Machine Learning Methods for Software Vulnerability Detection

Boris Chernis
University of Houston
Houston, TX
boris_chernis@yahoo.com

Dr. Rakesh Verma
University of Houston
Houston, TX
rverma@uh.edu

ABSTRACT

Software vulnerabilities are a primary concern in the IT security industry, as malicious hackers who discover these vulnerabilities can often exploit them for nefarious purposes. However, complex programs, particularly those written in a relatively low-level language like C, are difficult to fully scan for bugs, even when both manual and automated techniques are used. Since analyzing code and making sure it is securely written is proven to be a non-trivial task, both static analysis and dynamic analysis techniques have been heavily investigated, and this work focuses on the former.

The contribution of this paper is a demonstration of how it is possible to catch a large percentage of bugs by extracting text features from functions in C source code and analyzing them with a machine learning classifier. Relatively simple features (character count, character diversity, entropy, maximum nesting depth, arrow count, "if" count, "if" complexity, "while" count, and "for" count) were extracted from these functions, and so were complex features (character n-grams, word n-grams, and suffix trees). The simple features performed unexpectedly better compared to the complex features (74% accuracy compared to 69% accuracy).

CCS CONCEPTS

• **Software and its engineering** → **Automated static analysis**;
• **Security and privacy** → *Software security engineering*;

KEYWORDS

static analysis, buffer overflow, vulnerability detection, n-grams, suffix trees, software metrics, machine learning

ACM Reference Format:
Boris Chernis and Dr. Rakesh Verma. 2018. Machine Learning Methods for Software Vulnerability Detection. In *IWSPA'18: 4th ACM International Workshop on Security And Privacy Analytics, March 19–21, 2018, Tempe, AZ, USA,* Jennifer B. Sartor, Theo D'Hondt, and Wolfgang De Meuter (Eds.). ACM, New York, NY, USA, Article 4, 9 pages. https://doi.org/10.1145/3180445.3180453

1 INTRODUCTION

In the IT industry, software vulnerabilities are a big concern, especially in security-sensitive programs. Some of these bugs can be exploited for nefarious purposes when discovered by malicious attackers. In some cases, an attacker can crash an important running program, leading to a DoS (denial of service). In other cases, the attacker can escalate his privileges or even achieve full control over the machine.

Over the years, numerous countermeasures have been implemented in both compilers and operating system to minimize the damage that malicious hackers can cause via buffer overflow attacks. For example, DEP (data execution prevention) makes the call stack non-executable, preventing hackers from being able to execute their payloads, and ASLR (address space layout randomization), randomizes the address space layout of the process, making it more difficult for hackers to insert correct addresses into their payloads [17]. However, these techniques have proven to be little more than a nuisance to determined adversaries.

Thus far, the only way to prevent hackers from successfully completing an attack is to write secure code. However, complex programs, particularly those written in a relatively low-level language like C, are difficult to scan for bugs, even when using both manual and automated techniques. Microsoft spends roughly 100 machine years per year using automated techniques to detect bugs in their code [7], but their products often contain numerous bugs, because complex pointer arithmetic can sometimes be difficult to follow, especially when the developers are under constant time pressure to meet their deadlines. Since attackers use software to uncover security holes in programs, it is important for developers and security professionals to keep up with the latest automated vulnerability detection technologies.

The contribution of this paper is a methodology for analyzing features from C source code to classify functions as vulnerable or non-vulnerable. After finding 100 programs on GitHub, we parsed out all functions from these programs. We then extracted trivial features (function length, nesting depth, string entropy, etc) and non-trivial features (n-grams and suffix trees) from these functions. The statistics for these features were arranged in a table, which was split into training data and test data. Several different classifiers, including Naive Bayes, k nearest neighbors, k means, neural network, support vector machine, decision tree, and random forest, were used to classify the test samples. The trivial features produced the best classification result, with an accuracy of 75%, while the best n-grams result was 69% and the best suffix trees result was 60%. These results are discussed in more detail in Section 5. Section 2 discusses some background concepts, Section 3 discusses previous work, Section 4 outlines the details of the testing methodology, and Section 6 contains the conclusions.

IWSPA'18, March 19–21, 2018, Tempe, AZ, USA

ACM ISBN 978-1-4503-5634-3/18/03…$15.00
https://doi.org/10.1145/3180445.3180453

2 BACKGROUND INFORMATION

This section defines the terminology and concepts pertaining to some of the features used for classification and defines three types of security vulnerabilities that are mentioned in this paper.

2.1 Entropy

In the field of information theory, Shannon Entropy is the expected amount of information contained in a message, and it is given by the formula in Equation 1 below. When calculating the entropy of a string, $p(x_i)$ is the proportion of the string consisting of the i-th distinct character. In this study, entropy was used not only as a trivial feature, but also for calculating the "information gain" for each feature. A feature's information gain is the amount by which classification based strictly on that feature decreases the entropy associated with sample classes.

$$-\sum_{i=1}^{n} p(x_i) log(p(x_i)) \quad (1)$$

2.2 Suffix Trees

An important data structure for storing highly redundant text is called the "suffix tree" [15]. Let us consider the strings "good" and "hood". We start with the word "good" and extract all suffixes, which would be 1) "good", 2) "ood", 3) "od", and 4) "d". Next, we add each suffix to the tree (which initially consists of only the "root" node). When adding "good", we start by looking for a "g" attached to the root node. Since no "g" is found, we attach it, along with all the letters that follow. The same happens with the "ood" suffix. However, for the "od" suffix, we see an "o" attached to the root, so instead of creating a new "o", we follow the one that already exists. No "d" is attached to the "o", so we must attach a "d", and the "o" now has two branches. The final suffix is a "d", which we attach to the root. We continue the procedure with the four suffixes of "hood" and end up with the suffix tree shown in the figure below. We observe that each node has a blue number and a red number associated with it. The blue number is the "child frequency" (number of times it was traversed, including the time it was created), and the red number is the "parent frequency" (summation of the child frequencies of its children).

