Detection of Malware through Statistical Analysis of Source Code

Stephanie Lukin, Computer Science

Dr. Dawn Lawrie and Dr. Dave Binkley (Computer Science)

Sponsored by the Hauber Fellowship Fund Loyola University Maryland – Summer 2009

I. Introduction

Software forensics claim malware looks different from clean code. Previous code analysis has not been able to identify malware from looking at the structure of a program alone. We believe that these ‘differences’ that distinguish malware from clean code, are reflected in the language of the code. Therefore in our research we examine the source code of a program to see if malware can be detected. In particular, we can examine the identifiers, comments, and strings of the source code because the programmer has freedom of naming in these three fields. To analyze the code, comment, and string sections of the source code, we will count the character frequencies of these fields, producing unigrams, bigrams, and trigrams. We will then create probability distributions and smooth these results. Finally, we perform statistical methods, specifically the Kullbeck-Leibler divergence on the probability distributions. Our goal is to determine if we can detect malicious code through inconsistencies in the Kullbeck-Leibler distribution data. In addition, our test hypotheses are that 1) the bigram distribution will be the most useful in helping us identify malicious code, and that the unigrams will not display enough variability and the trigrams will display too much variability and 2) the code will be more of an indicator of malware than the comments.

II. Background

Our analysis of the language of the code brings us to examine source code (Figure 1). Source code is a collection of statements and declarations written in some programming language that allows the programmer to communicate with the computer through a series of instructions. These instructions are written in a text file and together create a program. In all source code, no matter what language in which it is written, there are two distinct characteristics, comments and identifiers. The focus of our analysis will be on these two aspects. Our program has been adapted to include the examination of string literals, but have been excluded from the primary study and will not be heavily focused on for the duration of the paper. Identifiers are like variables in mathematics. The programmer has the freedom of naming an identifier to be anything he or she chooses. Comments are a programmers' way of making notes in the margin of a text book. They remind the programmer of what is happening in the code, especially in very large pieces of code, and they also help others who are unfamiliar with the code to understand it.

III. Our Approach

Refer to figure 3 for the detailed program flow

A. Tagger

Rather than study all sections of the source code, for there is much more to source code than comments and identifiers, we look specifically at these comments and identifiers because we believe they can give us the best insight into program comprehension. To begin our examination of the language of the code, we first "tag" all of the source code we wish to examine. The Tagger is a program which automatically extracts the comments and identifiers from source code, making the process of examining them much easier for our program. The output of the Tagger can be seen in figure 2. The Tagger is a program developed previously and independently of this research, but because of its flexibility we could use it to facilitate our own new purposes.

B. n-Gram Extractor

Next we examine the extracted comments and identifiers using the program we built, the n-Gram Extractor. For our purposes, we let n be 1, 2, or 3 such that a single character is a unigram (1-gram), two characters in a row are a bigram (2-gram), and three characters are a trigram (3-

gram). At the start of the program, the user can enter what he or she wants to be calculated. The options are code, comments, both combined, strings, or all. Additionally, the user can choose to examine the unigrams, bigrams, trigrams or all. Depending upon the user's choice of what he or she wants to examine, the program seeks to the appropriate block of tagged source code, i.e. the comment and/or code blocks, and counts the n-gram character patterns.

The n-Gram Extractor can examine a directory and all of its contents, i.e. a program, a single file, or a single function of a file. If the input is a directory, the n-Gram Extractor will calculate the pattern frequencies for each individual file of the directory and then sum the results together to achieve a total frequency for the entire set of files. After all the frequencies have been calculated, the program will output the resulting patterns and their corresponding frequencies to a file unique to the inputted tagged source code, in the directory case, the output files will be based on the directory as a whole.

Our program can handle source code that has been written in C, C++, or Java. While the Tagger will tag regardless of language (it has its own rules for tagging), the specific language becomes important when extracting the identifiers. We have gathered a list of what we call Stop Words in these three languages. Stop Words are words that are unique and are special identifiers in a language. For example, *int* is a Stop Word in C, C++, and Java. We do not want to incorporate the generic words of the language in our examination because they are not named by the author of the code. Therefore, when we come across a Stop Word in the code block, we ignore it.

C. Probability Distributions

Once we have calculated these n-gram frequencies, we feed the data into the statistical program we built. First, we calculate the probability distribution over a given data set. This can be easily calculated by dividing each individual frequency by the total number of pattern occurrences in the current data set. However, to use only this simple division is a naïve approach; the distribution must be supplemented to account for all possible patterns. Our approach implements a technique which smoothes the probability distribution data. By adding a small probability to all possible patterns, those with an observed frequency and those without, we ensure that it is possible to observe a particular pattern, but that it was not observed in the current data set. The primary reason for the smoothing will become more apparent in the calculation of the Kullbeck-Leibler divergence numbers. Once we have our smoothed probability distributions, we are able to compute statistical analyses on our distributions.

