WordAdjust - A Deobfuscation Frontend to Content-Aware Anti-Spam Tools

Lars Tabro Sørensen
larsts@cs.aau.dk

Martin Møller Larsen
martinml@cs.aau.dk

June, 2008

Abstract

In modern world spam has become a great threat for the entire email system. Advanced techniques are constantly being developed by spammers to bypass anti spam tools. This article aims to describe how to counter three of these techniques, namely unicode, scrambling and misspell obfuscation. We introduce WordAdjust - a frontend deobfuscation filter for the state of the art anti spam tool SpamAssassin. This deobfuscation filter is capable of deobfuscating 3 types of obfuscation methods; unicode, scrambling and intentional misspell. The Misspell deobfuscation is based on N-gram fuzzy search techniques. SpamAssassin has a hard time dealing with spam emails that are obfuscated by these 3 methods. Experiments shows how WordAdjust increases the average SpamAssassin score by 56% on intentional obfuscated spam emails. Further experiments shows how WordAdjust enables SpamAssasin to catch, in average, 10% of the spam mails that normally would end in the users inbox.

1 Introduction

Every user of electronic mail(email) has received unsolicited email with a commercial purpose. Such email is known as Unsolicited Commercial/Bulk Email (UCE/UBE) and is normally referred to as spam. The word Spam comes from the meat product Spam, and the relation to UCE was established when Monty Python performed a comedy sketch with the word spam being repeated constantly. Emails that are not classified as spam are often referred to as ham. Ever since email emerged, the popularity of email as an advertising medium has increased. Compared to many other advertising medias email has the advantage of being sent electronically. This removes the costs of manual delivery as, e.g., a postal service requires. Because the sending of email is free, people began to see the advantage of using email for advertising, thus sending emails to as many consumers as possible. Those individuals who sent these advertising emails became known as spammers.

Economically spam has a large impact on the global market. Studies show that 6% of all men, and 5% of all women [1] have ordered a product advertised through spam. Spammers claim that a rate of 0.001% [2] of positive responses are needed to make spamming economically viable, thus making this a highly profitable buissness. The downside of spam clearly lies at the recipient of the spam emails. When email addresses are being flooded by spam it does not only cause annoyances to the recipient but it can render email addresses unusable. Many email addresses receive so much spam that removing it takes a huge effort as recipient. Especially companies suffer from this fact, and this is where the economic downside of spam lies, as the effectiveness of the staff drop as the number of spam emails increases.

Interest in fighting spam automatically has in-

creased as the number of spam emails exploded. Anti spam tools have become a huge industry since all emailers have an interest in saving time by automatically removing spam. The methods used in these spam filters have evolved from rather simple techniques into more complex in order to improve their efficiency. But as the methods of the anti spammers evolve, so do the methods of the spammers too. The spammers study the methods of the spam filters and find ways to bypass them. This obviously forces anti spammers to study the methods of the spammers to find ways to counter the techniques used by the spammers. This causes a never ending battle between the anti spammers and the spammers. A common way of bypassing spam-filters is to write words in alternative ways. Spam-filters analyze the content of an email message by looking for certain words that often appear in spam. In the spam email, these words are spelled so humans perceive them as the proper words, but spam-filters cannot make any connection between the misspelled words and the right words.
In a previous project, we developed an algorithm for finding the longest substring of text T that is approximately equal to a search string P. This functionality was used in the area of song recognition where a sample piece of a song (search string P) was matched against a set of songs (strings T). The song containing the longest approximate substring was chosen as the matched song. A similar problem occurs when words are obfuscated in order to bypass content based spam filters. We have a set of strings (the words in the email) and a search string (a word with a known connection to spam). By matching each word in the email to the spam related word, we get a match to the most similar word in the email. Words that are misspelled will be matched to the original word, the spam related word, and the spamfilter will stand a higher chance of detecting emails that are spam.

In this project we develop a spam filter to deobfuscate emails that spammers have obfusacted in order to bypass anti spam programs. Our filter deobfuscates three known ways of obfuscation; unicode, scrambling and misspell. Unicode obfuscation involves replacing characters with visually similar characters, but with a different unicode in order to bypass content-aware filters. Scrambling obfuscation mixes the inner characters of spam related words. The first and last characters will keep their position, which makes humans able to perceive the word, however this is not the case for anti spam tools. Misspell obfuscation intentionally misspells spam related words in order to bypass content-aware anti spam filters.

After the filter is applied to an email we pass it to SpamAssassin, an open-source state of the art anti spam tool. The intention of this project is not to make a new anti spam tool, but rather a frontend to support existing programs with these deobfuscation tools.

The structure of the article is as follow: Section 2 starts by describing the research that have been done in the area of preventing spam. It follows by describing what commercial and open-source tools that are popular for the means of fighting spam. It ends with describing the work that have been done on fighting obfuscation. Section 3 describe the 3 ways of obfuscation we aim to solve. Section 4 describe how we implement the solutions to the methods named in Section 3. Then we test the system in Section 5 to see how well it performs is. Section 6 makes conclusions on the project and Section 7 describes possible future work.

2 Related Work

Substantial research on how to fight and remove spam has been carried out. There are several methods to approach this problem; this chapter will describe the most essential.