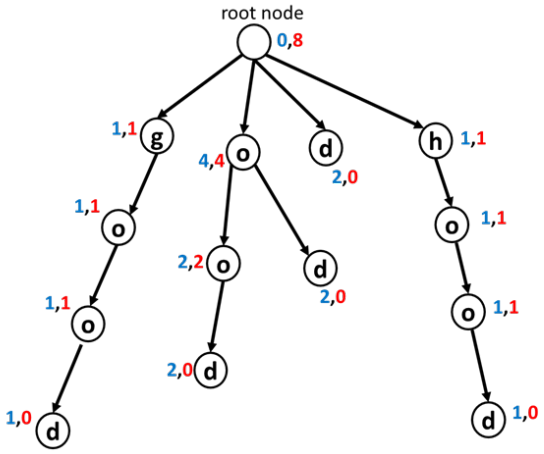

Figure 1: Suffix tree for "good" and "hood"

Research has shown that suffix trees can be used to classify e-mails as ham or spam, so in this study, we attempt to use suffix trees to classify functions as non-vulnerable or vulnerable. We construct two suffix trees from the training data: one for non-vulnerable functions and one for vulnerable functions. Then, to classify a test function as non-vulnerable or vulnerable, we score it against both trees and check whether the vulnerable score is higher than the non-vulnerable score multiplied by some pre-set threshold. Let us consider a function consisting of the word "hog" that we are trying to score against the tree in the above figure. Its score is the summation of all the scores of its suffixes. To score a suffix, we traverse it as far down the tree as it will go. At each node in the traversal, we divide that node's child score by its parent's parent score. The summation of these ratios is the score for the suffix. The suffix "hog" would score 1/8 ("h" node)+1/1("ho" node)=1.25. Next, the suffix "og" would score 4/8 ("o" node)=.5. Finally, the "g" suffix would score 1/8=.125. Summing these together, we get 1.875 as the total score for "hog".

2.3 Buffer Overflow Vulnerability

Sometimes, an attacker is able to give a vulnerable program a malicious input that corrupts the program's function call stack. In the simplest case, the attacker is able to inject a shellcode (code for spawning a root shell) into the stack and execute it by overwriting the return address in the vulnerable function's stack frame.

2.4 Integer Overflow Vulnerability

An integer overflow occurs when a mathematical operation attempts to assign an integer a larger value than it can represent. If the integer being targeted is responsible for a security-sensitive aspect of the program (like memory buffer allocation size), this can indirectly facilitate other memory corruption attacks, such as the buffer overflow.

2.5 Format String Vulnerability

This is when an attacker can inject a malicious format string to gain unauthorized access to memory contents. For example, if the C

code has a line that says "printf(str1)" and the attacker has control over str1, the attacker can inject a long series of "%x"s into str1 in order to get a hex dump of the memory.

3 PREVIOUS WORK

Although many different countermeasures exist that attempt to make programs crash in a fail-safe manner in the event a vulnerability is successfully exploited, they do not always work against determined hackers. Since manual vulnerability detection is difficult, many automated methods have been developed to assist programmers with this task. These methods fall into two main categories: static analysis (analyzing the code without running it) and dynamic analysis (analyzing the code in an operating environment). Since this work deals entirely with static analysis, several earlier static analysis approaches are discussed in this section.

A popular static analysis method in the relatively early days of security-sensitive computing was the grep utility, which was used with well-known functions (like strcpy and sprintf) that often result in format string vulnerabilities [19]. The issue with grep is its lack of flexibility, inability to detect features beyond regular expressions, and numerous false positives.

ITS4 ("It's the Software, Stupid! Security Scanner"), an improvement over grep, was able to offer real-time feedback during the development process [19]. ITS4 works by breaking the code up into "lexical tokens" (substrings that hold useful information about the code). It takes the resultant stream of tokens and compares it against a vulnerability database of "suspects," which was constructed from archives of known vulnerabilities. In addition to this tokenization, ITS4 is able to check for format string vulnerabilities and race conditions.

Another early static analysis tool is discussed in [11]. This tool analyzes annotations (special user notes embedded in comments and noticed by the tool, but not the compiler) that allow the developer to specify things like pre- and post-conditions for functions. Additionally, the tool uses "loop heuristics" by "taking advantage of the idioms used by typical C programmers."

Another static analysis tool, presented in [20], focuses on string-related bugs and attempts to trade off precision (avoiding false positives and false negatives) for scalability (ability to analyze large programs). It treats C strings as abstract data types (defined by behavior as opposed to the data they represent) and represents buffers as pairs of integer ranges (bytes allocated for the string and bytes currently in use). This tool detects buffer overflows by checking, for each string buffer, whether the allocated size is at least as large as the maximum possible length.

