Natural Language Insights from Code Reviews that Missed a Vulnerability

A Large Scale Study of Chromium

Nuthan Munaiah[1], Benjamin S. Meyers[1], Cecilia O. Alm[2], Andrew Meneely[1], Pradeep K. Murukannaiah[1], Emily Prud'hommeaux[2], Josephine Wolff[2], and Yang Yu[3]

[1] B. Thomas Golisano College of Computing and Information Sciences
{nm6061,bsm9339,axmvse,pkmvse}rit.edu
[2] College of Liberal Arts
{coagla,emilypx,jcwgpt}@rit.edu
[3] Saunders College of Business
yyu@saunders.rit.edu
Rochester Institute of Technology, Rochester, NY 14623, USA

Abstract. Engineering secure software is challenging. Software development organizations leverage a host of processes and tools to enable developers to prevent vulnerabilities in software. Code reviewing is one such approach which has been instrumental in improving the overall quality of a software system. In a typical code review, developers critique a proposed change to uncover potential vulnerabilities. Despite best efforts by developers, some vulnerabilities inevitably slip through the reviews. In this study, we characterized linguistic features—inquisitiveness, sentiment and syntactic complexity—of conversations between developers in a code review, to identify factors that could explain developers missing a vulnerability. We used natural language processing to collect these linguistic features from 3,994,976 messages in 788,437 code reviews from the Chromium project. We collected 1,462 Chromium vulnerabilities to empirically analyze the linguistic features. We found that code reviews with lower inquisitiveness, higher sentiment, and lower complexity were more likely to miss a vulnerability. We used a Naïve Bayes classifier to assess if the words (or lemmas) in the code reviews could differentiate reviews that are likely to miss vulnerabilities. The classifier used a subset of all lemmas (over 2 million) as features and their corresponding TF-IDF scores as values. The average precision, recall, and F-measure of the classifier were 14%, 73%, and 23%, respectively. We believe that our linguistic characterization will help developers identify problematic code reviews before they result in a vulnerability being missed.

1 Introduction

Vulnerabilities in software systems expose its users to the risk of cyber attacks. The onus of engineering secure software lies with the developers who must assess the potential for an attack with each new line of code they write. Over

the years, software development teams have transitioned toward a proactive approach to secure software engineering. Modern day software engineering processes now include a host of security-focused activities from threat modeling during design to code reviews, static analysis, and unit/integration/fuzz testing during development. The code review process, in particular, has been effective in uncovering a wide variety of flaws in software systems, from defects [27] to vulnerabilities [7, 9, 29]. Code reviews have now become such an essential part of software development lifecycle that large software development organizations like Google [13] and Microsoft [25] mandate code review of every change made to the source code of certain projects.

Developers make mistakes. Code reviews provide an opportunity for these mistakes to be caught early, preventing them from becoming an exploitable vulnerability. Done in an systematic way, code reviews have the potential to uncover almost all defects in a software system [14].

Code reviews contain a wealth of information from which one can gain valuable insights about a software system and its developers. Conversations between developers participating in a code review often represent instances of constructive criticism and a collaborative effort to improve the overall quality of the software system. However, in some cases, the same conversations could contain clues to indicate potential reasons for a mistake, such as a vulnerability, to have been missed in the review. As with software development, code reviews involve humans—the developers. Developers participating in code reviews could exhibit a wide variety of socio-technical behaviors; some developers may be verbose, inquisitive, overly opinionated, and security-focused, while others may be succinct, cryptic, and uncontroversial. The natural language analysis of code review conversations can help identify these intrinsic (linguistic) characteristics and understand how they may contribute to the likelihood of a code review missing a vulnerability. Furthermore, we can use automated natural language processing techniques to identify these characteristics on a massive scale aiding developers by highlighting problematic code reviews sooner.

Our goal in this study is *to characterize the linguistic features that contribute to the likelihood that a code review has missed a vulnerability.* We empirically analyzed 3,994,976 messages across 788,437 code reviews from the Chromium project. We addressed the following research questions:

RQ1 Feedback Quality Do linguistic measures of inquisitiveness, sentiment, and syntactic complexity in code reviews contribute to the likelihood that a code review has missed a vulnerability?

RQ2 Lexical Classifier Can the words used differentiate code reviews that have missed a vulnerability?

The remainder of this paper is organized as follows: we begin with brief summary of prior literature closely related to ours in Section 2. In Section 3, we describe the approach used to obtain the data from a variety of sources, collect various metrics from the data, and statistically analyze the metrics. We present our results in Section 4, highlight some of the limitations in Section 5, and conclude the paper with a brief summary in Section 6.

2 Related Work

Code reviews are used widely in the software engineering field with the goal of improving the overall quality of software systems. Despite the popularity and evidence justifying the benefit of code reviews [5, 27], some research suggests that code reviews are not always carried out properly, diminishing their utility in the software development cycle [14, 15]. The focus of prior research on code reviews has been to understand attributes of code reviews that express their usefulness [8] and identify aspects of the code review process that enable timely conclusion of reviews [4]. While some studies from prior literature [5,16,29] have questioned the effectiveness of code reviews in finding vulnerable code, these studies do not provide an insight into the factors that may have led to the code reviews missing vulnerabilities.