2.1 Spam Fighting Methods

Blacklisting is a widely used technique for preventing spam emails. Several online lists exist containing DNS addresses of servers that have been observed sending spam emails to

a substantial extent. Anti-spam tools search these lists to determine if a newly received email has been sent through a server that has been recorded as a spam providing server. If this is the case, the email is marked as spam. DNS Blacklisting was introduced in 1997 by Paul Vixie [3] where he started the organization Mail Abuse Prevention System which distributed the first DNS blacklist.

Whitelisting [4] works much like blacklisting, by observing which server an email is sent through. However, instead of matching the sending server to a list of known spam providing servers, the sending server is matched to a list of non-spam providing servers. Whitelisting seems quite aggressive, since a server must be added to the whitelist before it is possible to receive email from it. Some whitelisting filters employ a technique that makes it possible to be added to the filter automatically, by prompting the sender with a challenging question. If the sender can answer this question he will be added to the whitelist. However, whitelisting in general causes quite some work in order to maintain the list, especially for a person who frequently need to send emails to new addresses and therefore have to answer such questions.

Another common technique for fighting spam is to parse each email and scan it for content that could suggest the email is spam. A widely used approach is to create a set of rules that describe cases that often are represented in spam emails [5]. Such a case would for instance be the appearance of the word "replica" that is often used in spam emails. Each rule is associated with a value describing how likely an email containing this specific case is spam. When more cases are represented in one email, their values are added together. When a certain threshold is reached, the email is classified as spam. A rule can be based on both specific content of the email, i.e., certain words, or it can be based on metadata such as information about what character set is being used. Creating such rules can be very time consuming since new rules are required frequently since the methods of spammers evolve rapidly.

A typical technique used to assign values to rules is to feed the spam filter with a set of spam emails. The filter then observes the occurrence of each rule in all the spam emails and derives a value based on the observation. New cases are also recognized in the spam emails, and new rules are thereby created. This is normally implemented by a bayesian network.

SPA (Single-Purpose Address) is a method that aims on preventing spam, instead of automatically removing it [6]. The idea is to have one email-address for each purpose. For example, a user needs an email to confirm a login the user just created on a homepage, or the user needs an invoice for a product bought online. The user still needs a unique email-address when he needs to communicate with others, but overall this would reduce spam because spammers get their email-addresses from forums, IRC channels, or web forms that would only be a SPA. A concrete example of an implementation of this idea is the project called remailer. This project provides a tool that serves as an interface between the sender and the receiver, which hides the original email-address of the sender.

Instead of fighting spam at the end users Andreolini et al. [7] use different kinds of honeypots (traps) to track the sources of spam. Some of the techniques are open proxies and open relays.
Filtering by Duplicate Detection [8] is a method that takes advantage of the fact that spam emails are often similar. Whenever a new email is received it is compared to all previously received emails that were marked as spam. If a similarity exists, the new email is removed. If an email is marked manually as spam by the user, it is added to the list of spam emails. This method is also used in a distributed manner, so whenever a user marks an email as spam, the email will automatically be deleted by other

recipients of similar email.

2.2 Spam Fighting Tools

There are many tools that aim to aid in fighting spam emails. Most of popular the spam fighting tools utilize several of the previously mentioned methods. SpamAssassin [9] is an open-source project that employs blacklisting and the rule-based concept. SpamAssassin keeps it rule-base updated by defining new rules, and it keeps the associated values updated by learning a Bayesian Network with a set of known spam emails. Another spam filter is the commercial product SpamFighter [10] that, as SpamAssassin, is rule based. Razor [11] is a open-source project that employs the idea of distributed duplicate detection. Whenever a user of Razor detects a spam email, Razor notifies all other Razor users so if they receive a similar email, it will be deleted automatically.

2.3 Fighting Obfuscated Spam Emails

As previously mentioned, spammers keep changing and improving their methods in order to spread their emails to as many recipients as possible. Through the last couple of years, there has been a trend toward obfuscating the content of spam emails in order to bypass rule-based filters. An example could be an email containing a sentence like "Buy cheap drugs at www.cheapdrugs.com". It is quite straight-forward for a rule-based spam filter to detect this spam but the problem comes when content of the spam emails, is modified. Modifications could take form of constructed spelling mistakes like "cheep" instead of "cheap" or could involve the substitution of letters with a corresponding set of symbols. An example of this is replacing "A" with "/-\ " or "V" with "\/". One of the first articles that mentioned some techniques that might be used for detecting these constructed errors in spam emails was Ahmed and Mithun [12]. For example, they remove almost all non-alpha characters and replace consecutive repeated characters by a single character. Then they apply a phonetic algorithm to see whether the resulting string is acceptable.

The unicode charset is used on a global scale and contains most of the letters used in different languages. Unicode is used because of the needs to have one charset when communicating with people around the world, so that an "s" in one charset would not be confused with an "e" in another charset. Although this is smart, it makes spam even harder to catch because it is easier to obfuscate emails with unicode. A cyrillic v (U+FF56) can be used to replace the latin v (U+0076), and the cyrillic i (U+0456) can be used to replace the latin i (U+0069) in "Viagra" to get a technically different but identically the same for the eye. Liu and Stamm [13] consider the fighting of spam that is obfuscated using the unicode charset.

3 Methods of Obfuscation

This section explains the ideas behind the three obfuscation methods unicode, scrambling and misspelling in details.

