UNIVERSITY OF GOTHENBURG

Predicting software vulnerabilities using topic extraction

An investigation into the relation between LDA topic models, and ability of machine learning to predict software vulnerabilities

Bachelor of Science Thesis in the Software Engineering and Management Programme

SAIMONAS SILEIKIS

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, June 2016

Predicting software vulnerabilities using topic extraction
An investigation into the relation between LDA topic models, and ability of machine learning to predict software vulnerabilities

Saimonas Sileikis

Examiner: Jan-Philipp Steghöfer

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2016

Predicting software vulnerabilities using topic modeling

Using extracted topics from source code using LDA algorithm (latent Dirichlet allocation) as features for machine learning, to predict vulnerable files in source code

Bachelor of Science Thesis in the Software Engineering and Management Programme

SAIMONAS SILEIKIS

Predicting software vulnerabilities using topic extraction
An investigation into the relation between LDA topic models, and ability of machine learning to predict software vulnerabilities

Saimonas Sileikis

Examiner: Jan-Philipp Steghöfer

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2016

CONTENTS

Abstract*—A vulnerability database for a large C++ program was used to mark source code files responsible for the vulnerability either as clean or vulnerable. The whole source code was used with latent Dirchlet allocation (LDA) to extract hidden topics from it. Each file was given a topic distribution probability, as well as the status of being either clean or vulnerable. This data was used to train machine learning algorithm to detect vulnerable source files, based only on their topic distribution. In total, three different sets of data were prepared from the original source code with varying number of topics, number of documents, and iterations of LDA performed. None of data sets showed ability to predict software vulnerability based on LDA and machine learning.

I. Introduction

The scope and complexity of software projects are constantly expanding. The greater size of complexity slows down or even makes it impossible for a reasonable amount to completely understand what is happening, and see the big picture. These hurdles contribute to the fact that the big software projects happen not to fit in budget and time frames, resulting in either a late delivery of the product, or rushing the product and thus increasing the chance of defects. Some of the defects can manifest as a software vulnerabilities that might allow attackers to exploit them for personal gain, which usually results in a loss for either the product owners or the product users.

Dealing with software vulnerabilities is harder than usual software defects due to the fact that attackers are disincentivized to report these vulnerabilities. The attackers can either exploit the vulnerabilities themselves or sell them to a third party [1], whom could exploit the vulnerability for a greater effect. To combat this a number of companies have implement vulnerability reward programs [2]. These programs specify a protocol for reporting vulnerabilities, and researchers who disclose new-found vulnerabilities according to this protocol are rewarded. Increasing the amount of people working to find vulnerabilities, increases the likelihood of them being found, at the disadvantage of requiring compensation for these people. Some companies have spent over $500,000 in their reward programs [2]

Another alternative for employing more people to search for vulnerabilities in a company's software is to employ machines to do that. Automation of discovery of software vulnerabilities has a two-fold advantage. First of all, it can be done faster, as there is no need to wait for a person to stumble upon a vulnerability, and then decide if to ignore, to report or, in worst case, to exploit it or sell it. Secondly, it would allow to fix the vulnerabilities before they are deployed to the general public, thus saving money in regards to vulnerability reward programs. Although automated vulnerability search is unlikely to replace contemporary methods, it would be a suitable supplement to them.

A. Contribution

There are multiple methods of automatic code assessing to find defects [3]. Although there is previous research into the relationship between software concepts, and defects [4][5][6] majority of them place the emphasis on traditional software defects, while the aim of this paper is to investigate the subset of software defects - vulnerabilities. Another mark to distinguish this paper from others is to pair topic extraction with machine learning, using the topics as features describing a source code file.

If there is a statistically significant link between the topics extracted using latent Dirichlet allocation and a machine learning algorithm to detect vulnerability, it could be used as a justification for further research in this area. If topic extraction can be used to detect software vulnerabilities reliably, that would imply that it is possible to automate such a task. The implication could be huge, from saving developer time looking for vulnerabilities in peer code, or enhancing the software vulnerability detection for experts using it as an aid. It could result in more vulnerabilities being found at a faster rate, before they are exploited or sold to a third party. This could prevent losses that would elsewise incur if such vulnerabilities remained undetected

B. Research question

RQ: Research question: can a machine learning algorithm be trained to reliably detect files associated with software vulnerabilities, using hidden topics discovered by latent Dirichlet allocation (LDA) topic extraction model. Where reliable means, that the **precision** and **recall** (described in section IV)

II. Terminology

Bag - An unordered collection of objects, where duplicate objects are allowed
$Classifier$ - A machine learning algorithm used to classify items into classes i.e. groups $Corpus$ - A collection of documents
$Document$ - A mixture of different topics that contains words
LDA - Latent Dirichlet allocation. An algorithm that is used to extract topics from a source.
$Mean$ - An average, where all the items are summed up and divided by the total number of items
$Topic$ - A list of words and their probability to occur in that topic
$Tokenization$ - Act of separating a single block of text into individual items that are commonly referred as "tokens" $Word$ - A element of a document that is has a certain probability to occur in topics. A word can occur in any topic but has different probabilities to occur in different topics.

III. Background

A. Latent Dirichlet allocation

Latent Dirichlet allocation is an algorithm used to extract hidden (latent) topics from a document. Paper about LDA was first published in 2003 by Blei et al. [7], which describes the algorithm. Remarkably, the same model was also published

by Pritchard et al. in 2000 [8]. Both of these were done independently.