Bosu and Carver [7], who found evidence to support the notion that code reviews are effective at uncovering vulnerabilities, used text mining techniques to compile a list of keywords related to various types of vulnerabilities. Using a similar approach, Pletea *et al.* [35] performed sentiment analysis of comments on GitHub pull requests and found that security-oriented comments were typically more negative. Guzman *et al.* [22] performed similar analysis but with correlating sentiment in commit messages with social and environmental factors. While these studies present interesting findings, they still do not explain why code reviews often overlook vulnerabilities. To address this question, we not only use sentiment analysis and lexical information, but also explore more complex natural language processing approaches for analyzing code reviews at the structural and word frequency distribution level. To the best of our knowledge, our work is one of the first to attempt analyses of this nature, especially in the context of a large-scale data set of 788,437 code reviews from the Chromium project.

3 Methodology

In the subsections that follow, we describe the methodology used in the empirical analysis. At a high-level, our methodology may be organized into three steps: (1) data collection, (2) metric collection, and (3) statistical analysis.

3.1 Data Collection

Data Sources The data set used in the empirical analysis is a collection of code reviews with their associated messages and metadata (bug and vulnerability identifiers), which was obtained from a variety of managed sources. The code reviews, specifically, the messages posted by reviewers, were the central pieces of information in our data set. The Chromium project uses Rietveld[4] to facilitate the code review process. We used Rietveld's RESTful API to retrieve all code reviews (2008–2016) for the Chromium project as JSON formatted documents.

[4] `https://codereview.chromium.org/`

A typical code review in the Chromium project is created when a developer wishes to have changes to the source code integrated with the Chromium repository. The group of files changed is called a patchset. A code review may have one or more patchsets depending on the changes the developer had to implement to address the comments in the code review. In retrieving the code reviews, we also retrieved any associated patchsets. The patchsets were used to identify the files that were reviewed and those that were committed as part of the code review.

The goal in our study is to characterize the linguistic features of code reviews and their relationship with the likelihood of missing a vulnerability. A code review is said to have missed a vulnerability if at least one of the files reviewed was later fixed for a vulnerability. Therefore, the key piece of information needed in the analysis is a mapping between code reviews and vulnerabilities. In the Chromium project, the association between code reviews and vulnerabilities is achieved through an issue tracking system. The Chromium project tracks bugs using Monorail.[5] The report of bugs that resulted in the resolution of a vulnerability have the Common Vulnerabilities and Exposure (CVE) identifier of the vulnerability as a label. We used the bulk export feature supported by Monorail's web interface to download, in CSV format, a list of all bugs in the Chromium project. All code reviews that have bugs associated with them are expected to have the bug identifier(s) mentioned (using the template: `BUG=<bug_id>,<bug_id>`) in the description of the reviews. We parsed the code review description to identify the bug(s) that were associated with a code review.

We compared the vulnerabilities obtained from Monorail to those obtained from the National Vulnerability Database (NVD[6]) to ensure completeness. We found a small set of vulnerabilities that were resolved by the Chromium project team but no record of a mapping between the vulnerability and a bug existed in Monorail. We manually identified the mapping between the vulnerability and bug using posts from the Chrome Releases Blog[7] and the references list from the vulnerability report on NVD.

Data Annotations The code reviews in our data set were annotated with labels to enable us to group reviews into categories that were relevant to our empirical analysis. In the subsections that follow, we introduce these labels and describe the approach we used to assign those labels to code reviews.

(1) Code Reviews that Fixed Vulnerabilities - We used the label *fixed vulnerability* to identify code reviews that facilitated the review of a fix for a vulnerability. In our data set, we annotated all code reviews that were associated with a bug that resolved a vulnerability with the *fixed vulnerability* label. The fixed vulnerability code reviews were crucial in identifying the files that were reviewed (and possibly committed) in resolving a vulnerability. Intuitively, the fixed vulnerability code reviews represent the conversations that

[5] `https://bugs.chromium.org/p/chromium`
[6] `https://nvd.nist.gov/`
[7] `https://chromereleases.googleblog.com/`

the developers should have had in the past to potentially discover and resolve vulnerabilities.

(2) Code Reviews that Missed Vulnerabilities - We used the label *missed vulnerability* to identify code reviews that *potentially* missed a vulnerability. We identify missed vulnerability code reviews by searching for code reviews that reviewed a file that was later fixed for a vulnerability. In our study, We followed a two step process to identify missed vulnerability code reviews:

Step 1 For each fixed vulnerability code review, identify all files committed.

Step 2 For each committed file, identify all code reviews, created before the review in question, that included the file.

Intuitively, the missed vulnerability code reviews represent the *missed opportunities* in which vulnerabilities could have been discovered. We base our intuition on prior research by Meneely *et al.* [30], who found that vulnerabilities tend to exist in software for over two years before being discovered. While no single code review can be blamed for missing a vulnerability, we believe that, in aggregate, the missed vulnerability code reviews constitute an opportunity in which the vulnerability could have been discovered.

(3) Neutral Code Reviews - We used the label *neutral* to identify code reviews that neither reviewed the fix for a vulnerability nor missed a vulnerability. The neutral code reviews serve as the control group in our analysis. The choice of using the generic label "neutral" is intentional; one cannot definitively say that a code review did not miss a vulnerability since there may be latent vulnerabilities in the source code that are yet to be discovered [30]. Intuitively, the set of neutral reviews could potentially include those reviews in which reviewers may have overlooked a vulnerability.