Later, another static analysis tool was presented in [5]. CSSV (C String Static Verifier) detects overflows (updates to memory beyond the bounds of allocated buffers), unsafe pointer arithmetic, references beyond null termination, unsafe library calls, and even multi-level pointers. It does so by translating the code from C to CoreC (a subset of C) and then combining user annotations with pointer analysis.

Another old tool, called "Splint" [6], also uses annotations. The authors claim it to be lightweight, trading off classification accuracy for performance. In addition to annotations, it also uses the same loop heuristics as [11] to determine whether a warning is to be issued[22].

Later on (2007), a static analysis tool called RICH (Runtime Integer CHecking) was developed to statically detect integer overflows [4]. RICH focuses on integer sub-types (8-bit, 16-bit, etc) and flags potentially unsafe operations, such as downcast (conversion to an integer type with fewer bits).

An earlier work worth mentioning that used source code metrics to classify object-oriented C++ code as vulnerable or non-vulnerable was published in 1996. The researchers used the following metrics for classification: 1) number of member functions and operators per class) 2) depth of the inheritance tree 3) number of children per class 4) "coupling between class objects" (whether objects use each other's functions or variables) 5) "response for a class" (number of methods in a class that can be executed because of a message the class received) 6) "lack of cohesion on methods" (number of pairs of functions with no variables in common minus the number of pairs of functions that have variables in common) [3].

Using code metrics to classify functions as vulnerable or non-vulnerable has been researched in the recent years as well. In 2014, a study was published where the authors extracted metrics and performed logistic regression on those metrics to classify functions [18]. The metrics they used were 1) "cyclomatic complexity" (number of decision statements plus one), 2) "nesting complexity" (maximum depth of control statements), 3) number of possible execution paths, 4) number of lines of code 5) number of lines of code excluding declaration, and 6) number of lines of executable statements.

Another study that used code metrics was published in 2007. The authors were able to find defects in the "Eclipse" code by extracting 198 code metrics and using logistic regression to predict whether a section of code contains bugs. They published these 198 metrics in a publicly-available dataset [21]. This was followed by another study, in 2008, that used 31 of those 198 metrics and referred to them as "product-related" (extracted directly from the code) [14]. The authors also extracted "process-related" metrics, which reflected changes, between different versions of the program, in the product-related metrics. In addition to logistic regression, they also used the Naive Bayes and decision tree classification methods, both on product-related and process-related metrics. They concluded that process-related metrics were, overall, better at indicating the presence or absence of software defects.

The idea that code changes can cause bugs was used in 2007 as well. A paper was written that introduced the concept of four different types of locality [10]. "Changed-entity" locality referred to the idea that recently modified functions are likely to contain new bugs, "new-entity locality" meant that recently added functions are likely to be buggy, "temporal locality" meant that if a function introduced a bug, it will likely introduce other bugs in the near future, and "spatial locality" meant that functions that interacted with a buggy function were also likely to contain bugs. Using these locality ideas, they constructed a "cache", consisting of 10% of the source code files that they were analyzing. Functions selected for this cache included functions pertaining to discovered bugs, functions that were changed during the bug fixes, recently added functions, and recently changed functions. The authors were able to use the cache

to predict bug locations fairly accurately, both at the source file level and at the function level.

In 2009, another static analysis tool that analyzes code changes was developed [9]. Their idea was that if a program undergoes modifications in numerous places (as opposed to a similar number of modifications limited to a few places), it becomes relatively difficult for developers to maintain a good grasp on what the program is doing. Also, a highly scattered code modification pattern might indicate that many developers are working on the code simultaneously, which also complicates things. How scattered the modifications to a program are can be measured in terms of Shannon Entropy. This is calculated by applying Equation 1 to the frequencies of source code segments (files or functions) getting changed over a pre-defined time period. Since it is possible to calculate the entropy of code changes as a function of time, we can predict that segments of code modified during a high-entropy period will be more likely to have issues.

In 2010, a static analysis tool was developed that also applies the entropy concept (Equation 1). Since the source code being analyzed is written in Java, they subdivide the code into classes and extract a feature metric (such as number of lines of code, number of methods, number of attributes, or number of other classes that reference the class) for each class. This is done for several versions of the program that are regularly spaced in time. They generate an n by m matrix, where the rows are classes and the columns are points in time. Then, they compute an element-by-element absolute-value difference between every two neighboring columns. This results in an n by m-1 difference matrix, with each column representing a time interval. For each time interval, they compute the entropy, relative to the column total. In addition to entropy, they compute a metric called "churn" by simply summing each column in the difference matrix. They noticed that both entropy and churn had strong positive correlations with the probability that a bug was introduced during a time interval.

Static analysis remains to be a popular vulnerability detection technique, so other static analysis tools were developed in the last three years. One of them, called "Stacy", uses "control flow analysis" [12]. It constructs a CFG ("control flow graph") from the program, where blocks of code are represented as nodes and dependencies are represented as edges. In the case of an "if" statement, the CFG branches. Stacy traverses all nodes of the CFG (representing all possible execution paths) and verifies that no variable is initialized using the value from an uninitialized variable. Another issue that Stacy detects using the CFG is memory leaks, which often occur during dynamic memory allocation operations, such as malloc. To do this, it traverses the CFG and ensures that for all possible execution paths, any dynamically allocated memory gets deallocated by the developer at some point. Finally, Stacy uses the CFG to detect buffer overflows by ensuring that for all possible execution paths, array accesses do not exceed buffer limits.