The algorithm assumes that the document was created in a further described fashion. There are n pre-determined topics. When document is created, a distribution over topic is chosen, that is, the document will have a certain amount of topic A and certain amount of topic B, and so on for each topic. Once a topic distribution is known, a topic is chosen at random according the topic distribution in the document (more popular topics have a greater chance to be chosen). When topic is chosen, a word is chosen at random based on the word distribution in that topic (more relevant words within the topic have greater chance to be chosen). After the word is determined, a new topic, and thus a new word is chosen. This process is continued until the whole document is generated. Of course, no documents are written in such manner, and this is merely a abstraction that reduces the complexity of a natural language and gives some resemblance of an underlying structure, to which the algorithm can cling to

Even if word are recognizable to a reader, the document as a whole does not have any coherent meaning. Despite these problems, an ordinary human reader could still distinguish what is the topic distribution purely on the choice of words. As an example, a document filled with words as "tree", "root", "leaf", "petal", "fruit", and "summer", "warm", "harvest", "cut", would not make much sense, but it would be a sensible guess that it is a document about farming and gardening. Therefore, even a "bag of words" model is enough to reveal the hidden topic distribution. This model uses the topics to create document with words. Such model is desired, because it is possible to reverse and use a document with words to create a list of topics.

A simplistic explanation of algorithm is described in this paragraph. The algorithm is given a number of topics, as well as a list of documents, both of which are supplied by the user. Each word in the whole corpus is randomly assigned a topic. Once every word has a random topic assigned to it, the algorithm tries to infer the hidden topics. This is done by choosing an individual instance of a word from a single document, removing the associated topic from that individual word, and guessing which topic the word should actually be assigned to. This is done mainly by two values: how well the word fits the document and how well the word fits the topic. As an example, if a $wordX$ is found in $documentD$, and the $documentD$ has many words from $topicA$, and very few words from $topicB$, then the $wordX$ has much higher probability to be from $topicA$, showing how well the word fits in the document. On the other hand, if we look at $topicB$ and see that $topicB$ has many instances of $wordX$ occurring in it, and $topicA$ has very few instances of $wordX$, that would mean that $wordX$ fits $topicA$ well, and thus is more likely to be from $topicA$. The final decision as to which topic the word belongs to is a sum of both values.

The above process is repeated for every word in the corpus. Once that is done, an iteration is complete. It is desirable to have multiple iterations to arrive at less random values. Even though the initial distribution is completely random, multiple iterations, words that occur in same topics, and same documents start to congregate and stick to their own topics with greater chance. This is possible because the initial topics are functionally meaningless and just proverbial houses that words can occupy, and a lot of words change their initial topic very fast. To reiterate, each topic has a chance for any word to be in it, and therefore each word has a chance to belong to any topic, despite that those chances could be extremely low. That is the probabilistic nature of LDA.

B. Machine learning

The data obtained from LDA will be used in machine learning to investigate the relationship between that data and ability to find vulnerabilities based on it alone. Machine learning is a method of automatic data analysis. It allows to efficiently process high quantities of data, as well as to provide a deeper insight, that is much harder to come up with manually [9]. It can be used for sorting e-mails into categories (e.g. spam, social), in marketing to predict customer groups and relevant stimuli for those groups, or to analyze trends in social networks [10] [11]. Machine learning can be generally divided into two categories - supervised and unsupervised [12]. Supervised learning means that the data provided to the machine learning algorithm is labeled, that the algorithm knows what it must find, and it only needs to find a way how to find that relationship. The other alternative is using unlabeled data, which is used in unsupervised machine learning. In that case, the algorithm does not know what it must find, and must infer the information by itself [13].

C. Related research

There has been research about using software metrics and text mining as indicators for vulnerable components [14]. A more general search, for software defects in general, was done by Gupta et al. in 2015, where they used software metrics as well [3]. Another research team [4] did an investigation of the localization of software defects, using static code analysis, with the help of LDA. Other group has researched LDA as a tool for modeling topics to enhance developer understanding of software and ease maintenance [6]. Similar goals can be seen in [5] where the researchers tried to use LDA as a tool to visualize topics from source code, as well as to find related files, based on their topic distribution. LDA has also been used to analyze Common Vulnerability and Exposures (CVE) reports to find most popular vulnerabilities, as well as to help to identify trends in new vulnerabilities [15]. This is one of few examples of LDA used for security and not just bug prediction. Despite it being used for natural language, and not source code, it still had positive results, and this gives hope for a link between source code and vulnerabilities. Similar approach has been used by [16] to use National Vulnerability Database (NVD) as input for topic modeling, and use topic distribution to evaluate other software, thus using LDA as a tool of quantitative security risk assessment. This is the most

similar research that has been conducting, in relation to this paper, becausea vunerability database was used in order to evaluate software. The evaluation was performed by searching for vulnerabilities in files, same as it is done in this experiment.

IV. Method

A. Preparation for topic modeling

Chromium source code for version 8.552.215 was obtained, totaling 26 576 files. In addition, a database in form of comma separated values that contains vulnerability IDs, as well as the physical location of a file that is responsible for that vulnerability. The total number of vulnerabilities in the set was 1488. Due to the fact that the database was compiled from multiple versions of Chromium, some of the files identified in the vulnerability database were not present in the version that was used. The final number of files that are were responsible for vulnerabilities was 897.