Summary The data set used in the empirical analysis consists of 788,437 code reviews containing a total of 3,994,976 messages posted by 8,119 distinct participants. On average, each review had 2 participants, reviewed 9 files, had 6 messages, and lasted 67 days. The data set also includes 436,191 bugs and 1,462 vulnerabilities. Among the code reviews, 877 were labeled as *fixed vulnerability*, 92,030 were labeled as *missed vulnerability* and 695,530 were labeled as *neutral*.

3.2 Metric Collection

In the subsections that follow, we describe the metrics used to address the research questions and the approaches used to extract these metrics. All metrics are defined at the message level; however, we aggregate them at the review level for empirical analysis. We used the Natural Language Toolkit (NLTK) [6], the Stanford CoreNLP [26], and the Speech Processing & Linguistic Analysis Tool (SPLAT) [32] in collecting these metrics from the code review messages.

Inquisitiveness Uncovering security flaws in a software system involves a speculative thought process. A reviewer must consider the possibility that even the

most unlikely scenario could have an impact on the piece of code being reviewed. The *inquisitiveness* metric is an attempt to quantify this speculative type of conversation in code reviews.

The inquisitiveness metric is estimated by counting the number of questions in a code review message. We have used a naïve approach to estimate the value of this metric by the frequency of the symbol "?" in the message text. The assumption here is that the number of questions in a message is correlated with the number of occurrences of "?". We validated this assumption by manually counting the number of questions in a sample of 399 code review messages obtained by random stratified sampling of messages with zero, one, and more than one occurrences of the symbol "?". In the manual analysis, we not only looked for the "?" symbol but also read the content of the messages to consider sentences phrased as a question but not terminated by "?". For instance, here is an excerpt from a message that contains a question (terminated by "?") and a sentence that is phrased as a question but without being terminated by "?": "I'm not sure ... immutable. Is it OK ... is called? If it's OK, why we don't ... of MIDIHost.". We used Spearman's rank correlation co-efficient (ρ) to quantify the correlation between the manually estimated number of questions and the number of occurrences of the symbol "?". We found a strong, statistically significant (p-value $\ll 0.01$), positive correlation with $\rho = 0.93$.

In our approach to calculating the inquisitiveness metric, we used NLTK to tokenize the data (i.e. break up sentences into words, separating out punctuation marks) and to compute the frequency of "?". In the manual analysis of the 399 code review messages, we found that the inquisitiveness metric tends to over-estimate the actual number of questions in cases when a single question is terminated with multiple question marks or when a URL is incorrectly terminated at the "?" by the NLTK tokenizer. The inquisitiveness metrics also misses questions if they are not terminated by "?". We, however, found only a few instances of these cases in our manual analysis.

We chose to use the seemingly simplistic approach rather than more sophisticated ones such as regular expressions or syntactic parse trees because, unlike traditional natural language, code review messages tend to be informal and, sometimes, fragmented. Furthermore, the notion of a question in code review messages tends to go beyond the message itself, requiring additional context such as the line of source code that the sentence in a message is associated with.

The number of questions is likely to be positively correlated with the size of the message. We accounted for this likelihood by expressing the metric as inquisitiveness per sentence in the message. We used Stanford CoreNLP to split a message into sentences to count the number of sentences.

Sentiment Conversations about the security of a software system tend to have a non-neutral sentiment [35]. The *sentiment* group of metrics is an attempt to capture the sentiment associated with code review messages. Each of the sentiment metrics is calculated as the proportion of sentences in a code review

message exhibiting that sentiment. We used proportion of sentences, rather than the actual number of sentences, to control for message size.

We used the sentiment analysis model [39] from the Stanford CoreNLP to identify the sentiment expressed in sentences. The model uses information about words and their relationships to classify sentences into one of five sentiment classes: very positive, positive, neutral, negative, and very negative. In our study, we merge very negative and negative into a single negative class and very positive and positive into a single positive class. Furthermore, we do not consider the neutral sentiment in the analysis of the sentiment metrics. In effect, we only consider the positivity (proportion of positive sentiment sentences) and negativity (proportion of the negative sentiment sentences) in the analysis.

Term-Frequency Inverse-Document-Frequency (TF-IDF) Widely used in the field of information retrieval [38], TF-IDF is a measure of the relative importance of a term (word) within a document with the term's frequency (TF) in that document normalized by its frequency in the corpus to which that document belongs (inverse document frequency, IDF). In our data, a single code review is a document and the corpus is the collection of all code reviews.

There are many ways to calculate TF-IDF. We have used the approach implemented by the TextCollection[8] class of NLTK to compute the TF-IDF metric. In this approach, the TF-IDF of a term (t) in a document (d) is computed as $tf\text{-}idf_{(t,d)} = tf_{(t,d)} \times df_t$, where $tf_{(t,d)}$, the term frequency, is the ratio of the frequency of the term t (in document d) to the total number of terms in d and df_t, the document frequency, is the natural logarithm of the ratio of the number of documents in the corpus to the number of documents containing the term t.

As with the inquisitiveness metric, we used NLTK to tokenize code review messages, but we perform a key preprocessing of the tokens before computing the TF-IDF: map token to base form, or lemma. We compute the TF-IDF of lemmas instead of tokens to account for morphological variation (i.e., suffixes we add to words to express different grammatical functions); the words *compiles, compiling* and *compiled*, for instance, are all forms of the lemma *compile*.