D. Kullbeck-Leibler Divergence

For our research purposes, we have chosen to compute Kullbeck-Leibler divergences. These numbers give us the degree of difference between two given probability distributions. While there are other statistical methods of comparing probability distributions, Kullbeck-Leibler divergence is most frequently used in information retrieval and natural language processing. These numbers will always be greater than or equal to zero. The closer the number is to zero, the less difference exists between the two probability distributions in question. If the number is zero, then the probability distributions are exactly the same; there is no difference or dissimilarity between the two. On the other hand, the higher the Kullbeck-Leibler divergence number, the more the probability distributions are different from each other.

The formula for calculating Kullbeck-Leibler divergence numbers is as follows:

$$KLd(p,q) = \sum p(x) \cdot \log\left(\frac{p(x)}{q(x)}\right)$$

In order to calculate these numbers, we define a Q and P. We let Q represent our background, and let P represent the thing we want to compare to the background. In the case of our research, we can let Q be a list of files or a single file. We allow P to be a file when Q is a list of files, and a function when Q is a file. In this way, we are comparing the smaller part P against the larger background Q.

Kullbeck-Leibler divergence deals directly with probability distributions, so that p(x) is the probability of a pattern x appearing in P, and q(x) is the probability of that same pattern x appearing in Q. For each pattern x that appears in P, this formula takes the log of the ratio of p(x) over q(x) and multiplies it by p(x). These individual contributions are then summed up to calculate our final Kullbeck-Leibler divergence number for the specific P and Q.

This is where the importance of having smooth data comes into play. In the instance where P is not a subset of Q, Q may contain a pattern x that does not appear in P. Due to the nature of the Kullbeck-Leibler divergence formula, this could create a 0 in the denominator of the p(x)-q(x) ratio. But since we smoothed the probability distributions, we know we will never have a probability distribution of 0, since we account for all possible patterns, even those that are not observed.

Furthermore, if the Kullbeck-Leibler divergence produces a 0 as a result, we know it is because there exists no difference between P and Q, and not that there was an unobserved pattern in P. We know that p(x) can never be zero because of the smoothed date. Then if the distributions are the same, the p(x)-q(x) ratio will be 1. The log(1) is 0, so no matter what p(x) is, the Kullbeck-Leibler divergence is 0. We guarantee that a result of 0 means that there is no difference, and not a questionable ‘was it simply not observed?’

IV. Results

A. Unigram, Bigram, Trigram Percentages

From our earlier steps of calculating frequencies using our n-Gram Extractor, we have produced the top 10 most frequent characters and a chart of all observed patterns for the code and comments of our entire collection of data. In figures 4 and 5 we see the unigram data, figures 6 and 7 show the bigram data, and the trigram data is seen in figures 8 and 9. Rather than just display the raw frequency of each observed pattern for our data set, we calculated the percentage of which an individual pattern was observed. In this way, we could compare one data set’s percentage distribution to another data set’s percentage distribution. In this specific test, we compare the code and comment distribution to the natural language distribution.

In the unigram data, we see that both the code and comment probability distribution mirror the distribution of the natural language patterns. Of the top 10 most common characters in the natural language, 9 are also in the top 10 of the code and comment distribution. There are some outstanding differences, such as the use of the ‘*’ in the comment distribution, but this is a character unique to the standard nomenclature of comments.

The code and comment bigram probability distributions show an overall very similar trend to the natural language distribution, again with the major exception of special stylistic comment characters. Upon closer examination, we can see that the comments more closely mirror the natural language distribution than the code. We are beginning to see how similar the comments are to the natural language, since comments are most often written in a cohesive manner, and how the code, while still following a similar trend, begins to differ. At this point, we notice that 6 of the top 10 natural language patterns appear in the most common comment patterns, but only 4 appear in the most common code patterns.

Finally in the code and comment trigram probability distribution, we observe a similar overall trend in the patterns observed, but many of the patterns that frequently appear in the natural language have a lower probability of appearing in the code. We can see that the probability of the comment patterns is closer to the probability of the natural language patterns. If we exclude stylistic patterns in the comment, such as '***', '---', and '===', we see that 8 patterns are in the top 10 most frequent natural language patterns. Only 3 patterns in the code appear in the top 10 of the natural language patterns.

B. Authorship Test

Due to the nature of our methodology for detecting source code, we can expand our usage of our tool. The Kullbeck-Leibler divergence test can also be used as a test for authorship. There are two distinct tests that can be run and can best be described through an example.

Let us assume we have three programmers: Alice, Bob, and Malroy, who each have a set of x files they authored. In the first test, we execute our process three times, one for each author. First we extract the source code into tagged source code, use our n-Gram Extractor to count character frequencies, create a smooth probability distribution, and then calculate the Kullbeck-Leibler divergence numbers. For this test, we let the background Q is the same for each author, for example the background is Alice's files, letting P be the author's own distinct files. So each Kullbeck-Leibler divergence test is as follows:

Author	**Q - background**	**P – to compare to background**
Alice	Alice	Alice’s files
Bob	Alice	Bob’s files
Malroy	Alice	Malroy’s files

The results for this test are seen in *figure 11*. We see that the line representing Alice is on the bottom. This is confirming that Alice’s files are the most similar to Alice’s background.