GibbsLDA++ was used to extract hidden topics from Chromium source code. GibbsLDA++ is licensed under GNU General Public License and is available for public use. The required input for GibbsLDA++ is a single file, which contains all documents that are to be analyzed and used for extraction. Each document should have an identifier, as well as a list of tokens associated with that document. For the purpose of this paper, each source file was treated as a single document, and the path to that file was treated as the identifier.

The tokenization of source code was done by a Python script, which was written solely for the purpose of this paper. Tokenization script not only separates the text into individual words, but also performs text ”cleaning” functions, such as removing white space characters (tabs, spaces, new lines, etc.), as well as various commonly used programming language symbols (exclamation marks, parentheses, brackets, ampersands, slashes, asterisks, etc). Final result is compiled into a single text file, where the first line is the total number of document the whole corpus contains, and each line after that is a document with its respective tokens. That is the required format of input data for GibbsLDA++.

B. Topic modeling

The corpus is every single file in the source code. The total size of the corpus is over 26 thousand files. These files are compiled into a single text document of large size. This document is a single file, and takes up over 160 megabytes of space. The way that LDA works means that each word in a document will increase the time needed to produce a topic model. This, coupled with the size of the of corpus, means that LDA computation takes a long time to complete. This means that mistakes in the experiment design can result in a significant amount of time spent that did not produce anything of value. To avoid such problem, initial experiment - preparing data, processing data with LDA, preparing the topic models from LDA to be used in machine learning, and finally using machine learning was done with a smaller scope. Even if this smaller scope preliminary experiment is suboptimal and might not produce statistically significant result, it allowed to detect problems with the design earlier on, without requiring a greater amount of effort invested. Since LDA requires the user to input the amount of topics to discover, as well as desired number of iterations to perform, 25 topics and 200 iterations were given. That means the LDA assumes that the document was generated using 25 topics, and now those topics are hidden in the corpus and must be discovered. Since the number 200 was given for the number of iteration, the process described in the III subsection **Latent Dirichlet allocation** was performed 200 times before the final topic distribution is calculated. Lower amount of topics and iterations resulted in a faster computational time.

The first actual run followed, after the experiment design was shown to be feasible. The first run was supposed to be the main point of analysis, thus the default number of topics - 100, was used. This was done because lower amount of topics might produce less reliable topic models, due to words being able to be categorized into a smaller number of topics. On the other hand, having a large number of topics has two disadvantages. First disadvantage is that the computational time increases with each topic, because each word has an associated probability with each topics. The second disadvantage is that with large number of topics, more topic remain mostly empty. Thus, the default value of 100 was chosen. Even though this being the focus of the experiment, 500 iterations were performed. The default amount of iterations for GibbsLDA++ is 2 000. This number was not used, because the aforementioned 500 iterations took about two weeks of uninterrupted compilation time, with a minimal amount of background processes running. Because of diminishing returns nature of LDA in regards to number of iterations, the number of iterations was compromised to 500.

To compensate for a lower than desired number of iterations in the first run, a second run was designed. This time the number of topics was kept the same, to keep the results approximately the same, but the number of iteration was increased to the desired 2 000. To allow a topic model extraction with such a number of topics to be completed in a reasonable time frame, a compromise in other attributes had to be made. A smaller number of files were used. The files used came from a stratified sample. This sample of the source code, which is over 26 000 files, kept the non-vulnerable to vulnerable file ratio. The stratified sample contained 2580 clean files, that were not responsible for vulnerabilities, and 89 vulnerable files, that were responsible for a vulnerability. This results in an approximately 90% smaller sample, while still being faithful to the original ratios between clean and vulnerable files, but with an added benefit of having an advantage of shorter computation times for machine learning, and even more so for LDA processing.

Once a distribution of topics over documents is produced by LDA, the last step is to mark each document as either responsible for vulnerability or not. This was done using a Python script that was written for the purpose of this paper.

The script simply reads each line of data that was inputted into LDA, that is, a source code file location on the hard drive and associated tokenized code, as well as reading outputted topic distribution for that file. The associated file is looked up in the vulnerable database, and if it is found, that file is marked as responsible for a vulnerability. If a file is not found in the vulnerability database, then it is marked as not responsible for the vulnerability. Regardless of the outcome, the topic distribution and the status of being responsible for a vulnerability is written to a third text file, which will later be used for machine learning part of the paper.

C. Machine learning

Weka, version number 3.8, a GNU Public Licensed third-party software, was used to for machine learning aspect of the paper. Weka allows to import a data set, use a variety of native functions to format and sanitize the data, and provides a wide variety of machine learning algorithms. Different algorithms have various requirement for the form of the data set. As an example, an amount of more popular algorithms require the data to be nominal. That is, data cannot be numeric, with infinite number of values between 0 and 1, and has to be confined in a specific number of categories, referred to as ”bins”. To take advantage of multiple machine learning algorithms, the distribution of topics among the documents had to be discretized, that is - grouped to into 200 bins, of the same size. Such a high number of bins was chosen because the word-over-topic probabilities range from very low, to very high, and to avoid having too many items in one bin, a higher number was chosen. The use of equal-frequency bins, that is, each bin should have about the same amount of items in it, was not used, since such grouping would give more weight to words with lower frequencies, and topics should be influenced more by high frequency, more topical words, rather than less often used ones.