Syntactic Complexity Software engineers participating in a code review are required to process a considerable amount of information in reviewing a piece of code. The structural complexity of the language used in code review messages could lead a developer to misunderstand a comment and consequently introduce a spurious change. The *syntactic complexity* group of metrics is aimed at quantifying the complexity that may be in code review messages.

A variety of metrics have been proposed to quantify the syntactic complexity of natural language sentences. While some of these metrics focus simply on sentence or utterance length [34] or on part of speech information, others use information about the structure of sentences that can be extracted from syntactic parses or trees. In particular, there are several parse-based measures of

[8] `http://www.nltk.org/api/nltk.html#nltk.text.TextCollection`

complexity relying on the assumption that deviations from a given language's typical branching structure (in the case of English, right branching) are indicative of higher complexity [18,42]. In our study, we use three measures of syntactic complexity to quantify the complexity of code review messages: (1) Yngve score (2) Frazier score, and (3) propositional density. We describe the three metrics in greater detail below. The implementation of these metrics, which are adaptations of the algorithms described in Roark *et al.* [37] and Brown *et al.* [10], have been borrowed from the previously mentioned open-source tool SPLAT [32].

– Yngve Score: The Yngve score [42] is a measure of syntactic complexity that is based on cognitive load [28, 40], specifically on the limited capabilities of the working memory [2, 3, 33]. The Yngve score is computed using the tree representation of a sentence obtained by syntactically parsing the sentence. The tree is scanned using a pushdown stack in a top-down, right-to-left order. For every level of the tree, branches are labeled starting with 0 for the rightmost branch and incrementing by 1 as the parse progresses toward the left branch. Each word is then assigned a word score by summing up the labels for each branch in the path from the root node to the word (leaf node) as seen in Figure 1.
We used the Stanford CoreNLP in conjunction with NLTK to obtain syntactic parse trees for code review messages. In this study, we use the mean Yngve score over all words in a sentence. The Yngve score is then averaged for each sentence in a message and for each message in a code review. See Table 1 for sample Yngve scores.

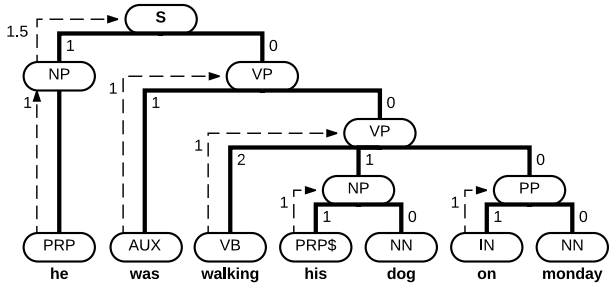

Fig. 1. Parse tree with Yngve (Y) and Frazier (F) scores on the solid and dotted lines, respectively. Word scores are boxed, with the mean score for the sentence to the right.

– Frazier Score: The Frazier score is similar to the Yngve score with one key distinction: while Yngve score is a measure of the the breadth a syntactic tree, Frazier score measures of the depth of the tree [37]. The Frazier score emphasizes embedded clauses (sub-sentences) based on the notion that the number of embedded clauses is associated with greater complexity [18–21].

The algorithm to compute the Frazier score of a sentence is similar to that used to compute its Yngve score. Starting with the leftmost leaf node, each branch leading up to the root of an embedded clause (sentence node) is labeled as 1, with the exception of the branch on the uppermost level of the tree that leads directly to the sentence node, which is labeled as 1.5. This process is repeated for every leftmost leaf node. Labeling stops when the path upward from the leaf node reaches a node that is not the leftmost child of its parent, as shown with dotted lines in Figure 1.

– Propositional Density: The propositional density metric (often referred to as p-density) is a socio-linguistic measure of the number of assertions in a sentence. For example, Chomsky's famous sentence "colorless green ideas sleep furiously [12]" makes two assertions: (1) colorless green ideas sleep, and (2) they do it furiously. Part-of-speech tags and word order are used to determine likely propositions, and a series of rules is applied to determine how many propositions are expressed in a sentence [10]. After propositions have been identified, p-density is simply the ratio of the number of expressed propositions in a sentence to the number of words in that sentence.

Table 1. Sample sentences with syntactic complexity scores

Sentence	**Yngve**	**Frazier**	**P-density**
This CL will add a new H264 codec to the SDP negotiation when H264 high profile is supported.	2.056	0.722	0.278
There is a 3.9% decrease in initial load time observed.	1.500	0.750	0.200
fixed as spec says	0.750	1.500	0.750
lgtm then	0.500	1.750	0.500

Although sentence length can serve as a proxy for syntactic complexity, we chose to use more sophisticated metrics because these metrics take the structure of the sentence into consideration. We did, however, assess the correlation between our complexity metrics, aggregated at the message level, and message length, expressed as number of words. We found a statistically significant (p-value $\ll 0.01$) positive correlation (Spearman's $\rho = 0.69$) between Yngve score and message length. We normalized the Yngve score metric using message length to account for this correlation. The other two syntactic complexity metrics, Frazier score and propositional density, were not significantly correlated with message length in our data.

3.3 Analysis

In the subsections that follow, we describe the various statistical methods used in the empirical analysis of the metrics proposed in our study. All statistical tests were executed with `R` version 3.2.3 [36].