The first authorship test gives us a confirmation rather than something interesting to study. That is where the second authorship test is better used. We let P be the same for each author, in our case Alice’s files, and have our background Q change with the author. The test is as follows:

Author	**Q - background**	**P – to compare to background**
Alice	Alice	Alice’s files
Bob	Bob	Alice’s files
Malroy	Malroy	Alice’s files

In *figure 10*, we see there is much more to study about the results. This tells us that Alice’s files are most similar to Alice’s background, confirming Alice as the rightful author of the files used as P.

This test is best be used to confirm the authorship of code whose author is unknown. A test could be produced such that:

Author	**Q - background**	**P – to compare to background**
Alice	Alice	Unknown author’s files

Bob	Bob	Unknown author’s files
Malroy	Malroy	Unknown author’s files

If the author of the code is one of these three authors, then the line in the graph that represents that specific author will be on the bottom in a steady decline, not jagged and all over the place. This indicates that these files are most like this particular author. We found our technique to be valid 90% of the time.

V. Threats to Validity

So far in our research, we have only performed one statistical test and we claim that we are accurately identifying code that looks ‘different’ from the rest of the code surrounding it. This test is the Kullbeck-Leibler divergence test. Although our results are very promising, additional statistical tests should be performed on the same data sets to ensure that our test is not just a fluke. Furthermore, we have not yet had the opportunity to test our program on any code we know to be malicious. When we can get this code, we will then have a control group- code we know is clean or code we know is malicious. Further tests will be discussed in the Future Work section.

VI. Future Work

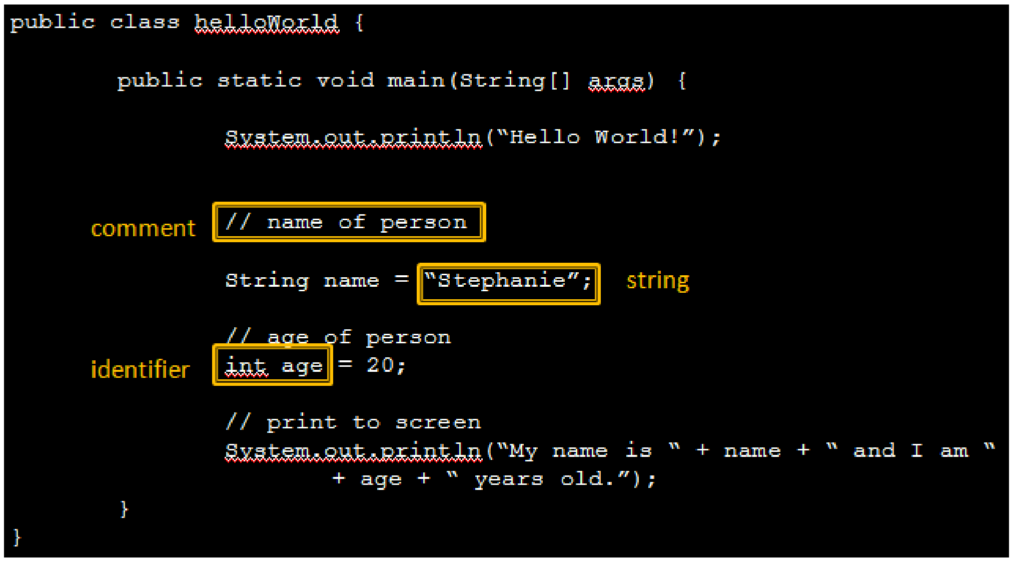
There are many different statistical methods of comparing probability distributions. The Kullbeck-Leibler divergence test was chosen because it was specifically developed for the purposes of natural language examination. In the future, it would behoove us to perform a different test on our probability data to reinforce that our approach is correct. A popular statistician test is Chi Squared, which we would use to compare two probability distributions, P and Q. We would notice whether or not we get the same numbers, and if we do not get the same numbers, does it mean that our test is inefficient? Or would the degree of dissimilarity between P and Q be the same due to the difference in interpreting Chi Squared and Kullbeck-Leibler divergence.

So far we have only been working with code we know is clean or that has been written by our research group or members of the second year Computer Science students at our University. This diverse group of authors has been extremely beneficial for the authorship test however we do not have any real malicious or obfuscated code to test. One possibility of obtaining real code outside of our testing data sets is to use a code obfuscator. Since we would be performing the obfuscation ourselves, we would have the original non-obfuscated code. After performing our tests, we could manually examine the clean code versus the obfuscated code to determine if our program is able to tell that there have been differences where the code was indeed changed.