Another tool extracts the "abstract syntax tree", using the GCC compiler. It then compares this AST to other ASTs, extracted from well-known buffer overflow vulnerabilities. If there is enough similarity, it then investigates the function in question in more detail by following its data flow [13]. An additional tool that uses a similar approach is called CppCheck [16].

4 EXPERIMENT OVERVIEW

In this study, we developed a methodology for classifying C language subroutines as "vulnerable" or "not vulnerable". This was done in seven major steps, corresponding to the sub-sections of this section: 1) data collection, 2) parsing and randomization, 3) preprocessing, 4) feature extraction, 5) feature selection, and 6) classification

4.1 Data Collection

C programs containing well-known vulnerabilities were found by querying the NVD (National Vulnerability Database) [2] with the following search string: "buffer overflow github". This limited the search results to open-source programs with vulnerable code. 100 programs with documented vulnerabilities were collected in this manner. For each of these 100 programs, the vulnerability was fixed in the last version, so the difference between the last version and the second-to-last version was used to pinpoint the location of the vulnerability. If all the lines of code that differed between the last two versions were located in the same function, then this function was added to the set of vulnerable functions, and all other functions were added to the set of non-vulnerable functions. If the difference spanned more than one function, it was sometimes possible to pick the correct function by looking at the notes related to the commit. If the difference spanned multiple functions and no notes were left by the developer specifying which function contains the vulnerability, then no functions from the program were used in the study.

4.2 Parsing and Randomization

Next, we extracted the functions from each program. First, gcc was used to remove comments. Then, a python script was used to parse each subroutine into a separate file. 100 vulnerable functions (one from each program) and roughly 5000 non-vulnerable functions were extracted. In order to create the dataset, all 100 vulnerable functions and 100 non-vulnerable functions (randomly selected from the 5000) were used.

4.2.1 The "Mixed" Dataset. Since GitHub utilities potentially have numerous bugs that have not yet been discovered, we decided to also use functions from widely-used open-source Linux utilities like ls, cp, etc. Arguably, these functions are less likely to have bugs, because they have withstood the test of time. For purposes of sound experimental design, the top 20 most familiar-sounding Linux utilities were selected after filtering out utilities that had vulnerabilities show up in a Google or NVD search. Code from the following 20 utilities was downloaded from http://ftp.gnu.org: cat, cp, du, echo, head, kill, mkdir, nl, paste, rm, seq, shuf, sleep, sort, tail, touch, tr, uniq, wc, and whoami. As was expected, the code for each of these utilities consisted of multiple functions. Tests were run for both the mixed dataset (100 vulnerable functions from GitHub, 50 non-vulnerable functions from GitHub, and 50 non-vulnerable functions from the 20 Linux utilities) and the non-mixed dataset (100 vulnerable functions from GitHub and 100 non-vulnerable functions from GitHub)

4.3 Preprocessing

It was hypothesized that code contains a lot of information (such as constant literals and variable names) that might not benefit our classification at all, and could possibly even hurt it. Therefore, we attempted to discard this information before extracting feature statistics. Intuitively enough, we did not want to discard special characters, because they often contain useful information. For example, square brackets are often associated with array operations, which, according to [8], are almost always present in buffer overflow vulnerabilities. One idea is to simply discard all alphanumeric characters (referred to from now on as Method 1 preprocessing). In this context, "discard" means to remove, without replacing with spaces. However, since keywords of the C language consist of letters and potentially give us a lot of useful information, we can also attempt to preserve this information by globally applying a more sophisticated procedure (Method 2 preprocessing): 1) Discard all numbers except for 1 and 0. 0 and 1, generally speaking, appear more frequently in code than other constants, because programmers often check for things like binary conditions, whether the number of remaining items in a list is 0, etc.Of course, this can potentially have a drawback, in case there are numerous variable names like "x0" and "x1". 2) Replace the top 8 most frequent keywords appearing within the dataset with numbers 2-9. These top 8 keywords are "if", "int", "case", "return", "break", "struct", "char", and "else". This way, we preserve the useful information pertaining to these keywords, yet avoid having potentially useless n-grams (from things like variable names) potentially interfering with our classification. 3) Discard alphabetic characters. The next section details test results pertaining to data with no preprocessing, Method 1 preprocessing, and Method 2 preprocessing.

4.4 Feature Extraction

Various features were extracted, including trivial features (function length, nesting depth, string entropy, etc) and non-trivial features (n-grams and suffix trees). We know that a vulnerable function will oftentimes have complex pointer arithmetic, as well as complex and highly nested control structures. This means that a vulnerable function is likely to have a wide variety of special characters (leading to both high entropy and high character diversity), a high "maximum nesting depth", and a lot of "->"s. These are all important features discussed in Section 5.

4.5 Feature Selection

As described in Section 2, the suffix tree classifier was implemented as a "stand-alone" classifier. In other words, no statistics had to be fed into a separate machine learning classifier after suffix tree computation. N-gram statistics, on the other hand, had to be fed into a separate classifier, and the sheer number of n-grams was overwhelming, especially for word n-grams of size greater than two. Therefore, it was important to determine the best way to select which n-gram statistics to use. We decided to initially sort the n-grams, in descending order, for each n-gram type (words or chars) and length, by the amount of information gain that each n-gram gives us (Section 2.1). Later in this paper, something like "100 2-char n-grams" means "top 100 2-char n-grams, ranked by information gain". To avoid overfitting, we could only rank the n-grams based on the training data.