3.1 Unicode Obfuscation

As mentioned in 2.3 unicode obfuscation is a method for spammers to obfuscate emails in order to bypass spamfilters. The unicode standard has about 100000 different characters that all have a different code. Although each character has a different code to represent it, some of them look very similar. This means that spammers can spell known blacklisted words in other ways to overcome the problem. An example is how Cockerham shows how Viagra can be spelled in 600,426,974,379,824,381,952 ways with use of unicode obfuscation [14].

3.2 Obfuscation by Scrambling

Scrambling is a technique used to distort words in a special way so that they still can be read. The technique is to keep the position of the first and last character of a word, but randomly change the position of the characters in between [15]. This means that the same

characters, and the same amount of characters, are present in the obfuscated word. A known example by many is this sentence which uses scrambling:

"Aoccdrnig to a rscheearch[1] at an Elingsh uinervtisy, it deosn't mttaer in waht oredr the ltteers in a wrod are, the olny iprmoetnt tihng is taht frist and lsat ltteer is at the rghit pclae. The rset can be a toatl mses and you can sitll raed it wouthit porbelm. Tihs is bcuseae we do not raed ervey lteter by it slef but the wrod as a wlohe. ceehiro."

This sentence was passed around as a chainmessage on emails and instant messaging program like messenger for a long period [16].

3.3 Intentional Misspelling

A popular way to bypass anti spam tools is to obfuscate through intentional misspelling. As previously mentioned, rule based anti spam tools generate rules based on emails marked as spam. The spam filter will look through the spam emails and look for common denominators, creating rules that enable the spam filter to recognize the common denominators. These rules will often describe the presence of certain words (spam words) due to the fact that the recipient will typically receive multiple spam emails with the same intention, for instance advertising some product. Spammers take advantage of the fact that the spam filters depend on the words described in the rules being spelled correct. If this is not the case the spam filters will not recognize them.

By misspelling spam words intentionally, the spam filters will not recognize them and there will be less chance the email will be marked as spam and thereby end in the inbox of the recipient. Even though the spam word is unrecognizable to the spam filter, the human mind will still perceive the meaning of it since it automatically translates it to the correct word. Thereby the spammer succeeds in his goal; bypassing spam filters and delivers his message to the recipient.

[1] Notice however that this word does not keep the definition of scrabling since it adds extra character

4 Implementation

In this section we will go through the implementation of each of our deobfuscation methods. We will show important areas of the implementation in simplified C# code in order to keep a certain level of abstraction. First we will explain how our unicode deobfuscation is implemented, followed by scramling deobfuscation and at last our misspell deobfuscation.

4.1 Overview

Figure 1 illustrates the flow of the email systems and shows where in the email architecture our system should be placed. The email is first collected by the Mail Transfer Agent from the SMTP server. Before the email is passed on to SpamAssassin, it is being examined by each of our deobfuscation techniques. SpamAssassin then declares the email being either spam or ham. If the email is declared as ham it is parsed on to the Local Delivery Agent that has the responsibility to send the email to the correct inbox. If it is declared as spam it is parsed to the Junk Mail Processor that takes care of spam email.

The implemented system is written in the C# language that is based on the .Net 3.5 Framework. The system is written in object oriented manner. The total system, along with auto generated GUI[2] code, consists of 2357 lines of code. The source-code along with precompiled binaries are attached on the DVD. Screenshot of the GUI is added in the appendix.

4.2 UnicodeFactory

Our unicodeFactory class got two main purposes; to obfuscate and deobfuscate emails regarding unicode.

[2] Graphical User Interface.

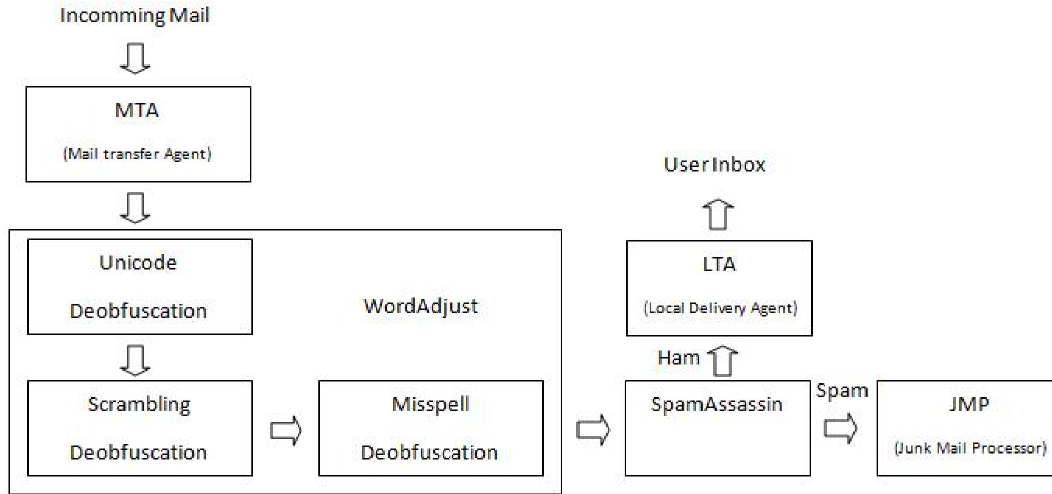

Figure 1: Illistration of the system implementation.