The main motivation behind these choices were the accuracy rankings between algorithms [17]. Fernandez et al. 2014 [17] performed a comparison of 179 classifiers, using 121 data sets, to evaluate the performance of each classifier in regards to all the others. After removing duplicates, that is classifiers that were tested more than once, because of having been implemented in various environments (C, MatLab, R, etc), best performing classifier algorithms were found. Out of these classifiers some were not available in Weka version 3.8 (the version used for this experiment), and others were removed from not being capable to process the data that LDA has produced. Another qualifier for the method was complexity, meaning how fast it arrives at the result. Having over 26 000 word-per-topic probabilities, with 100 topics, disqualified classifiers that were too slow with large amounts of data. Where too slow means, that it did not arrive at the result after 2 hours. The following machine learning algorithms were used in the end: J48, RandomForest, RandomTree, REPTree, DecisionTable, BayesNet, NaiveBayes. All of these algorithms the default settings, and not parameters, were changed from how they are initially set in Weka 3.8

It was possible to find and tailor the best method for each algorithm but to avoid researcher bias and environment variation, same training and testing option was used for each algorithm. The method for training, as well as testing the validity of the algorithm is cross-validation, using 10 folds. This means that the dataset (the total output from LDA) is separated into 10 groups with the same number of files in them, called folds. Then, 90% of those folds, 9 in this case, will be used to train the classifier to create an algorithm to arrive at the correct decision. The last 10% are used to validate the algorithm that machine learned, that is, the algorithm obtained from the previous 90% will be used on the last 10%, and whether it arrives at the correct decision, will determine how accurate the algorithm is. Once that is done, a new unique mix of 90% of dataset will be used to create an algorithm, which will be tested on the remaining 10%. This will be done 10 times in total, and the results from each iteration will be summed up. The benefit of such method of training and testing is reduced variance between algorithms, due to the fact that each algorithm is tested using same dataset. Another benefit is that, outliers in the data set do not skew the performance as much, because they are used only for 10% of evaluation. If cross-validation with 10 folds was not used, and a part of a document that contains majority of outliers was employed for validation, it would produce significantly different results because the algorithm was trained using normal values from the data set, but it was evaluated on the basis of outliers.

D. Performance indicators

To evaluate how reliable topic modeling is as an attribute provider for machine learning algorithm that tries to detect software vulnerabilities, thus to answer the research question **RQ**, following metrics will be used. True positive, false positive, true negative, false negative, precision, and recall. The metrics indicate how well the algorithm classifies files as vulnerable. Positive result, means that the machine learning algorithm identified the file as being positive, in regards to it belonging to a vulnerable file class. On the other hand, if the result is negative, it means that the algorithm classified the file as not belonging to the vulnerable file class. In other words, positive means that the algorithm thinks that the file is vulnerable, and negative means that it thinks that the file is not vulnerable. This does not mean that the classifier is correct, and ”true or false” prefixes are used. **True positive** means that a vulnerable file was correctly identified as vulnerable. **False positive** means that a non-vulnerable file was incorrectly identified as vulnerable. **True negative** means that a non-vulnerable file was correctly identified as a not vulnerable. **False negative** means that a vulnerable file was incorrectly identified as a not vulnerable.

The last two indicators are calculated using the previous ones. $Precision$ is calculated as $\frac{TP}{TP+FP}$, where TP is true positive, and FP is false positive. $Precision$ shows how many positively identified items were correct, that is, how many files are actually vulnerable, out of all files that classifier identified as vulnerable. The other indicator is $Recall$. It is calculated as

TABLE I
A LIST OF INTERESTING TOPICS. THE NUMBER BEFORE THE # SYMBOL INDICATES THE RUN NUMBER, THIS IS USED TO DISTINGUISH THE DATA BETWEEN FIRST AND SECOND RUN. THE VALUES ARE PROVIDED AS PROBABILITIES, WHERE 0 MEANS IT WILL NEVER OCCUR, AND 1 MEANS THAT IT ALWAYS OCCURS. μ REPRESENTS THE MEAN, AND σ REPRESENTS THE STANDARD DEVIATION

Topic name	μ	*Maximum value*	σ
1#Topi14	0.029	0.381	0.033
1#Topic19	0.026	0.288	0.03
1#Topic21	0.006	0.075	0.007
1#Topic43	0.098	0.781	0.1
1#Topic52	0.16	0.898	0.146
1#Topic63	0.319	0.985	0.22
1#Topic73	0.179	0.939	0.158
1#Topic95	0.174	0.928	0.156
2#Topic39	0.087	0.696	0.13
2#Topic60	0.121	0.78	0.137
2#Topic65	0.103	0.843	0.134

$\frac{TP}{TP+FN}$, where TP is true positive, and FN is false negative. $Recall$ indicates how many true positive files were found, that is, out of all actually vulnerable files how many were classified as vulnerable.

V. RESULTS

A. LDA

GibbsLDA++ produces few files. A .others file that contains the information about the model, such as number of topics, or number of iterations, this is mostly what is provided by the user. A .phi file, that contains a word-over-topic distribution, that is, a probability that a given word appears in that topic. The .phi file is used for finding the most popular words in the topic. A .theta file that contains the the topic-over-document distribution, that is a probability that a given topic appears in a document, where a document is a file containing source code. The topics in this document are used as training attributes in machine learning. A .tassign file is used to store model training data, it is desired to continue iterating model, with same, or new parameters, this file is not relevant for this paper. Lastly, .twords contains the most popular words from each topics.