4.6 Classification

After extracting feature statistics and selecting the appropriate features, the data was split into "training" data and "test" data. Naive Bayes was used as the default classifier, but other classifiers were tested as well.

5 RESULTS AND ANALYSIS

Classification test results were generated in the eight steps, and each step corresponds to a sub-section of this section:

(1) Run initial classification tests using trivial features with the NaÃŕve Bayes classifier.
(2) Decide how to select n-grams, and test various combinations with the NaÃŕve Bayes classifier.
(3) Test suffix trees.
(4) See if additional feature selection via PCA improves the result.
(5) Compare the results of using different classifiers (other than NaÃŕve Bayes).
(6) Run a classification test on a "mixed" dataset.
(7) Perform an error analysis.

In the tables showing the results, TN, FN, TP, and FP refer to "true negative", "false negative", "true positive", and "false positive", respectively.

5.1 Trivial Feature Tests

Several trivial features were extracted from the functions, and each trivial feature was individually tested with 5-fold cross-validation. The classification results are summarized in the Table 1. Features 1, 2, and 3 have suffixes "a", "b", or "c", which mean no preprocessing, Method 1 preprocessing, and Method 2 preprocessing, respectively. Here, "character count" is the number of characters in a function, "character diversity" is the number of distinct characters, "string entropy" was discussed in Section 2, "max nesting depth" is the maximum nesting depth of the curly braces, and "average if complexity" is defined as (number of "else" + number of "else if") / number of "if". In the results, we notice that features 3a and 2c produce an accuracy of 74%, which is higher than what we see for all the other trivial features. When we tried all possible combinations (2^{15}-1=32767) of trivial features, the top five results had accuracies of 75-76%, and all combinations involved both features 3a and 2c. It is important to note that an accuracy of less than .55 is not significantly better than a coin flip (assuming we use an alpha level of 0.05).

Table 1: Trivial feature classification

ID	Feature	Accuracy	TN	FN	TP	FP
1a	Character Count	0.63	94	69	31	6
2a	Entropy	0.65	96	67	33	4
3a	Character Diversity	0.74	68	20	80	32
1b	Character Count	0.67	96	63	37	4
2b	Entropy	0.70	64	24	76	36
3b	Character Diversity	0.60	45	25	75	55
1c	Character Count	0.65	96	67	33	4
2c	Entropy	0.74	67	19	81	33
3c	Character Diversity	0.55	81	72	28	19
4	Max Nesting Depth	0.55	80	70	30	20
5	Arrow Count	0.60	93	74	26	7
6	If Count	0.50	78	78	22	22
7	If Complexity	0.63	93	68	32	7
8	While Count	0.65	92	63	37	8
9	For Count	0.57	96	82	18	4

5.2 N-grams Tests

When word n-grams of lengths 1-4 and character n-grams of lengths 1-5 were used for classification, the accuracy went as high as about 70% (see Appendices A and B). Because we were able to get a 74% accuracy from a single trivial feature and because n-grams are expected to contain a lot more information than a single value, this immediately raised a red flag. To address this, we performed an implementation verification by checking how well 1-grams and 2-grams could be used to classify e-mails (ham vs spam) that were randomly selected from the publicly-available Enron dataset[1]. The e-mail classification accuracy was around 90% (see Appendix A), so we concluded that the issue was in the method itself and not in the implementation. As an additional test, we combined all extracted n-grams of different lengths (see Table 2 below), but we were unable to get an accuracy of higher than 59%.

Table 2: Combinations of n-grams

Type	n	Preproc	Acc'y	TN	FN	TP	FP
Character	1-4	Method 1	0.56	8	2	48	42
Character	1-4	Method 2	0.55	7	2	48	43
Word	1-3	Method 1	0.59	11	2	48	39
Word	1-3	Method 2	0.59	11	2	48	39
Char+word	1-3	Method 1	0.57	9	2	48	41
Char+word	1-3	Method 2	0.54	6	2	48	44

Another test result, shown in Table 3 below, is for the combination of relatively small (and carefully chosen) numbers of n-grams of several types (word/char n-grams of lengths 1-5). For instance, we see that for 1-char n-grams, we only used the top 20 (selected by information gain, as mentioned earlier), for 2-char n-grams, we used only the top 75, etc. The number of n-grams (and preprocessing method) used for each n-gram type was subjectively chosen by examining plots of accuracy vs number of n-grams (like in Appendix A) and seeing what number of n-grams (and what preprocessing method) yields the best accuracy. These selections are listed in Table 4.

Due to the risk of overfitting, this cannot be done in real life, so the purpose of this test was strictly the "proof of concept". We wanted to check how much of an advantage we can get, in the best-case scenario, by carefully combining n-grams of many different types. We see that while character n-gram combinations showed some marginal level of improvement over using only one value of n (see Appendix B), word combinations showed a significant decrease in performance.