Unicode Deobfuscation

In order to implement our unicode deobfuscation we need a list of which characters that are alike. We use a list called UC-Simlist [17] which is a list that is generated by a formula that checks how similar each character is visually. In Figure 2, a small section of the UC-simlist is shown. Each line represent a character and which similar matches there are for that character. The first four numbers in each line is the given character in hexadecimal, and the following text is each character that matches. An example is the character 0043(C), which got a 1:1 match with the hexadecimal character 0421, which is what 1:0421:C means in the UC-simlist.

```
0041    1:FF21:A    1:0410:A    1:0391:A
0042    1:0392:B    1:FF22:B    1:0412:B
0043    1:0421:C    1:216D:C    1:FF23:C
0044    1:0044:D    1:FF24:D    0.9518072:1E0C:D
0045    1:0415:E    1:0045:E    1:0395:E
```

Figure 2: Example of the UC-Simlist

To search an email for these similar characters we first construct an index where the key is each similar character in the UC-simlist and the value is the real unicode character. Listing 1 shows how we replace each similar character and return the email afterwards.

```
1
2 foreach (char c in blacklistedWord)
3 {
4       char value;
5       if(index.TryGetValue(c, out value)
6          blacklistedWord = blacklistedWord.
              replace(c, value);
7 }
8 return blacklistedWord;
```

Listing 1: Simplified C# code for the Unicode deobfuscation.

Unicode Obfuscation

To obfuscate emails with unicode, we again use the UC-Simlist to see what characters that are alike. For each character in the email we chose a random visual similar character from the UC-Simlist and replace it with that. We only do this for blacklisted words because there is no reason to obfuscate others.

4.3 ScramblingFactory

The ScramblingFactory class contains methods to deobfuscate and obfuscate words by scrambling.

Scrambling Deobfuscation

To check an email for scrambling we search through each word in it, so with regular expressions we parse the email to get a clean list of words only. In order to know how a scrambled word should be spelled, we need a blacklist that contains the most common spam words. In Listing 2, the first foreach loop goes through each word in the blacklist and the second foreach loop go through each word in the email. The first check is if the two words are of same lengths, if they are not there is no reason to make further checks. Then we check if each character in the scrambled word in the email is contained in the blacklisted word. If all the characters are contained we replace the word in the email with the blacklisted word.

```
1
2  foreach (string blackListedWord in
       arrayBlackList)
3  {
4    foreach (Match wordInMail in wordsInMail)
5    {
6      if(wordInMail.Length == blackListedWord.
           Length)
7      {
8        if(wordInMail.ContainsAllCharsIn(
             blackListedWord))
9          theMail = theMail.Replace(wordInMail
               .ToString(), blackListedWord);
10     }
11   }
12 }
13 return theMail;
```

Listing 2: Simplified C# code for the deobfuscation of Scrambling.

Scrambling Obfuscation

To obfuscate the email using scrambling, we again use regular expression to get each word easily parsed. We then make sure the first and last character of each word stays first and last. The characters in between are then mixed up using a random generator so that it is different from the original word.

4.4 MisspellFactory

This section describes both the deobfuscation and obfuscation techniques of the misspell class MisspellFactory. Note $|string|$ is not the normal length notation, but instead a notion for a gram, e.g., $|bf|$ dedicates the gram bf and not the length of bf. As substitution we use the notion $string.Length$ to denote the length of the string, e.g., $hello.Length = 5$.

Deobfuscation

The implementation of the misspelling deobfuscation tool is based on an n-gram technique developed in an earlier project. Gram division is a technique that divides a string into a number of substrings of n characters. For instance the string "obfuscation" would correspond to the following set of substrings assuming that n=2

$|ob|;|bf|;|fu|;|us|;|sc|;|ca|;|at|;|ti|;|io|;|on|$ which means a resulting set of $string.Length - n + 1$ grams, where each subsequent gram shares a common character.
The algorithm that is used to deobfuscate intentional misspellings builds on an idea of dividing a misspelled string into a set of grams of n size, and create groupings of these grams based on how well they match string that is given as input - more on this later.

Indexing Algorithm

The process starts by taking the content of the email as input, represented as a list of strings where each string is a word from the content of the email. Each string in the list is then split into grams of size n by sending them through the **buildIndex()** method - see Listing 3

```
1  Input:
2  N - size of gram
3  M - a list of all words in the email
4
5  Output:
6  index - hashtable representing the index
7
8  Dictionary index;
9  foreach(word S in M){
10   position := 0
```

```
11     while(position < S.Length-(N-1)){
12         Add characters from position counter to
               position+N as key and wordID and
              position as value to the index.
13         position := position + 1
14         }
15 }
16 return index;
```

Listing 3: Simplified C# code for the indexing algorithm.

Each word in the email is split into grams and for each unique gram an entry in a hashtable is created with the characters representing the gram itself as key, and its position and wordID as value.

Figure 3 shows an example of the hashlist created by **buildIndex()** where M = (hey,there) and N = 2. Notice how the gram $|he|$ is present in both the string "hey" and "there", so it has multiple values though with different wordID and position. The Node object, represents a gram.

|he| -> {Node(pos = 0, wordID = hey),
{Node(pos = 1, wordID = there)}
|ey| -> {Node(pos = 1, wordID = hey)}
|th| -> {Node(pos = 0, wordID = there)}
|er| -> {Node(pos = 2, wordID = there)}
|re|-> {Node(pos = 3, wordID = there)}

Figure 3: Example of indexed email

Now that the indexing is done it is time to take a look on the actual search algorithm.