Majority of the topics that were extracted were not particularly interesting. Not interesting means, that the $mean$ of all the probabilities of all words is extremely close to 0. The $mean$ being very close to 0, means that most of the words have very low chance of appearing in the topic. Since majority of the words have very low chance appearing, it also means that majority of the words have equal chance of appearing, thus implying that the topic is mostly an equal mixture of all words from the corpus. For a source code file, it means that the file composition is very similar to all other files in the whole program.

Majority of topics in the first run, had the $mean$ probability smaller than 0.001, where as the majority of topics from the second run had a $mean$ probability for the words was under 0.02. For the first run, an interesting topic was distinguished as a topic with a mean word probability over 0.001, and for the second run, a topic that had a mean world probability of 0.02. Different qualifiers are used for the two runs, because the mean word probability over **all** topics in the second run was an order of magnitude higher than the first run. In total, 11 interesting topics were extracted. Since LDA cannot infer topic names, they are simply named numerically, and their corresponding is completely random, and has no meaning whatsoever. Following topics were found interesting from first run: topic14, topic19, topic21, topic43, topic52, topic63, topic73, and topic95. Following topics were found interesting from the second run: topic39, topic60, and topic65. A visualization of these topics can be seen in figure 1. The common theme of all these figure's is that their $means$ are is about 10 times larger than rest of the topics from the run. All of the $means$, the maximum value, and standard deviation can be see in table I

B. Weka

1) First run: This subsection contains the data obtained from the first run. The summary can be found in II The first run, with a total of 26 576 items, out of which 25 804 were marked as not vulnerable, and 772 files that were marked as vulnerable, with 100 topics and 500 iterations. This set-up is the main point of interest in this paper. It represents the full corpus, with a default value of 100 topics, that was iterated 500 times.

The first classifier to be tested was **J48**. It classified every single file as not vulnerable. This means that every single that is vulnerable, was incorrectly classified as a not vulnerable. Because none of the vulnerable files were identified, it gives 0% to both true positive and true negative. This also results, that all files that could have been classified as true negatives, were classified as such, giving 100% to true negative, and since every single vulnerable file was classified as non-vulnerable, it means that the maximum value for false negatives is reached - 100%. Because no positives were found, both precision and recall are at 0%

Second classifier to be used was **RandomForest**. It classified 137 files as being vulnerable, out of which only 2 were actually vulnerable, and 135 were not vulnerable. This results in the rate of 0.259% true positive rate, and 0.523% false positive rate. 26 439 items were classified as being not vulnerable, out of which 25 669 were actually not vulnerable and 770 were vulnerable. This results in the rate of 99.477% true negative and 99.741% as false negative. Using this information, precision is calculated to be 1.46% and recall as 0.259%

Third classifer to be used was **REPTree**. It classified, just as J48, all of the files as non-vulnerable. Because in both of the classifiers, all files are classified the same, it results in the same metrics, 0% TP, 0% FP, 100% TN, 100% FN, 0% precision, 0% recall.

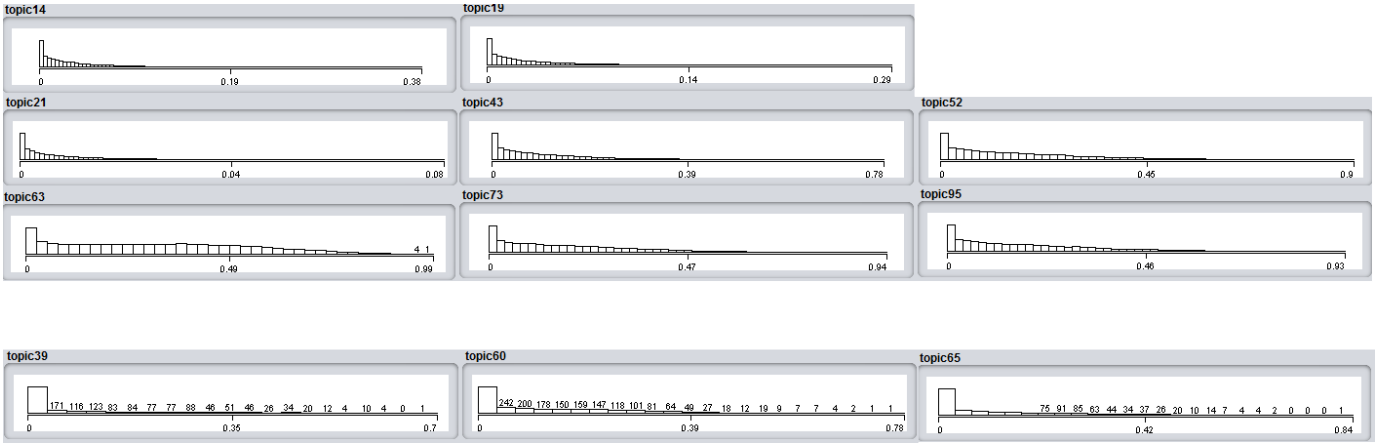

Fig. 1. The y axis represents the probability of a word occurring in the topics. The x axis represent how many words fall into the probability categories from y axis. The higher the bar, the more words occur at that probability. The upper half of the figure provides visualization for interesting topics from first run, while the lower half represents the interesting topics from the second run This graphic has been produced using Weka visualizer tool, and cropping out irrelevant topics.