Correlation We used Spearman's rank correlation coefficient (ρ) to assess the pairwise correlation between the various metrics that were introduced earlier. We considered two metrics to be strongly correlated if $|\rho| >= 0.70$ [24].

Association We used the non-parametric Mann-Whitney-Wilcoxon (MWW) test for association between the various metrics and the code reviews that missed a vulnerability. We assume statistical significance at $\alpha = 0.05$. We compare mean of the populations to assess if the value of a metric tends to be higher or lower between two populations (neutral and missed vulnerability code reviews).

Classification To assess if the words used in code reviews can be a differentiator between neutral and missed vulnerability code reviews, we built a Naïve Bayes classifier with lemmas of the words from code reviews as features and their corresponding TF-IDF as values. A typical code review could have hundreds of distinct lemmas; to constrain the number of features in the model, we used the criteria described below to filter the lemmas before attempting to build the classifier.

(1) We only considered alphanumeric lemmas that were fewer than 13 characters in length. We chose 13 characters to be a reasonable limit for lemma length because over 99% of the all words in the Brown [17], Gutenberg [23], and Reuters [1] natural language data sets are fewer than 13 characters in length.
(2) We only considered the top ten lemmas, ordered by the TF-IDF values, in each code review. In other words, we only considered the ten most important lemmas in code reviews.

In addition to constraining the number of features, we also had to constrain the number of observations (code reviews) in the data set used to build the classifier. We used random sampling to select 5% of neutral and 5% of missed vulnerability code reviews. We repeated the random sampling to generate ten (possibly) different data sets to be used to build the classifier. We used the information gain [41] metric to assess the relevance of features in differentiating the code reviews and removed all features with a zero information gain. We then used the relevant features to train and test a Naïve Bayes classifier using 10×10-fold cross validation. We used SMOTE [11] to mitigate the impact of the imbalance in the number of neutral and missed vulnerability code reviews on the performance of the classifier. We used precision, recall and F-measure metrics to assess the performance of the classifier.

4 Results

In the subsections that follow, we address our research questions through the empirical analysis of our metrics.

4.1 RQ1 Feedback Quality

Question: Do linguistic measures of inquisitiveness, sentiment, and syntactic complexity in code reviews contribute to the likelihood that a code review has missed a vulnerability?

We began the analysis by evaluating the correlation between the metrics themselves to understand if any of the metrics were redundant due to high correlation. The correlation analysis did not reveal any strong correlations between the metrics and the highest value of $|\rho|$ was 0.45, between inquisitiveness and Yngve score metrics. We then used the MWW test to assess the strength of association between the metric and the likelihood of a code review missing a vulnerability. The results are shown in Table 2, with all associations statistically significant at least p-value < 0.01. Shown in Figure 3 in the Appendix are the comparative box plots of the inquisitiveness, sentiment, and complexity metrics.

Table 2. Mann-Whitney-Wilcoxon (MWW) test outcome for association between the linguistic measures and the likelihood of a code review missing a vulnerability

Metric	p-value	$\text{Mean}_{\text{neutral}}$	$\text{Mean}_{\text{missed}}$
Inquisitiveness	3.28e−12	0.1785	0.1711
Negativity	< 2.2e−16	0.3707	0.4091
Positivity	< 2.2e−16	0.0625	0.0783
Yngve Score	< 2.2e−16	0.0498	0.0442
Frazier Score	0.0031	0.8568	0.8548
Propositional Density	1.77e−124	0.2634	0.2708

These results reveal some interesting insights into the nature of code reviews that have missed vulnerabilities. First, the lower inquisitiveness values in code reviews that missed a vulnerability suggest that the participants in these reviews may not have been as actively trying to unearth flaws. The approach to uncovering security vulnerabilities requires review participants to question the behavior of the code being reviewed in scenarios that are unlikely but still plausible.

Code reviews with lower inquisitiveness i.e. fewer questions per sentence tend to miss vulnerabilities.

Secondly, participants in code reviews that missed vulnerabilities expressed a higher degree of both positive and negative sentiment. The higher negativity in code reviews that missed a vulnerability confirms the results observed by Pletea *et al.* [35] in security-specific discussions in GitHub pull requests. However, the higher positivity in code reviews that missed a vulnerability is interesting because it seems that the conversation may have started on a negative sentiment but

culminated on a positive sentiment, or vice versa. We may have to chronologically analyze messages to better understand the evolution of sentiment.

> Code reviews with higher sentiment, regardless of the polarity of that sentiment, tend to miss vulnerabilities.

Lastly, the complexity metrics, specifically Yngve and Frazier scores, show that code reviews that missed a vulnerability tend to be less complex. While the result may seem counterintuitive, the lack of complexity could be an indicator that the code review conversation may be lacking substance. The propositional density metric, on the other hand, is higher for code reviews that missed a vulnerability indicating that those code reviews tend to have more assertions.

> Code reviews with lower complexity in terms of Yngve and Frazier and higher complexity in terms of propositional density tend to miss vulnerabilities.

4.2 RQ2 Lexical Classifier

Question: Can the words used differentiate code reviews that have missed a vulnerability?

In RQ1, we found that the inquisitiveness, sentiment, and syntactic complexity in code reviews are associated with the likelihood of a code review missing a vulnerability. In this research question, however, we wanted to understand if the words used in the code reviews itself was different between reviews that were neutral and those that missed a vulnerability.