Search Algorithm

Now that the indexing is in order lets have a look on the actual search algorithm. We demonstrate the algorithm by showing an example of how functions. Figure 4 illustrates the process. For simplicity, we start by looking at only one word in the index; we will later expand to multiple words. Let us denote the indexed word w_{index} and the query word w_{query}. Specifically, assume that the we have already indexed the word "reference", with a gram size of 2. This gives us the following index: re, pos: 0; ef, pos: 1; ff, pos: 2; fe, pos: 3; er, pos: 4; re, pos: 5; en, pos: 6; nc, pos: 7, ce, pos: 8. We let w_{query} be the word "reference". The first thing that happens is that we extract the first gram from w_{query}, in this case $|re|$, and note its position in the word being position 0. We look up in the index to determine whether there are any identical grams, and we find that there are actually two grams that are equal to $|re|$. Now let us look at the first $|re|$ gram. We compare the position difference, p_{diff} between the gram from w_{query} and the gram we are looking at from the index. Both grams have position 0, so $p_{diff} = indexgram.pos - querygram.pos = 0$.

Since this is the first occurrence where the position difference p_{diff} is 0, we create a "position group" with the name "0". We also assign one point to this position group. We now look at the second $|re|$ gram from the index and find that the p_{diff} is 5. This is the first occurrence of $p_{diff} = 5$, so we create a new position group with the name "5" and assign one point to it.

Moving on to the next gram from w_{query}, we find the gram $|ef|$ in position 1 in w_{query}. We look for equal grams in the index and find one occurrence whose position is also 1; therefore, $p_{diff} = 0$, and yet another point is assigned to position group "0" that now has two points. The next gram in w_{query} is $|fe|$. One occurrence of $|fe|$ is present in the index with position 3; because the position of $|fe|$ in w_{query} is 2, we get a p_{diff} of 1. Now one might think it is time to create a new position group since this is the first occurrence of $p_{diff} = 1$, but we introduce instead the feature "threshold" that describes by how many positions a gram is allowed to differ between the indexed word and the query word. Let us say that the threshold in this example is 1. We saw that the gram $|fe|$ had a p_{diff} of 1, so we check whether a position group exists within a range of 1 from 1 (either 0 or 2), and we notice that a position group 0 is active. So instead, we add another point to position group 0.

This process is continued until the last gram

in w_{query} has been examined. The created position groups are then returned, and the number of points of the leading position group is used to evaluate whether the query word should be replaced with the indexed word or not. In this case, the leading position group is 0 and has 8 points. This means that the index word has 8 grams that follow the order from the query word, with a character position difference allowance of 1. The total number of grams of the indexed word is $reference.Length - n + 1 = 10 - 2 + 1 = 9$. Because 8 grams are close to 9 grams, we substitute the query word with the indexed word. If the number of grams in the indexed word would have been larger, a substitution would not have happened, since this would indicate that only a substring of the query word matches the index word.
The full example is shown in Figure 4.

Gram	Position in q	Position in index	Group
re	0	0 5	0 5
ef	1	1	0
fe	2	3	0 (1 but t = 1)
er	3	4	0 (1 but t = 1)
re	4	0 5	-4 0 (1 but t = 1)
en	5	6	0 (1 but t = 1)
nc	6	7	0 (1 but t = 1)
ce	7	8	0 (1 but t = 1)

Group	Points (in each group)	Total points
0	1+1+0+1+1+1+1+1	8
5	1+0+0+0+0+0+0+0	1
-4	0+0+0+0+0+1+0+0	1

Figure 4: Example of the misspell searching algorithm. Gram-size = 2, t (threshold) = 1, indexed word = "refference", query word = "reference"

As mentioned this example does only cover the occurrence of one word in the index. Our system obviously needs to handle multiple words since an email consists of multiple words. We handle multiple words in the index by assigning a wordID, that points to the specific word in the email, to each gram in the index. When we create position groups we make sure that they are each associated with a word from the email.

In the implementation we handle this situation by maintaining multiple hashtables. We maintain a hashtable that takes a position difference as key (this is the name of the position group) and it returns a new hashtable that takes a wordID as key. The returning value is the position group of the specific wordID. The search algorithm in simplified C# code can be seen in listing 4.

The algorithm starts by extracting the first gram in the query word (lines 11-12). It then extracts entries in the index that matches the query gram (line 13). For each of the matching grams, the position difference is calculated (line 14). We check whether a position group is active for the specific position difference on the word of the actual gram (lines 15-16). If this is the case, we add another point to the position group (line 19). If it is not the case we jump to line 22, where we create a new position group and add it to all the position differences that are within threshold range of the position group(line 23-28). The process continues until all grams from the query word have been examined.

```
1  Input:
2  Q - query word.
3  N - size of gram.
4  INDEX - The song database indexed with N.
5  T - threshold
6  Output:
7  Created Position Groups.
8
9  querypos:= 0
10 songlist := empty
11 while(querypos < querysong.length - N-1){
12    querysubstring := characters from position
          querypos to querypos+N
13    foreach(gram g in INDEX that is equal to
         querysubstring){
14       positiongroup := g.counter - querypos
15       tmpgrouplist := all groups from
             grouplist with positiongroup and
```

```
          same wordid as g
16      if(tmpgrouplist is not empty){
17         foreach word w in tmpgrouplist
18            if querypos != w.querypos
19               give w.word one more point
20         w.querypos = querypos
21      }
22      else{
23         tcounter := -1 * T
24         groupobject group
25         group.points = 1
26         while(tcounter <= T){
27            add group(wordid,positiongroup+
                  tcounter and querypos) to
                  grouplist
28            T := T + 1
29         }
30      }
31   }
32   querypos := querypos + 1
33 }
34 return position groups
```

Listing 4: Simplified C# code for the searching algorithm.