TABLE II
MACHINE LEARNING PERFORMANCE INDICATORS FROM THE FIRST RUN. FIRST NUMBER SHOWS THE TOTAL NUMBER OF ITEMS IN THE CATEGORY, WHILE THE SECOND NUMBER SHOWS THE PERCENTAGE IN REGARDS TO THE MAXIMUM VALUE POSSIBLE FOR THAT CELL. THAT IS, TP AND FN ARE PERCENTAGES OF ALL FILES ASSOCIATED WITH A SOFTWARE VULNERABILITY, AND FP AND TN ARE PERCENTAGES OF ALL FILES THAT HAVE NO CONNECTION TO SOFTWARE VULNERABILITIES

Algorithm name	*True positive (TP)*	*False positive (FP)*	*True negative (TN)*	*False negative (FN)*	*Precision*	*Recall*
J48	0 (0%)	0 (0%)	25804 (100%)	772 (100%)	0 (0%)	0 (0%)
RandomForest	2 (0.259%)	135 (0.523%)	25669 (99.477%)	770 (99.741%)	1.46%	0.259%
REPTree	0 (0%)	0 (0%)	25804 (100%)	772 (100%)	0 (0%)	0 (0%)
RandomTree	5 (0.648%)	217 (0.841%)	25587 (99.159%)	767 (99.352%)	2.252%	0.648%
DecisionTable	0 (0%)	1 (0.004%)	25803 (99.996%)	772 (100%)	0 (0%)	0 (0%)
BayesNet	4 (0.518%)	34 (0.132%)	25770 (99.868%)	768 (99.482%)	10.526%	0.518%
NaiveBayes	4 (0.518%)	35 (0.136%)	25769 (99.864%)	768 (99.482%)	10.256%	0.518%

RandomTree classified 222 files as vulnerable. From those 222 files, 5 were actually vulnerable, and the remaining 217 were non-vulnerable. This means that TP is 0.648% and that FP is 0.841%. It also classified 26 354 files as not vulnerable, from which 25 587 were actually not vulnerable, and 767 were actually vulnerable. This information allows to calculate, TN of 99.159% and FN 99.352%. Using this data, the precision is found to be 2.252% and the recall to be 0.648%.

DecisionTable classified only 1 files as vulnerable, and this file was actually not vulnerable. This means that TP is 0% and FP is 0.004%. Remaining 26 575 were classified as not vulnerable, out of which 25 803 were actually not vulnerable, and 772 were vulnerable. This means that every single vulnerable file was misclassified, resulting in TN of 99.996% and FN of 100%. Because no true positives were found, both recall and precision are at 0%.

BayesNet classified 38 files as vulnerable. From these 38, only 4 were true positives, and remaining 38 were false positives. Thus, in percentage, the TP is 0.518% and FP is 0.132%. This leaves 26 538 files identified as negatives, from which 25 770 are true negatives, and 278 as false negatives. This results in the TN being 99.868% and FN being 99.482%. Using this, the precision is calculated to be 10.526%, and recall is calculated as 0.518%

NaiveBays classified 39 files as vulnerable. From these, 4 were vulnerable, and 35 were non vulnerable. The TP rate is 0.518% and FP rate is 0.136%. Other 26 537 files were classified as negatives, that not vulnerable, out of which 25 769 were actually not vulnerable, and 768 were vulnerable. The TN rate is found as 99.864% and FN rate is 768%. The precision is 10.256% and the recall is 0.518%.

2) Second run: This subsection describes the data obtained from training machine learning algorithms, with the data obtained from the second run. A second run was performed to compensate for the lower than desired number of iterations in the first run. This run consisted of 100 topics, and 2 000 iterations, but was only using 10% of the code base. In total 2 669 were used, out of which 2 580 are known to be not associated with known vulnerabilities, and 89 are known to be associated with known vulnerabilities in the code base. This keep the vulnerable non vulnerable file ratio of approximately 1 to 29.

RandomForest identified 26 files as vulnerable. Out of these 26 files, 7 items were actually vulnerable, and 29 files were not vulnerable. This results in true positive being 7.865%, and the false positive being 1.124%. The remaining 2 633 files

TABLE III
MACHINE LEARNING PERFORMANCE INDICATORS FROM THE SECOND RUN. FIRST NUMBER SHOWS THE TOTAL NUMBER OF ITEMS IN THE CATEGORY, WHILE THE SECOND NUMBER SHOWS THE PERCENTAGE IN REGARDS TO THE MAXIMUM VALUE POSSIBLE FOR THAT CELL. THAT IS, TP AND FN ARE PERCENTAGES OF ALL FILES ASSOCIATED WITH A SOFTWARE VULNERABILITY, AND FP AND TN ARE PERCENTAGES OF ALL FILES THAT HAVE NO CONNECTION TO SOFTWARE VULNERABILITIES