Another possibility which is closer to fruition than the code obfuscator is a partnership with Northrop Grumman. A few months ago, we visited a representative from the Cyber Tech division and discussed the possibility of a partnership where we would be supplied with code that is known to be malicious. We would run the malicious code through our program and manually check to determine if our program is indeed flagging the correct sections.

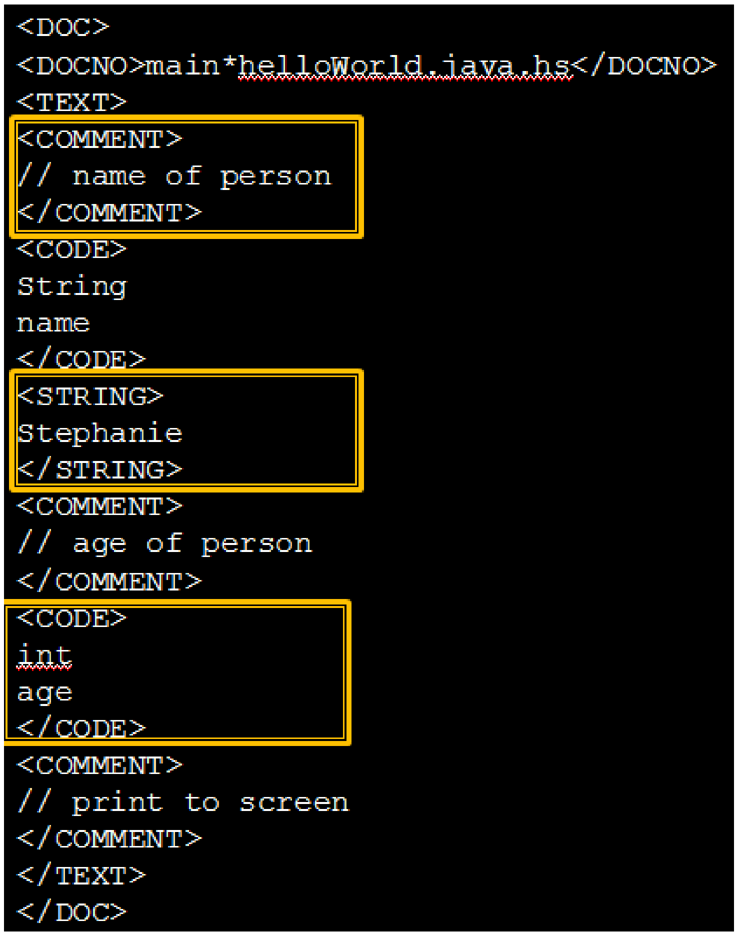
We also need more concrete evidence to conclude if our test hypotheses are true. In order to do so, we need more unigram, bigram, and trigram distributions over diverse groups. We also want to compare the code and comments and determine if one is a greater significance than the other when calculating these Kullbeck-Leibler Divergence numbers. When we gather additional data and test sets, we will look for relationships between the data, such as the presence of a power law.

Figure 1: Source Code



```
public class helloWorld {

        public static void main(String[] args) {

                System.out.println("Hello World!");

                // name of person

                String name = "Stephanie";

                // age of person
                int age = 20;

                // print to screen
                System.out.println("My name is " + name + " and I am "
                        + age + " years old.");
        }
}
```

Figure 2: Tagged Source Code



```
<DOC>
<DOCNO>main*helloWorld.java.hs</DOCNO>
<TEXT>
<COMMENT>
// name of person
</COMMENT>
<CODE>
String
name
</CODE>
<STRING>
Stephanie
</STRING>
<COMMENT>
// age of person
</COMMENT>
<CODE>
int
age
</CODE>
<COMMENT>
// print to screen
</COMMENT>
</TEXT>
</DOC>
```