Obfuscation

The obfuscation method picks a random position in the input word that is supposed to be obfuscated. It inserts the special character "*" at this position. The rest of the string is untouched. An example could be the word "replica" being obfuscated to the word "re*plica". Note that the method ensures that the random position cannot be position 0 or the last position of the word, since this would leave the original word unchanged, e.g *replica or replica*.

5 Performance Studies

5.1 Test settings

In order to measure how well the system performs a proper test platform is needed. We take use of the anti-spam filter called "SpamAssassin". SpamAssassin determines a score of how likely a mail is spam. This is done by examining all content of the mail and matching it to a set of rules. Each rule in the ruleset adds a certain value to the score based on how large an impact the specific rule is considered. SpamAssassin is rather slow though, so to optimize its performance in order to do test on bulks of data we take use of the tool called SpamD. SpamD is an optimized interface to SpamAssassin that makes bulk operations much faster, however the score of the individual emails will not change. SpamAssassin is a Bayesian-based spam filter that enables it to create new rules by feeding it with emails known to be classified as spam. SpamAssassin employs a technique of using distributed lists of webpages that are known to have a connection to spam. If any of these webpages are shown the mail, SpamAssassin will add a high value to the final score. We disable this feature in order to focus on the methods our system aims to solve. In order to determine how the system performs tests are performed on each module of the system; the unicode module, the scrambling module and the misspelling module. The tests are performed by giving SpamAssassin a bulk of obfuscated emails. We obfuscate the emails with each of previously described obfuscation types, and measures the score from SpamAssassin. Then the deobfuscation techniques are applied to the bulk of emails and they are used once again as input to SpamAssassin. Now the scores of each mail, obfuscated and deobfuscated, are compared in order to evaluate how big an influence the modules have. In order to evaluate only on what is relevant to us we have disabled all online checks e.g. ban-lists of relay IPs or blacklisted URLs listed in the email.

In order to make these tests we got 7500 spam emails, all from the same inbox [18]. We know that all of these emails are spam because of the way they have been collected. They have been gathered by posting the specific email address across forums and other public places on the Internet so that the spammer's crawlers[3] can get the email address.

[3] An application that scan homepages, in this case for email addresses

5.2 Specific Example

Before doing tests on hundreds of emails we want to see what output SpamAssassin gives us on a specific spam email. First we give SpamAssassin a unicode obfuscated spam email and see what rules it trigger. Then we deobfuscate the spam email and see what rules it then triggers. The obfuscated email should of course have a lower score than the deobfuscated email. The actual emails can be seen appendix A.

Rules triggered by obfuscated email:

- 2.6 INVALID_MSGID Message-Id is not valid, according to RFC 2822

Rules triggered by deobfuscated email:

- 1.2 FS_REPLICA Subject says "replica"
- 3.8 FS_REPLICAWATCH Subject says Replica watch
- 3.4 REPLICA_WATCH BODY: Message talks about a replica watch
- 2.6 INVALID_MSGID Message-Id is not valid, according to RFC 2822

As shown on the two examples, when we deobfuscate an email SpamAssassin gives it a much higher score. The obfuscated email has a score of 2.6 whereas the deobfuscated email has a score of 11. This means that this specific email would go from spam to spam if the standard threshold of SpamAssassin is used which is 5 point. In this example the spam email is about replica watches which of course gives a high score. But when it is obfuscated, SpamAssassin is not able to find these words which results in this spam becoming ham.
Specific examples for each obfuscation/deobfuscation method can be found in the Appendix.

5.3 Unicode Test

This test measures how well the unicode implementation performs. For this test we selected 150 spam emails from the set of 7500 spam emails which at least contain one word that we know would trigger a rule in SpamAssassin. This could be words as viagra, replicate or penis. We search through each email, replacing words from the blacklist, with obfuscated representations of them. A unicode obfuscated email is shown in Figure 11. When we do this SpamAssassin should very likely calculate a lower score to the spam mail which means less chances of being marked as ham. We pass each obfuscated email to SpamAssassin and calculate their score. Afterward each email is deobfuscated with our system. A unicode deobfuscated email is shown in Figure 12. On Figure 5 the red area represent scores of the deobfuscated email and the blue area represent the score increase the unicode deobfuscation system affect. The x-axis represent each email and the y-axis represent the score SpamAssassin assign to the email. At average the unicode deobfuscation module increase the score assigned to each spam mail by 2.9, which is an increase of 56.3% compared to the average score from the obfuscated emails.

5.4 Scrambling Test

The scrambling test is performed on exactly the same set of spam emails as the unicode test was. The difference is we use our scrambling obfuscation technique to scramble the words appearing both in the blacklist and the email. Theoretically we should get the same result in this test as we did in the unicode test since the same content of the emails is obfuscated. At average with our scrambling deobfuscation program we can increase the score assigned to each spam email with 2.9 points, which is an increase of 56.3% compared to the average score from the obfuscated emails. Notice how this result is the exact same as the unicode filter. This indicates that both filters obfuscates and deobfuscates the same words equally effective.