Algorithm name	*True positive (TP)*	*False positive (FP)*	*True negative (TN)*	*False negative (FN)*	*Precision*	*Recall*
J48	0 (0%)	0 (0%)	2580 (100%)	89 (100%)	0 (0%)	0 (0%)
RandomForest	0 (0%)	0 (0%)	2580 (100%)	89 (100%)	0 (0%)	0 (0%)
REPTree	0 (0%)	0 (0%)	2580 (100%)	89 (100%)	0 (0%)	0 (0%)
RandomTree	7 (7.865%)	29 (1.124%)	2551 (98.876%)	82 (92.135%)	19.444%	7.865%
DecisionTable	0 (0%)	0 (0%)	2580 (100%)	89 (100%)	0 (0%)	0 (0%)
BayesNet	0 (0%)	0 (0%)	2580 (100%)	89 (100%)	0 (0%)	0 (0%)
NaiveBayes	0 (0%)	0 (0%)	2580 (100%)	89 (100%)	0 (0%)	0 (0%)

were classified as not vulnerable, out of which 2 551 were actually not vulnerable, and 82 were known to be vulnerable. This results in true negative being 98.876% and false negative being 92.135%. Using the before mentioned data, and the formula described in section IV, precision is calculated to be 19.444% and, recall is calculated as 7.865

The remaining classifiers, that is **J48, RandomForest, REPTree, RandomTree, DecisionTable, BayesNet, NaiveBayes** resulted in same data. They all classified all of the items as not vulnerable. Because the data is exactly the same percentage-wise, as in first run's $J48$ and $REPTree$ results, the data is the same. TP and FP being 0%, TN and FN being 100%, and precision and recall being 0% because no true positives were found.

VI. DISCUSSION

First thing to note, that out of 100 topics in the first run, only 8 were interesting, and then only 3 topics were found to be interesting in the second run. This might be because the vocabulary in software engineering when it comes to coding is very limited, and once a certain words gets assigned to a topic, it is more likely that similar words will cling to it, creating bigger topics, because so many of the source code files have the same elementary composition. This is more pronounced in the second run, because the number of files, and thus the number of unique words is even smaller there. Therefore, the smaller corpus with a limited vocabulary results in fewer interesting topics, due to words belonging to just few topics.

Judging from results, it is clear that the current implementation of the experiment cannot be used to identify source code files, that might be responsible for vulnerabilities. Majority of the algorithms described in the result section, not only were they not able to distinguish reliable between vulnerable files, but could not distinguish between those two types in any statistically significant manner. This was particularly apparent in the algorithms from tree class, such as REPTree, RandomTree, and RandomForest. Compared to Bayesian class algorithms (BayesNet, NaiveBayes), were less likely to classify vulnerable files as vulnerable. This can be seen in trees results, where very few, to none at all, vulnerable files were classified, be it true positive, or false positive. One explanation could be that they were not able to model differences between vulnerable files and non-vulnerable files. This explanation could be argumented, because Chromium is a network application, a lot of the code is responsible for network functionality. Since software attacks usually happen over the Internet, in other programs with known vulnerabilities might manifest more often in files associated with networking, which would give the vulnerable files a more unique topic distribution compared to the rest of the codebase. Such difference might be harder to distinguish in software that is deeply embedded in working with Internet.

The best performing algorithms from the first run were Bayesian. Both BayesNet, NaiveBayes produced the highest precision, both over 10%. The second place goes to RandomTree with 2.252% precision, and RandomForest takes the third place with 1.46% precision. On the other hand, RandomTree achieved a higher recall rate of 0.648, in comparison to Bayesian algorithms, which got a smaller rate of 0.518%. In the second run, Bayesian algorithms performed significantly worse, getting 0% both in recall and in precision. Where RandomTree achieved an even greate precision and recall, even compared to the first run's Bayesian methods, having precision of 19.444% and recall of 7.865%.

Majority of the algorithms did not result in any useful data from the second run. I suspect the main cause for this is the reduced number of files, despite the fact that it did not affect the RandomTree negatively.

Another noteworthy item can be seen from results. Bayesian algorithms were able to identify more vulnerable files, as actual vulnerable files. This could be used to argue that Bayesian algorithms were able to predict the status of files better, but it does not seem to be the case. This is because the percentage of real vulnerable files that were classified as vulnerable was around 0.6%, which is about the same number as amount of non-vulnerable files that were classified as vulnerable. It can be assumed that the Bayesian models inferred that the chance for a vulnerable file is 0.6% and just randomly guessed which files were vulnerable. This would explain, why about 0.6% of both vulnerable and non-vulnerable files were classified as vulnerable.

Last observation, is that comparing accuracy of models generated by machine learning algorithms from the first run, to the models from the second run, the second run performed noticeably better. The main differences between the two runs were that the second run had greater number of iterations, as well as greater number of topics. This could be used as an argument that more iterations will produce noticeable more accurate results, but such would be disputed from the data second run. The second run had the most instances of algorithms grouped all data into a single class for non-vulnerable files, despite the fact that same number of topics were used, and greater number of iterations. Low accuracy of the second run shows, that using stratified sample did not prove to be a valid strategy. One reason for that is the small number of vulnerable files. With 89 vulnerable files, there might have not been enough to form a unique identity with topic models, compared to the rest of the code base, resulting in greater homogeneity of topic distributions between all of the files.