We computed the TF-IDF of the lemmas of the words used in a code review, and using these features, built a classifier to identify code reviews as neutral or missed vulnerability. Each random sample of data used to build the classifier contained 39,377 code reviews and an average of 25,652 features. Shown in Table 3 is the total number of features and the number of those features deemed relevant (i.e. non-zero information gain) in each of the ten random samples.

Table 3. Number of relevant features in each of the ten randomly sampled sets of code reviews

Sample Set	1	2	3	4	5	6	7	8	9	10
# Features	25,636	25,535	25,456	25,550	25,761	25,805	25,592	25,709	25,651	25,827
# Relevant	1,092	889	959	885	910	1,012	986	910	1,091	898

We incrementally selected increasing number of relevant features to build and evaluate the performance of a Naïve Bayes classifier. Figure 2 shows the distributions of precision, recall, and F-measure obtained from each of the ten random samples for varying number of relevant features selected. As seen in the

figure, the classifier built with 18 relevant features performs the best in terms of F-measure. The average precision, recall, and F-measure of the best performing classifier was 14%, 73%, and 23%, respectively. We note that, in classifying code reviews that are likely to miss a vulnerability, a higher recall is desired even at the cost of lower precision [31] since the cost of revisiting a few misclassified reviews is relatively small when compared to the cost of a missed vulnerability.

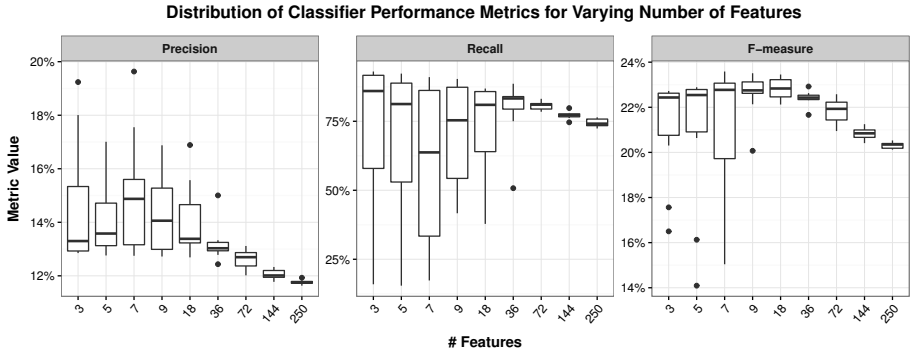

Fig. 2. Distribution of classifier performance metrics—precision, recall, and F-measure—obtained from ten random samples of code reviews for varying number of relevant features used in training the classifier

The ability of our classifier to differentiate between neutral and missed vulnerability code reviews indicates that the distribution of words used in these two types of code reviews is indeed different. In a code review framework, such a classifier could be used to identify code reviews that are potentially missing vulnerabilities. The development team could then revisit these flagged code reviews to ensure that the necessary steps are taken to uncover any latent vulnerabilities.

Yes. The accuracy of our classifier indicates that the words used in code review can be a differentiator between neutral and missed vulnerability code reviews.

5 Limitations

The scale of our data set posed certain computational limitations, specifically in the process of building a lexical classifier. We used the lemma of words in the code review as features in the lexical classifier, however, our data set contains 2,089,579 unique lemmas obtained through lemmatization of 2,197,114 unique words. An attempt to build a classifier using the entire data set $788,437 \times 2,089,579$ seemed intractable. We overcame this limitation by filtering lemmas by length and non-alphanumeric characters, by randomly sampling a small percentage of code reviews, and by selecting the top 10 lemmas by their relative

importance. We mitigated the arbitrariness in the sampling process by repeating it ten times and averaging the results.

The sentiment analysis model that we used was trained with movie reviews and may have misclassified sentences since code reviews tend to have variable names or other artifacts that could skew the analysis. A mitigation could be to train the sentiment analysis model with a sufficiently large sample of manually classified code review messages.

The default parser used to parse the syntax of sentences in Stanford CoreNLP is based on Probabilistic Context-Free Grammar (PCFG). In our initial of analysis, the parser would timeout when parsing long code review messages. In the subsequent analyses, we have used a faster but marginally less accurate Shift-Reduce Constituency Parser (SR). We do not believe that the use of the SR parser may have had an impact on any downstream operations performed on the syntactic parse trees.

6 Summary

In this study, we used natural language processing to characterize linguistic features—inquisitiveness, sentiment and syntactic complexity—in conversations between developers participating in a code review. We collected these features from 3,994,976 messages spread across 788,437 code reviews from the Chromium project. We collected 1,462 vulnerabilities and identified code reviews that had the opportunity to prevent the vulnerability in the past. We then used association analysis to evaluate if the linguistic features proposed were associated with the likelihood of code reviews missing a vulnerability. We found that code reviews with lower inquisitiveness, higher sentiment, and lower complexity were more likely to miss a vulnerability. In addition to the linguistic features, we computed the relative importance measure—TF-IDF—of 2,089,579 unique lemmas obtained from words in code reviews messages. We used a subset of the lemmas as features to build a Naïve Bayes classifier capable of differentiating between code reviews that are neutral and those that had missed a vulnerability. The average precision, recall, and F-measure of the classifier were 14%, 73%, and 23%, respectively.

We believe that our characterization of the linguistic features and the classifier will help developers identify potentially problematic code reviews before a vulnerability is missed.