5.5 Misspell Test

Misspell obfuscation is done by adding an extra character to each word defined in the blacklist. For instance the word "replica" will be obfuscated as "repl*ica". Easy to perceive to human beings, however unrecognizable

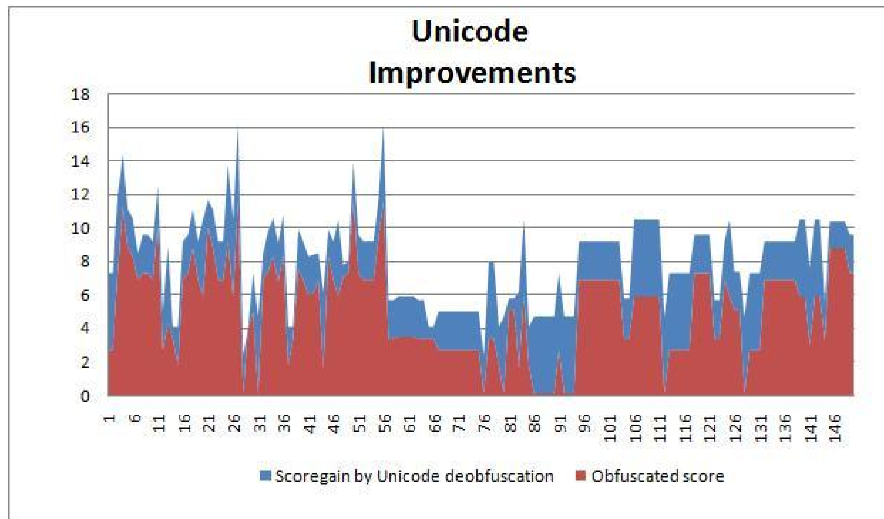

Figure 5: Illustration of score difference between obfuscated and deobfuscated spam emails.

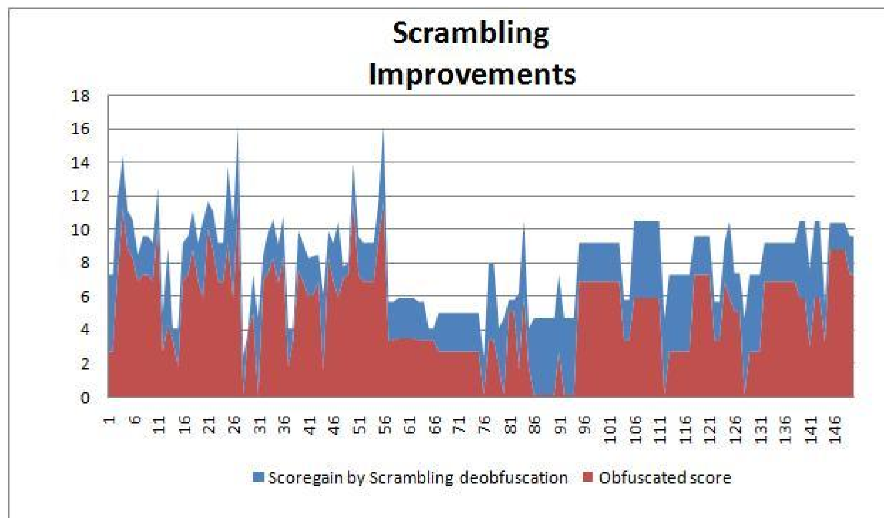

Figure 6: Illiustration of score difference between scrambling deobfuscated and obfuscated spam emails

to SpamAssassin. The same dataset is used as in previous tests. Increase in score from obfuscation to deobfuscation proof that the misspelling algorithm corrects the obfuscated word into the original intented word. This enables SpamAssassin to recognize it, and adds a proper rule from its rule base. In average the score is increased by 2.7 which corresponds to an increase of 51.8% percent compared to the average obfuscated score. The misspelling module does only obfuscate the words that is present in the blacklist, which means some words might trigger a rule in SpamAssassin but it might not be present in the blacklist.
In the previous project where we proposed the algorithm described in 4 we aimed to achieve high performance since this was the major purpose of the algorithm. We showed that

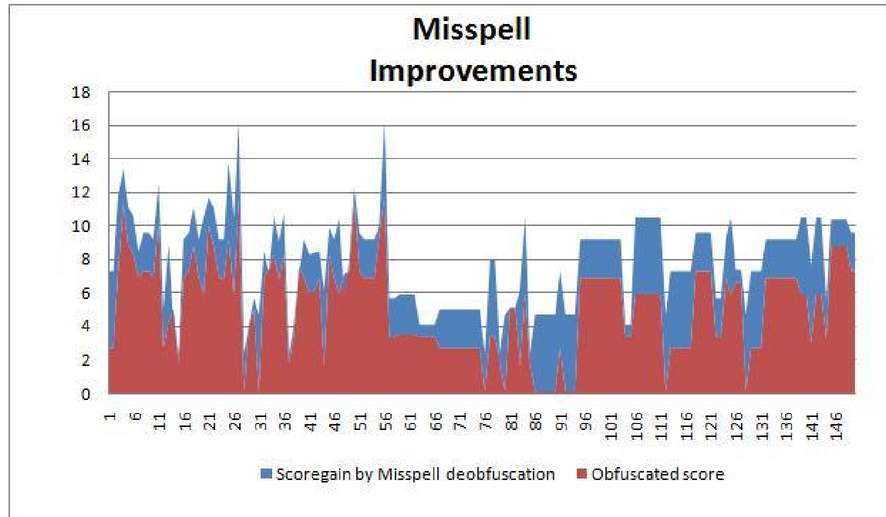

Figure 7: Illiustration of score difference between misspell deobfuscated and obfuscated spam emails.