Figure 3: Program flow

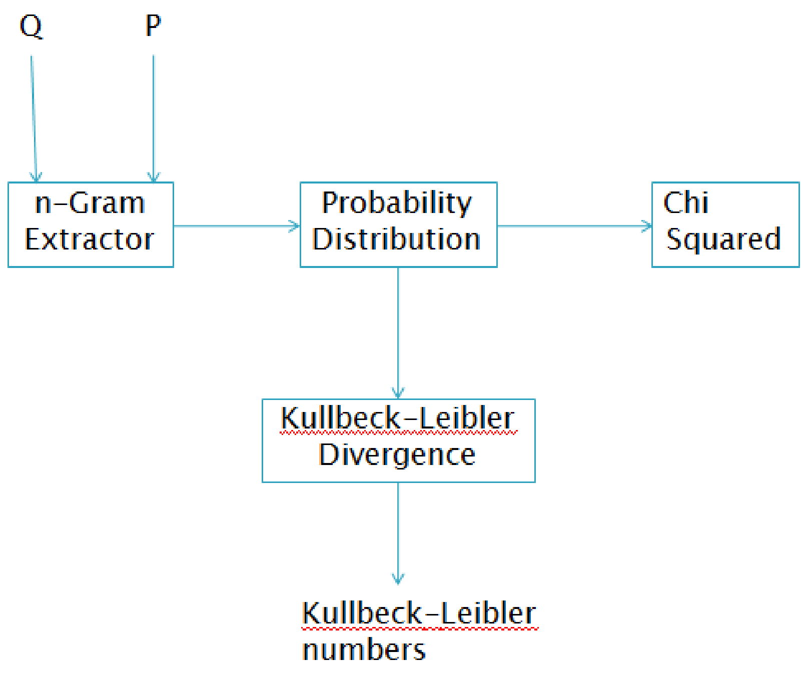


Figure 4: Unigram Probability Distributions

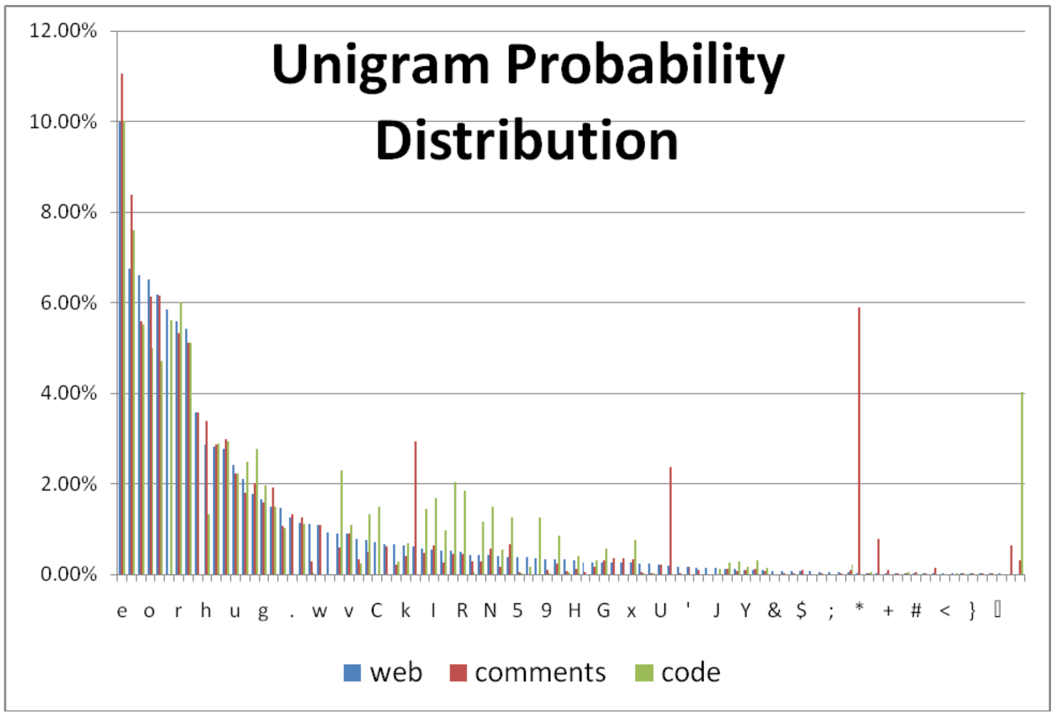
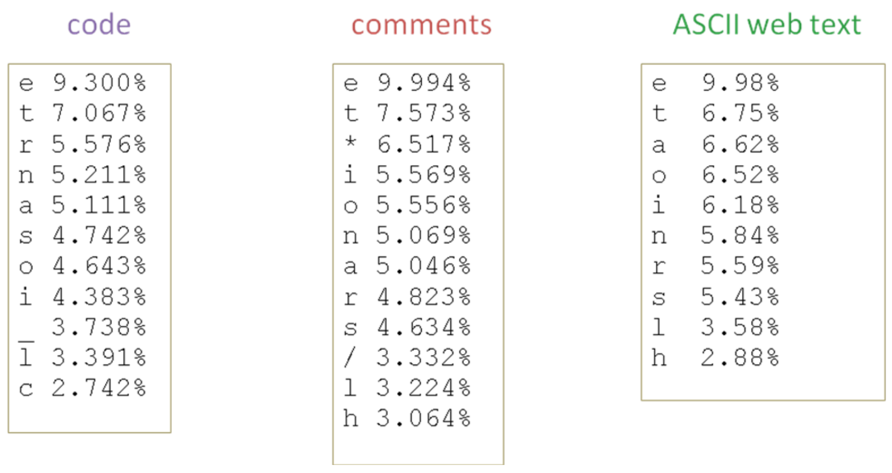


Figure 5: Most Frequent Unigrams



code		comments		ASCII web text	
e	9.300%	e	9.994%	e	9.98%
t	7.067%	t	7.573%	t	6.75%
r	5.576%	*	6.517%	a	6.62%
n	5.211%	i	5.569%	o	6.52%
a	5.111%	o	5.556%	i	6.18%
s	4.742%	n	5.069%	n	5.84%
o	4.643%	a	5.046%	r	5.59%
i	4.383%	r	4.823%	s	5.43%
_	3.738%	s	4.634%	l	3.58%
l	3.391%	/	3.332%	h	2.88%
c	2.742%	l	3.224%		
		h	3.064%		