A Comparing Distribution of Inquisitiveness, Sentiment and Complexity Metrics

The comparison of the distribution of inquisitiveness, sentiment and complexity metrics for neutral and missed vulnerability code reviews is shown in Figure 3.

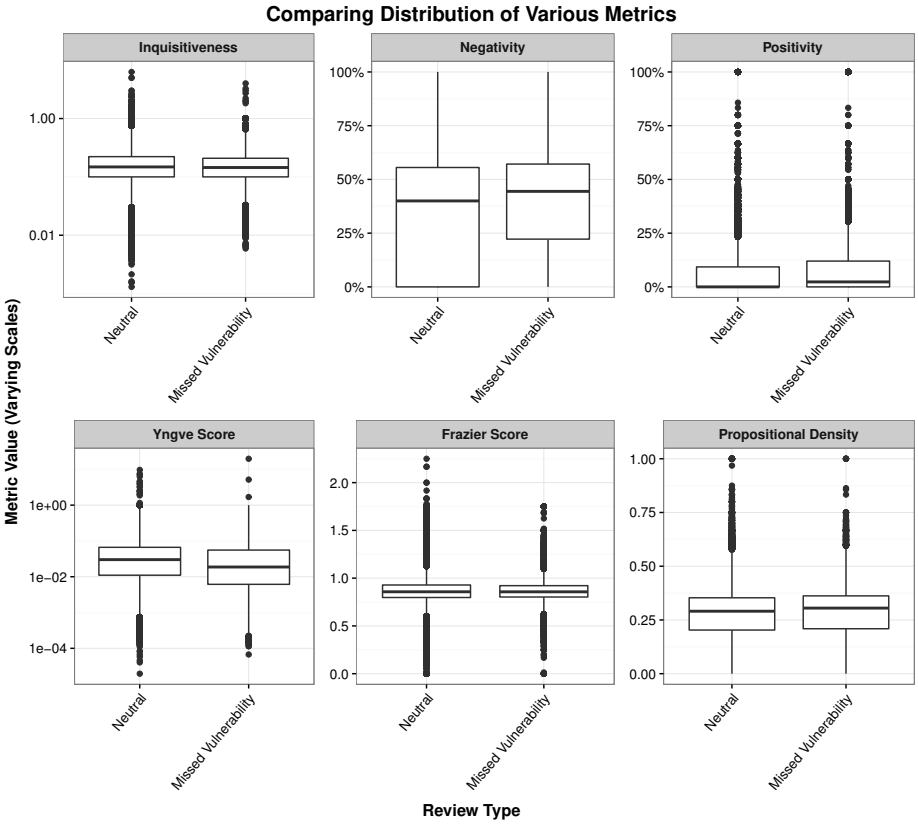

Fig. 3. Comparing the distribution of inquisitiveness, sentiment and complexity metrics for neutral and missed vulnerability code reviews

References

1. Reuters-21578, Distribution 1.0, `http://kdd.ics.uci.edu/databases/reuters21578/reuters21578.html`
2. Baddeley, A.: Recent developments in working memory. Current opinion in neurobiology 8(2), 234–238 (1998)
3. Baddeley, A.: Working memory and language: An overview. Journal of communication disorders 36(3), 189–208 (2003)
4. Baysal, O., Kononenko, O., Holmes, R., Godfrey, M.W.: The Influence of Non-technical Factors on Code Review. In: 2013 20th Working Conference on Reverse Engineering (WCRE). pp. 122–131 (Oct 2013)
5. Beller, M., Bacchelli, A., Zaidman, A., Juergens, E.: Modern Code Reviews in Open-source Projects: Which Problems Do They Fix? In: Proceedings of the 11th Working Conference on Mining Software Repositories. pp. 202–211. MSR 2014, ACM, New York, NY, USA (2014)
6. Bird, S., Klein, E., Loper, E.: Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit. O'Reilly Media, Inc. (2009)