VII. Threats to validity

Construct validity. The experiment was designed to measure if it is possible to predict software vulnerabilities using topic extraction and machine learning. The biggest threat to construct validity was using only a single implementation of topic modeling - LDA. This decision felt comfortable, because it is one of the most popular methods of topic modeling. The original paper by Blei et al. [7] has been cited over 14 500 times, and has multiple implementations in C, Java, MatLab, R, etc. Since this is not a comparative study of topic modeling algorithms, LDA was chosen as the most popular option.

Conclusion validity. The greatest threat to validity of conclusion would be claiming that there is no relationship between topics in source code, and the chance for that file to be responsible for a software vulnerability. The conclusion of the paper is that no correlation has been found with the current design of the experiment, and few reasons for that are mentioned, as well as suggestions how they can be remedied. To arrive at this conclusion, standard methods were used with little to no customization, such as GibbsLDA++ and Weka. Only changes that were made, was formatting the data to required format, to be used with respective software.

Internal validity. There is very little threat to internal validity, because there were no actual choices to make. All of the data that was available to used, were used uniformly without any discrimination. Stratified sample was done using pseudo-random number generator provided by Python, and cross-validation was done using folds, so that every part of data set was used both for training, and evaluating machine learning.

External validity. The reached conclusion only applies for Chromium when it comes to detecting software vulnerabilities using LDA. Different results could be reached using another method of topic modeling, as well as using it for other purpose than vulnerability prediction. For example, it is possible that LDA might lend itself for the use of finding source files, that are similar to each other. More research has to be done to definitively disprove connection between vulnerability prediction and topic extraction. This is because the experiment was done on a single program, using a single topic extraction model.

VIII. Conclusion

The most reliable algorithms for topic detection were Bayesian ones - **BayesNet and NaiveBayes**, in terms of $precision$, but **RandomTree** performed slightly better in the terms of $recall$, at the cost of significantly lower $precision$. On the other hand, when the data set became much smaller, **RandomTree** became a clear leader, both in $precision$ and $recall$

No data has been found, that would support the theory, that software vulnerabilities can be predicted using latent Dirichlet allocation analysis with combination of machine learning. Even though no clear link has been found between topic models and software vulnerabilities in this experiment, it does not mean that these two cannot be used together. As mentioned in discussion, results might have been different if source code from another program was used, due to how Chromium is closely related to networking. Another matter to investigate is a different experiment design, using different tokenization methods and removing fewer symbols from the source code, or using fewer topics for LDA. The data can also be affected by the number and a method of discretization for topic distribution. It also might be worth investigating this case more, if there is an access to more powerful resources that would allow to perform topic modeling with greater number of topics, as well as for more iterations, as it was demonstrated in the second run.

References

[1] J. J. G. Jaziar Radianti, "Understanding hidden information security threats: The vulnerability black market," in *Proceeding HICSS '07 Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, p. 156, 2007.

[2] M. Finifter, D. Akhawe, and D. Wagner, "An empirical study of vulnerability rewards programs.," in *USENIX Security*, vol. 13, 2013.

[3] V. Gupta, D. N. Ganeshan, and D. T. K. Singhal, "Developing software bug prediction models using various software metrics as the bug indicators," *IJACSA*, 2015.

[4] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent dirichlet allocation," *Information and Software Technology*, vol. 52, no. 9, pp. 972–990, 2010.

[5] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi, "Mining concepts from code with probabilistic topic models," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pp. 461–464, ACM, 2007.

[6] G. Maskeri, S. Sarkar, and K. Heafield, "Mining business topics in source code using latent dirichlet allocation," in *Proceedings of the 1st India software engineering conference*, pp. 113–120, ACM, 2008.

[7] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *the Journal of machine Learning research*, vol. 3, pp. 993–1022, 2003.

[8] J. K. Pritchard, M. Stephens, and P. Donnelly, "Inference of population structure using multilocus genotype data," *Genetics*, vol. 155, no. 2, pp. 945–959, 2000.

[9] P. Simon, *Too Big to Ignore: The Business Case for Big Data*, vol. 72. John Wiley & Sons, 2013.

[10] E. Alpaydin, *Introduction to machine learning*. MIT press, 2014.

[11] D. Freitag, "Machine learning for information extraction in informal domains," *Machine learning*, vol. 39, no. 2-3, pp. 169–202, 2000.
[12] R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, *Machine learning: An artificial intelligence approach*. Springer Science & Business Media, 2013.
[13] R. Raina, A. Battle, H. Lee, B. Packer, and A. Y. Ng, "Self-taught learning: transfer learning from unlabeled data," in *Proceedings of the 24th international conference on Machine learning*, pp. 759–766, ACM, 2007.
[14] J. Walden, J. Stuckman, and R. Scandariato, "Predicting vulnerable components: Software metrics vs text mining," in *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*, pp. 23–33, IEEE, 2014.
[15] S. Neuhaus and T. Zimmermann, "Security trend analysis with cve topic models," in *Software reliability engineering (ISSRE), 2010 IEEE 21st international symposium on*, pp. 111–120, IEEE, 2010.
[16] R. Das, S. Sarkani, and T. A. Mazzuchi, "Software selection based on quantitative security risk assessment," *IJCA Special Issue on Computational Intelligence & Information Security*, pp. 45–56.
[17] M. Fernández-Delgado, E. Cernadas, S. Barro, and D. Amorim, "Do we need hundreds of classifiers to solve real world classification problems?," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 3133–3181, 2014.