Figure 6: Bigram Probability Distributions

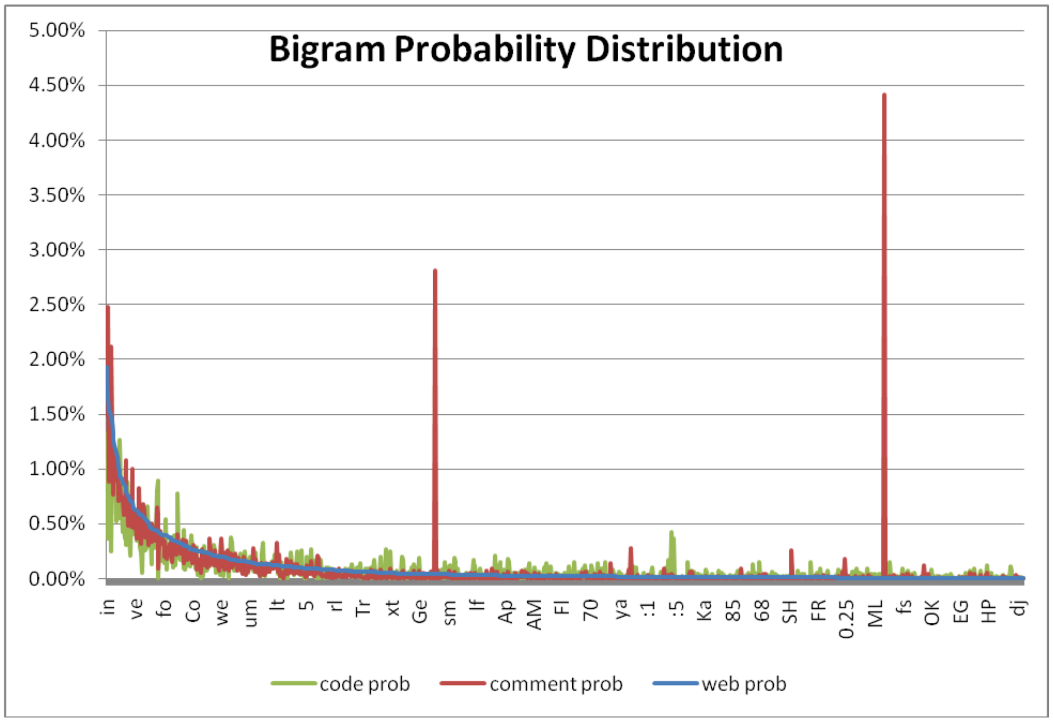
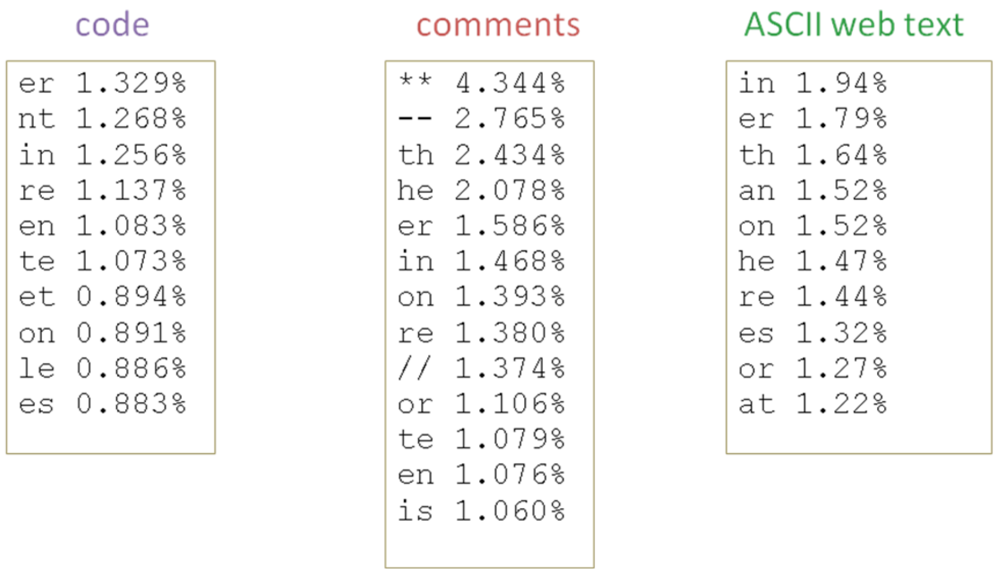


Figure 7: Most Frequent Bigrams



code	comments	ASCII web text
er 1.329%	** 4.344%	in 1.94%
nt 1.268%	-- 2.765%	er 1.79%
in 1.256%	th 2.434%	th 1.64%
re 1.137%	he 2.078%	an 1.52%
en 1.083%	er 1.586%	on 1.52%
te 1.073%	in 1.468%	he 1.47%
et 0.894%	on 1.393%	re 1.44%
on 0.891%	re 1.380%	es 1.32%
le 0.886%	// 1.374%	or 1.27%
es 0.883%	or 1.106%	at 1.22%
	te 1.079%	
	en 1.076%	
	is 1.060%	