Table 3: Results for each combination of n-grams

	Acc'y	TN	FN	TP	FP
Char combos n = 1 through 5	0.69	23	4	46	27
Word combos n = 1 through 5	0.51	40	39	11	10
All char+word combos	0.68	25	7	43	25

Table 4: Number of n-grams selected for each type/length

N-gram type	Preprocessing	# Features
1c	Method 2	20
2c	Method 1	75
3c	Method 1	200
4c	Method 1	175
5c	Method 1	250
1w	No preprocessing	100
2w	No preprocessing	140
3w	Method 1	95
4w	Method 1	60
5w	Method 1	60

5.2.1 Cross-Validation and Combining with Trivial Features. When we ran the character combinations test with 2-fold cross-validation, the accuracy dropped to 63.5%. This was because the features were carefully selected to accurately classify the second half of the data, so it makes complete sense for them to not have worked as well on the first half of the data. This is why a closer look at the results showed a 69% accuracy for the second half of the samples and 58% accuracy for the first half.

5.3 Suffix Trees

The suffix tree classifier produced an even worse classification result than the n-grams classifier. Even with ideal parametrization, it only gave a 60% accuracy. As with the n-grams, it is important to show that the implementation was not the issue, so we once again compared function classification to e-mail classification. Just like n-grams, suffix trees were much better at classifying e-mails than they are at classifying functions (see figure below).

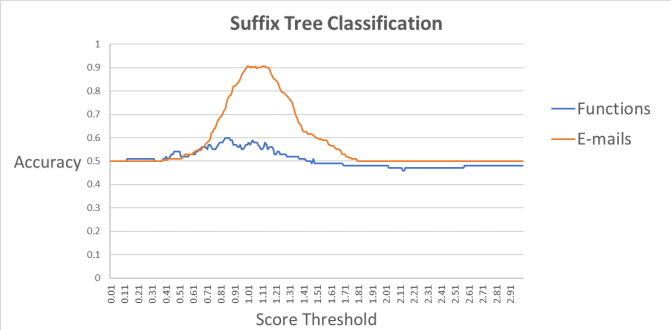

Figure 2: Suffix tree result: e-mails vs functions

5.4 Principle Component Analysis

Since the character n-gram combination gave a much better result than the word n-gram combination, it made sense to run a PCA test on it and determine whether the result could be improved further. When we ran PCA on the word n-grams combination, our accuracy dropped from 69% to 66%, although we were able to decrease the number of features from 718 to 195.

5.5 Test Results With Other Classifiers

Next, we took the PCA result and checked how the classifiers performed relative to each other. Since the Random Forest classifier gives a slightly different result each time it is run, the Random Forest result shown below is an average of 10 runs. We see that the K Means classifier performed as well as NaÃŕve Bayes for character diversity, and the SVM classifier performed as well as NaÃŕve Bayes for character n-grams. All classifiers were taken from the SciKit library, and they were all run with their default parameters. No preprocessing was used for any of these tests, and results are summarized in the table below. All tests ran successfully, except for neural networks for the character diversity feature (this is denoted by the "X").

Table 5: Accuracy results for different classifiers

Classifier	Char Diversity	Char n-grams combo
Naive Bayes	0.74	0.66
K Nearest Neighbors	0.67	0.54
K Means	0.75	0.55
Neural Network	X	0.57
Support Vector Machine	0.6	0.67
Decision Tree	0.72	0.59
Random Forest	0.72	0.58

5.6 Testing the "Mixed" Dataset

Next, we tested the mixed dataset with a) trivial features 3a and 2c and b) our character n-grams combo. The results are in the table below. We see that for the character n-grams combinations, the accuracy is comparable (67% instead of 69%). For the trivial features, however, the accuracy is significantly lower (63.5% instead of 75%). Closer examination of the trivial features test result showed that the buggy functions were easier to tell apart from the Linux functions (69% accuracy) than non-buggy functions from GitHub (58% accuracy). Finally, when the trivial features 3a and 2c were combined with character n-gram combinations, we had a slight increase in accuracy (69% compared to 67%).

Table 6: Mixed dataset results

	Acc'y	TN	FN	TP	FP
Features 3a and 2c	0.635	45	18	82	55
Character n-gram combos	0.67	54	20	80	46
Features 3a, 2c + char n-grams	0.69	70	32	68	30

5.7 Error Analysis

We attempted to find patterns pertaining to misclassified samples (false positives and false negatives). In order to perform error analysis, n-gram statistics from the best n-grams combination we found (69% accuracy) were analyzed. For each feature, we calculated its mean value over all samples. This was done separately for true negatives, false positives, true positives, and false negatives. When the mean values were plotted for all samples, we noticed that the points plotted for false positives had far more "zero" values (points touching the x-axis) than those plotted for true positives (see Appendix C). This pattern can potentially be exploited in a further study. Namely, we can run an n-grams classifier and follow it up with another classifier that attempts to reduce false positives by re-classifying samples marked as positives based on the number of zero-frequency n-grams that these samples have.

6 CONCLUSIONS AND RECOMMENDATIONS

After extensively testing function vulnerability classification using trivial features, n-grams, and suffix trees, we can draw several conclusions. First of all, we see that extracting numerous n-grams does not, thus far, seem to give good classification results, especially if we consider the 74% accuracy that we got from "character diversity" to be a baseline requirement. We also noticed that even when combinations of n-grams were manually selected (in a manner that would normally be illegal and lead to overfitting), the overall result did not improve. However, this study is a good proof-of-concept of a very important point: trivial features can tell us a lot about whether a function is vulnerable or not.

