number of changesets of defective and non-defective files. The average values of code metrics resulted in better prediction, which showed promising performance when presented with a larger dataset.

Nonetheless, even though we have foreseen some challenges we might face, the problem with dataset remained unsolved. Therefore, we were not able to test the true discrimination power of the classifier when given a more diverse dataset. I also was not able to host our classifier on-line, which was one of the purposes of building a web application in the Model View Control architecture.

Being one of the few vulnerabilities predicting tools for the Chrome browser, I hope this project could be the start of a more accurate and efficient application that can have an actual industrial use. As Chrome continues to dominate the browser market share, a small contribution toward a more secure browsing experience of users may become useful in the future.

Bibliography

[1] Sung-Whan Woo, Omar H. Alhazmi, and Yashwant K. Malaiya. An analysis of the vulnerability discovery process in web browsers. 2006.

[2] F. Massacci and V. H. Nguyen. An empirical methodology to evaluate vulnerability discovery models. *IEEE Transactions on Software Engineering*, 40(12):1147–1162, Dec 2014.

[3] N. Medeiros, N. Ivaki, P. Costa, and M. Vieira. Software metrics as indicators of security vulnerabilities. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 216–227, Oct 2017.

[4] Robert M. Bell, Thomas Ostrand, and Elaine Weyuker. Does measuring code change improve fault prediction? page 2, 09 2011.

[5] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6):772–787, Nov 2011.

[6] Daniel D. Hein. *A New Approach for Predicting Security Vulnerability Severity in Attack Prone Software Using Architecture and Repository Mined Change Metrics*. PhD thesis, 2017.

[7] Huzefa H. Kagdi, Jonathan I. Maletic, and Bonita Sharif. Mining software repositories for traceability links. *15th IEEE International Conference on Program Comprehension (ICPC '07)*, pages 145–154, 2007.

[8] Wikipedia contributors. Naive bayes classifier — Wikipedia, the free encyclopedia, 2018. [Online; accessed 12-June-2018].

[9] Wikipedia contributors. Precision and recall — Wikipedia, the free encyclopedia, 2018. [Online; accessed 13-June-2018].

[10] William Koehrsen. Beyond accuracy: Precision and recall choosing the right metrics for classification tasks, 2018. [Online; accessed 13-June-2018].

[11] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction, 01 2008.

[12] Biljana Stanić. Static code metrics vs. process metrics for software fault prediction using bayesian network learners. Master's thesis, Mälardalen University, Sweden, 2003.

[13] Yonghee Shin and Laurie Williams. An empirical model to predict security vulnerabilities using code complexity metrics. pages 315–317, 01 2008.

[14] T. Zimmermann, N. Nagappan, and L. Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 421–428, April 2010.

[15] Lile Hattori, Gilson dos Santos Jr, Fernando Cardoso, and Marcus Sampaio. Mining software repositories for software change impact analysis: A case study. In *Proceedings of the 23rd Brazilian Symposium on Databases*, SBBD '08, pages 210–223, Porto Alegre, Brazil, Brazil, 2008. Sociedade Brasileira de Computação.

[16] BeautifulSoup team. Beautiful soup documentation, 2018. [Online; accessed 13-June-2018].

[17] Various contributors. Stackoverflow, 2018. [Online; accessed 13-June-2018].

[18] Various contributors. W3school, 2018. [Online; accessed 13-June-2018].

[19] Google Chrome team. Chrome releases, 2018. [Online; accessed 13-June-2018].

[20] Jason Brownlee. Machine learning mastery, 2018. [Online; accessed 13-June-2018].

[21] Django team. Writing your first django app, 2018. [Online; accessed 14-June-2018].

[22] Python team. Python 3.6.6rc1 documentation, 2018. [Online; accessed 14-June-2018].

[23] PostgreSQL team. Postgresql 10.4 documentation, 2018. [Online; accessed 14-June-2018].

[24] Chromium team. Chromium bugs blog, 2018. [Online; accessed 15-June-2018].