Figure 8: Trigram Probability Distributions

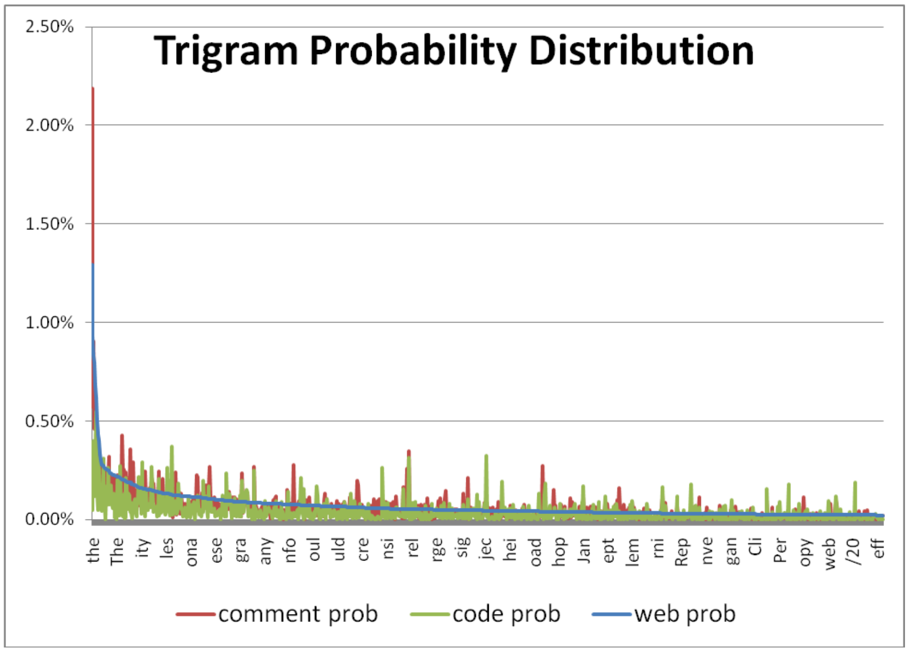
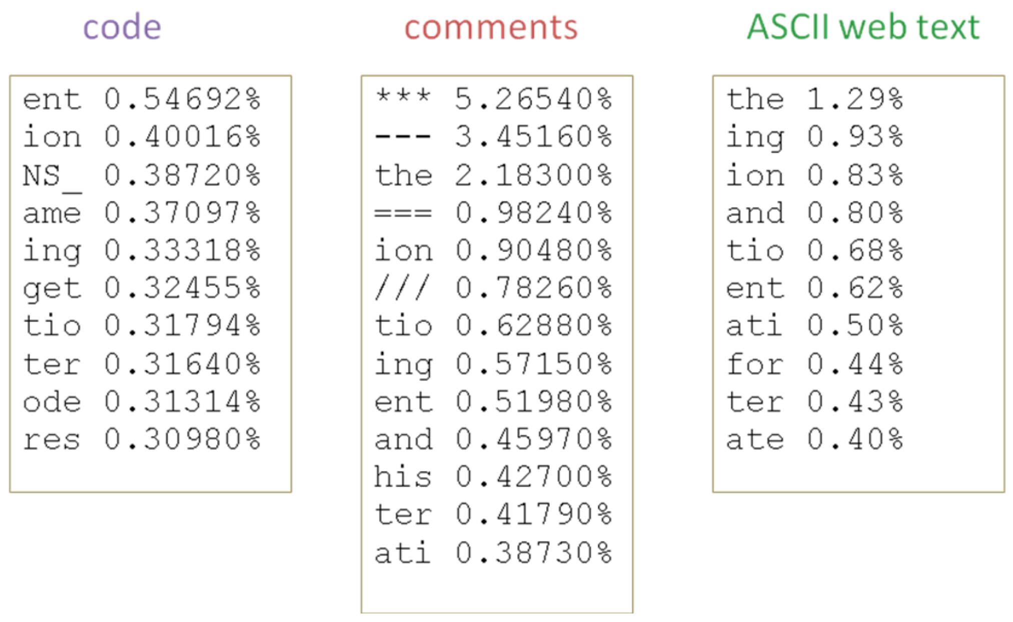


Figure 9: Most Frequent Trigrams



code

ent	0.54692%
ion	0.40016%
NS_	0.38720%
ame	0.37097%
ing	0.33318%
get	0.32455%
tio	0.31794%
ter	0.31640%
ode	0.31314%
res	0.30980%

comments

***	5.26540%
---	3.45160%
the	2.18300%
===	0.98240%
ion	0.90480%
///	0.78260%
tio	0.62880%
ing	0.57150%
ent	0.51980%
and	0.45970%
his	0.42700%
ter	0.41790%
ati	0.38730%