There are a few directions in which this research can be taken to improve the results further. First of all, it might be possible to think of additional trivial features to investigate. Secondly, it might also make sense to test some other n-gram selection techniques, as well as some of the SciKit library's other classification parameters (other than default settings). Third, it would be interesting to look more closely at which characters (or strings) are most important, since this would give us better insight than just "character diversity". One way to do this would be to run the same character diversity tests after eliminating different strings (square brackets, curly brackets, ++, etc) in pre-processing. Finally, it is possible to test whether the techniques presented in this paper can be used to efficiently detect vulnerabilities in other programming languages (in addition to C).

ACKNOWLEDGMENTS

This research was supported in part by NSF grants DUE 1241772, CNS 1319212 and DGE 1433817.

REFERENCES

[1] Enron email dataset. https://www.cs.cmu.edu/~enron/. Accessed: 2017-07-01.
[2] National vulnerability database. https://nvd.nist.gov. Accessed: 2017-07-01.
[3] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on software engineering*, 22(10):751–761, 1996.
[4] D. Brumley, T.-c. Chiueh, R. Johnson, H. Lin, and D. Song. Rich: Automatically protecting against integer-based vulnerabilities. *Department of Electrical and Computing Engineering*, page 28, 2007.
[5] N. Dor, M. Rodeh, and M. Sagiv. Cssv: Towards a realistic tool for statically detecting all buffer overflows in c. In *ACM Sigplan Notices*, volume 38, pages 155–167. ACM, 2003.
[6] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE software*, 19(1):42–51, 2002.
[7] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.
[8] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *USENIX Security Symposium*, pages 49–64, 2013.
[9] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88. IEEE Computer Society, 2009.
[10] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, pages 489–498. IEEE Computer Society, 2007.
[11] D. Larochelle, D. Evans, et al. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security Symposium*, volume 32. Washington DC, 2001.
[12] P. Lathar, R. Shah, and K. Srinivasa. Stacy-static code analysis for enhanced vulnerability detection. *Cogent Engineering*, 4(1):1335470, 2017.
[13] R. Ma, Y. Yan, L. Wang, C. Hu, and J. Xue. Static buffer overflow detection for c/c++ source code based on abstract syntax tree. *Journal of Residuals Science & Technology*, 13(6), 2016.
[14] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering*, pages 181–190. ACM, 2008.
[15] R. M. Pampapathi, B. G. Mirkin, and M. Levene. A suffix tree approach to anti-spam email filtering. *Machine Learning*, 65(1):309–338, 2006.
[16] E. Penttilä et al. Improving c++ software quality with static code analysis. *N/A*, 2014.
[17] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.
[18] Y. Shin and L. Williams. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 315–317. ACM, 2008.
[19] J. Viega, J.-T. Bloch, Y. Kohno, and G. McGraw. Its4: A static vulnerability scanner for c and c++ code. In *Computer Security Applications, 2000. ACSAC'00. 16th Annual Conference*, pages 257–267. IEEE, 2000.
[20] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*, pages 2000–02, 2000.
[21] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proceedings of the third international workshop on predictor models in software engineering*, page 9. IEEE Computer Society, 2007.
[22] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 97–106. ACM, 2004.

APPENDIX A: ONE-GRAM AND TWO-GRAM TESTS

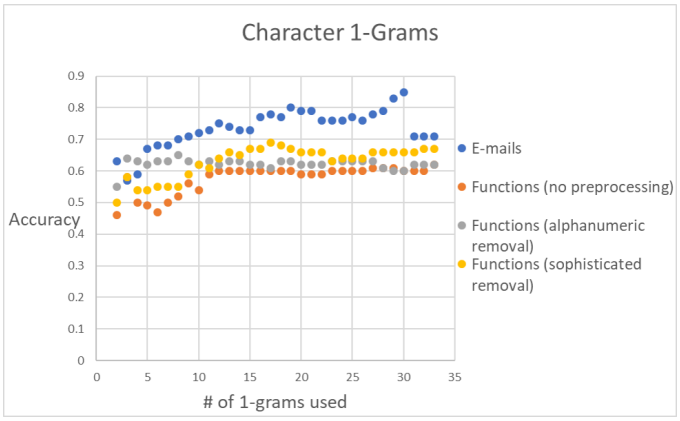

Figure 3: Character 1-grams (e-mails vs functions)

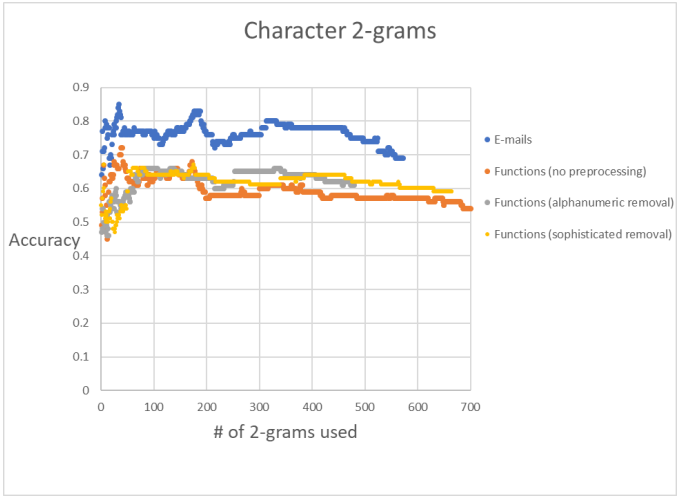

Figure 4: Character 2-grams (e-mails vs functions)