7. Bosu, A., Carver, J.C.: Peer Code Review to Prevent Security Vulnerabilities: An Empirical Evaluation. In: 2013 IEEE Seventh International Conference on Software Security and Reliability Companion. pp. 229–230 (June 2013)
8. Bosu, A., Greiler, M., Bird, C.: Characteristics of Useful Code Reviews: An Empirical Study at Microsoft. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. pp. 146–156 (May 2015)
9. Bosu, A., Carver, J.C., Hafiz, M., Hilley, P., Janni, D.: Identifying the Characteristics of Vulnerable Code Changes: An Empirical Study. In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 257–268. FSE 2014, ACM, New York, NY (2014)
10. Brown, C., Snodgrass, T., Kemper, S.J., Herman, R., Covington, M.A.: Automatic Measurement of Propositional Idea Density from Part-of-Speech Tagging. Behavior Research Methods 40(2), 540–545 (2008)
11. Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P.: SMOTE: Synthetic Minority Over-sampling Technique. Journal of Artificial Intelligence Research 16, 321–357 (2002)
12. Chomsky, N.: Syntactic Structures. Mouton (1957)
13. Chromium: Chromium OS Developer's Guide (2017), `https://www.chromium.org/chromium-os/developer-guide`
14. Ciolkowski, M., Laitenberger, O., Biffl, S.: Software Reviews: The State of the Practice. IEEE Software 20(6), 46–51 (Nov 2003)
15. Czerwonka, J., Greiler, M., Tilford, J.: Code Reviews Do Not Find Bugs: How the Current Code Review Best Practice Slows Us Down. In: Proceedings of the 37th International Conference on Software Engineering - Volume 2. pp. 27–28. ICSE '15, IEEE Press, Piscataway, NJ, USA (2015), `http://dl.acm.org/citation.cfm?id=2819009.2819015`
16. Edmundson, A., Holtkamp, B., Rivera, E., Finifter, M., Mettler, A., Wagner, D.: An Empirical Study on the Effectiveness of Security Code Review pp. 197–212 (2013)
17. Francis, W.N., Kucera, H.: A Standard Corpus of Present-Day Edited American English, for use with Digital Computers. College English 26(4), 267 (1965)
18. Frazier, L.: Syntactic complexity. In: Dowty, D.R., Karttunen, L., Zwicky, A.M. (eds.) Natural language parsing, pp. 129–189. Cambridge University Press (CUP) (1985)
19. Frazier, L.: Sentence Processing: A Tutorial Review (1987)
20. Frazier, L.: Syntactic Processing: Evidence from Dutch. Natural Language & Linguistic Theory 5(4), 519–559 (1987)
21. Frazier, L., Taft, L., Roeper, T., Clifton, C., Ehrlich, K.: Parallel structure: A source of facilitation in sentence comprehension. Memory & Cognition 12(5), 421–430 (1984)
22. Guzman, E., Azócar, D., Li, Y.: Sentiment Analysis of Commit Comments in GitHub: An Empirical Study. In: Proceedings of the 11th Working Conference on Mining Software Repositories. pp. 352–355. MSR 2014, ACM, New York, NY (2014)
23. Hart, M.S., Austen, J., Blake, W., Burgess, T.W., Bryant, S.C., Carroll, L., Chesterton, G.K., Edgeworth, M., Melville, H., Milton, J., Shakespeare, W., Whitman, W., Bible, K.J.: Project Gutenberg Selections. Freely available as a corpus in the Natural Language ToolKit, `http://www.nltk.org/nltk_data/#25`
24. Hinkle, D.E., Wiersma, W., Jurs, S.G.: Applied Statistics for the Behavioral Sciences. Houghton Mifflin (2002)

25. Lipner, S.: The Trustworthy Computing Security Development Lifecycle. In: 20th Annual Computer Security Applications Conference. pp. 2–13 (Dec 2004)
26. Manning, C.D., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S.J., McClosky, D.: The Stanford CoreNLP Natural Language Processing Toolkit. In: Association for Computational Linguistics (ACL) System Demonstrations. pp. 55–60 (2014)
27. Mäntylä, M.V., Lassenius, C.: What Types of Defects Are Really Discovered in Code Reviews? IEEE Transactions on Software Engineering 35(3), 430–448 (May 2009)
28. Mayer, R.E., Moreno, R.: Nine Ways to Reduce Cognitive Load in Multimedia Learning. Educational psychologist 38(1), 43–52 (2003)
29. McGraw, G.: Software security. IEEE Security Privacy 2(2), 80–83 (Mar 2004)
30. Meneely, A., Srinivasan, H., Musa, A., Tejeda, A.R., Mokary, M., Spates, B.: When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits. In: 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. pp. 65–74 (Oct 2013)
31. Menzies, T., Greenwald, J., Menzies, T., Dekhtyar, A., Distefano, J., Greenwald, J.: Problems with Precision: A Response to "Comments on 'Data Mining Static Code Attributes to Learn Defect Predictors'" 33, 7–10 (Nov 2007)
32. Meyers, B.S.: Speech Processing & Linguistic Analysis Tool (SPLAT) (Jan 2017), `https://github.com/meyersbs/SPLAT`
33. Miller, G.: Human Memory and the Storage of Information. IRE Transactions on Information Theory 2(3), 129–137 (1956)
34. Miller, J.F., Chapman, R.S.: The Relation between Age and Mean Length of Utterance in Morphemes. Journal of Speech, Language, and Hearing Research 24(2), 154–161 (1981)
35. Pletea, D., Vasilescu, B., Serebrenik, A.: Security and Emotion: Sentiment Analysis of Security Discussions on GitHub. In: Proceedings of the 11th Working Conference on Mining Software Repositories. pp. 348–351. MSR 2014, ACM, New York, NY (2014)
36. R Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2015), `https://www.R-project.org/`
37. Roark, B., Mitchell, M., Hosom, J., Hollingshead, K., Kaye, J.: Spoken Language Derived Measures for Detecting Mild Cognitive Impairment. Trans. Audio, Speech and Lang. Proc. 19(7), 2081–2090 (10 2011)
38. Salton, G., Buckley, C.: Term-weighting Approaches in Automatic Text Retrieval. Information Processing & Management 24(5), 513–523 (1988)
39. Socher, R., Perelygin, A., Wu, J.Y., Chuang, J., Manning, C.D., Ng, A.Y., Potts, C.: Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank. In: Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing. Association for Computational Linguistics (October 2013)
40. Sweller, J., Chandler, P.: Evidence for cognitive load theory. Cognition and instruction 8(4), 351–362 (1991)
41. Yang, Y., Pedersen, J.O.: A Comparative Study on Feature Selection in Text Categorization. In: Icml. vol. 97, pp. 412–420 (1997)
42. Yngve, V.H.: A model and an Hypothesis for Language Structure. vol. 104, pp. 444–466. American Philosophical Society (1960)