ASCII web text

the	1.29%
ing	0.93%
ion	0.83%
and	0.80%
tio	0.68%
ent	0.62%
ati	0.50%
for	0.44%
ter	0.43%
ate	0.40%

Figure 10: Authorship Test: Same file set

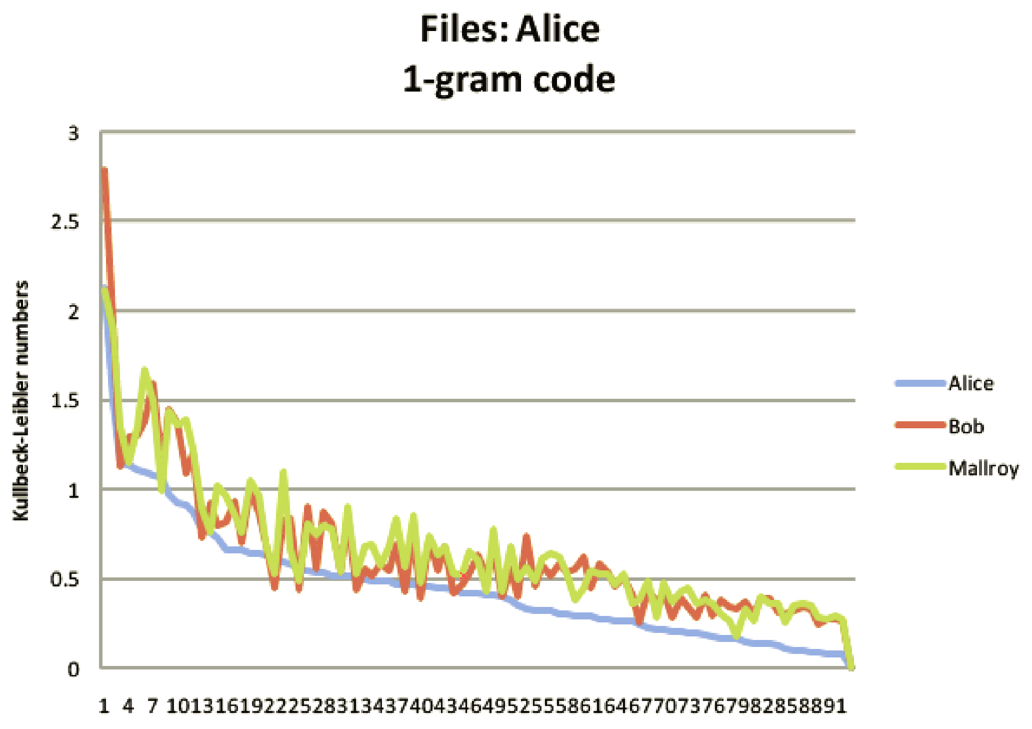
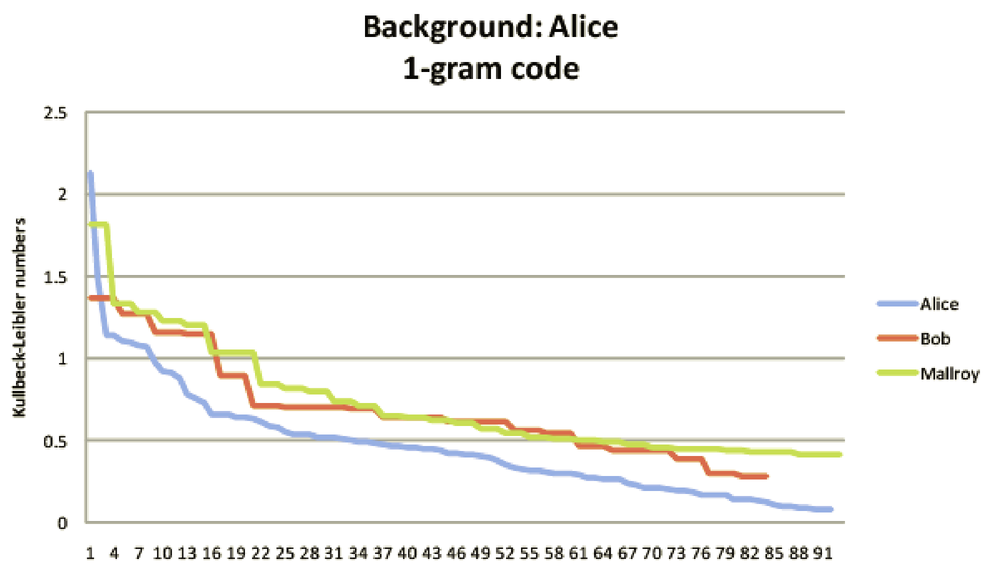


Figure 11: Authorship Test: Same background


Background: Alice
1-gram code
Kullbeck-Leibler numbers
Alice
Bob
Mallroy