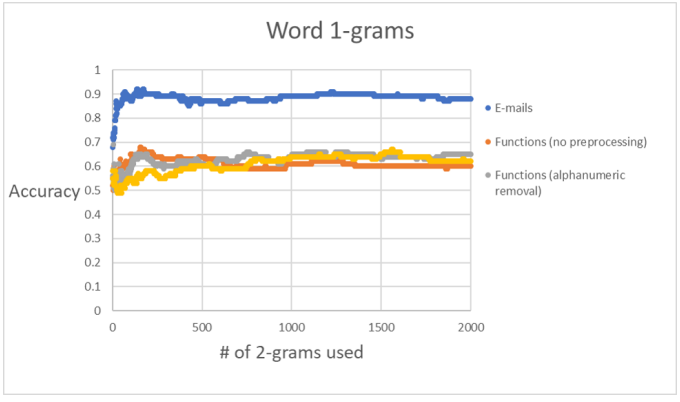

Figure 5: Word 1-grams (e-mails vs functions)

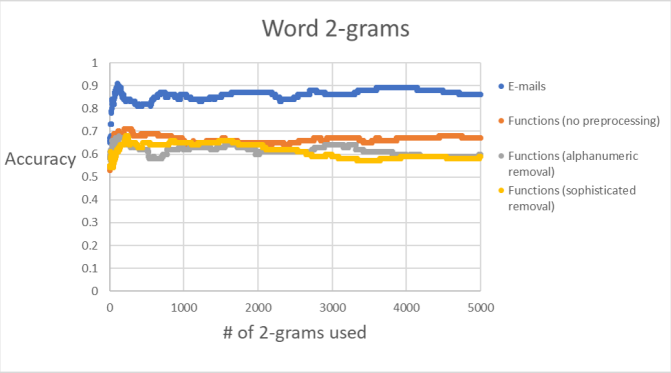

Figure 6: Word 2-grams (e-mails vs functions)

APPENDIX B: TESTING N-GRAMS OF DIFFERENT LENGTHS

Note: The rows with "X"s signify that the machine crashed during the classification. Also, the "preprocessing" column denotes "no preprocessing", "Method 1 preprocessing", and "Method 2 preprocessing" as 0, 1, and 2, respectively.

Table 7: Character n-grams of lengths 1-5

n	Pre-processing	Accuracy	TN	FN	TP	FP
1	0	0.64	34	20	30	16
1	1	0.62	22	10	40	28
1	2	0.62	25	13	37	25
2	0	0.53	9	6	44	41
2	1	0.62	22	10	40	28
2	2	0.59	19	10	40	31
3	0	X	X	X	X	X
3	1	0.60	14	4	46	36
3	2	0.56	9	3	47	41
4	0	X	X	X	X	X
4	1	0.56	9	3	47	41
4	2	0.58	10	2	48	40
5	0	X	X	X	X	X
5	1	0.55	8	3	47	42
5	2	0.59	11	2	48	39

Table 8: Word n-grams of lengths 1-4

n	Pre-processing	Accuracy	TN	FN	TP	FP
1	0	0.59	19	10	40	31
1	1	0.64	19	5	45	31
1	2	0.62	17	5	45	33
2	0	0.66	23	7	43	27
2	1	0.60	15	5	45	35
2	2	0.59	12	3	47	38
3	0	X	X	X	X	X
3	1	0.60	15	5	45	35
3	2	0.61	15	4	46	35
4	0	X	X	X	X	X
4	1	0.64	25	11	39	25
4	2	0.63	23	10	40	27

APPENDIX C: ERROR ANALYSIS

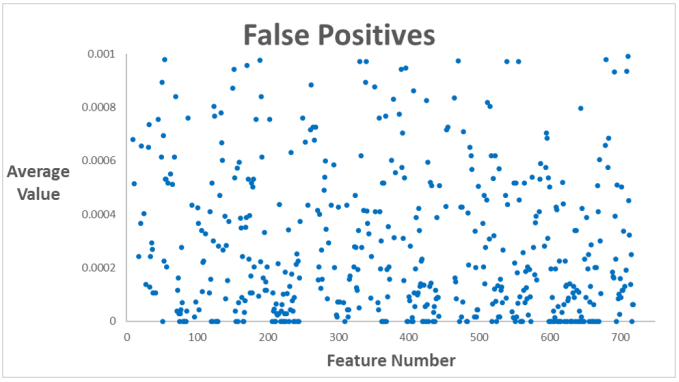

Figure 7: Error Analysis: False Positives

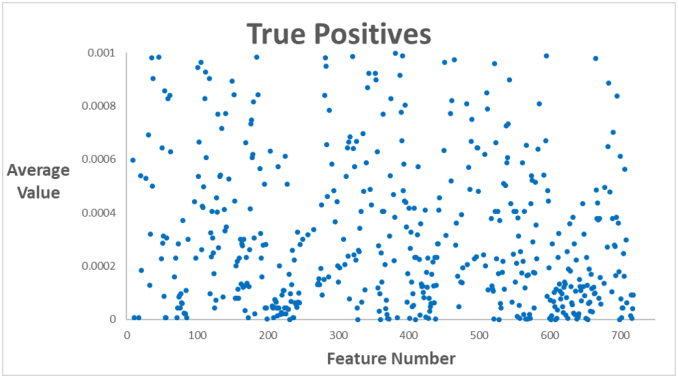

Figure 8: Error Analysis: True Positives