changing the parameters in the algorithm had great influence on performance in a manner of both correctness and processing time. Increasing the gram size n causes faster, but also less accurate, searches. A gram-size of three was suggested to achieve fast, yet still quite accurate, results. In this project, we use two as gram size, since the indexes are generated from each word consisting in an email. This gives us indexes with much less entries, so changing the gram-size to two gives us more accurate results in a decent time.

5.6 WordAdjust vs Unicode project

The previous three module tests shows that all the modules have an impact on the score evaluated by SpamAssassin. These tests, however,are individually. The unicode filter is as previously mentioned implemented as done by [13]. It would be natural to measure how big a difference the three modules combined causes in contrast to the unicode filter alone. This reflects the scenario where a spammer takes use of both unicode and scrambling obfuscation. The test is performed by collecting SpamAssassin scores based on a set of emails that have been both unicode and scrambling obfuscated. The emails are then unicode deobfuscated and SpamAssassin scores are collected. This reflects how well the project proposed in [13] performs when spammers takes use of scrambling as additional obfuscation. Then scrambling deobfuscation is applied in order to evaluate how well our system performs in contrast. Unicode deobfuscation should not theoretically have any effect at all on a email that have been both unicode and scrambling obfuscated. The first bar shows the average SpamAssassin score given to the emails added both unicode and scrambling obfuscation. Second bar shows the average score given after parsing the obfuscated emails through the unicode deobfuscation module. Notice how the average score is exactly equal to the obfuscated average. This shows that the unicode deobfuscation separately has no impact on the result. The third bar shows how the average score increases greatly when scrambling deobfuscation is applied along with unicode deobfuscation.

The results are unsurprisingly almost identical when we combine the unicode and misspelling techniques so we omit these graphs.

The reason we do not obfuscate with all three obfuscation techniques subsequentially, is due

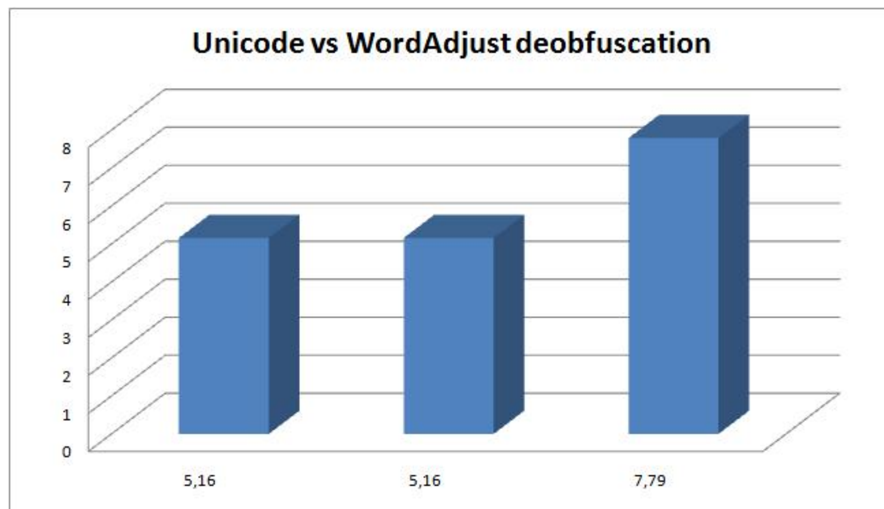

Figure 8: Illustration of the WordAdjust test. The first bar shows the average score of the obfuscated email. The second bar shows the average score of the unicode deobfuscated email. The third bar shows the average score of the unicode and scrambling deobfuscated email.

to the fact that the obfuscated words gets more or less unreadable. Our system will not be able to deobfuscate them either since adding extra characters by misspelling will disable the functionality of the scrambling deobfuscation. And if the words are still scrambled the misspelling deobfuscation will not have high chances of deobfuscating it, since it somewhat depends on correct order of the characters in the word.

5.7 Deobfuscation without obfuscation

The previous four tests measures the effectiveness of our system by assuring that the emails it is tested on were obfuscated. To measure how big an improvement our system causes to the number of spam emails that passes through SpamAssassin, even though being spam, we extract 100 emails that SpamAssassin evaluates a score lower than five. These emails will pass through SpamAssassin and end in the inbox of the recipient, since SpamAssassin by default marks emails as spam with a score higher than five. The score of these emails can be seen in Figure 9.
By applying unicode, scrambling and misspell deobfuscation to these emails an increase in score should be happening to emails that contains obfuscated words. The number of emails that exceed a score of five is the number of emails that our system prevents from ending at the receiver. As Figure 10 shows a total number of ten emails increased its score to above five. This corresponds to a decrease in spam emails that slips through SpamAssassin by 10%. We did this test on three independent sets of spam emails, each test consisting of 100 spam emails. As just shown, the first test decreaced spam emails by 10%, second test by 8% and third test by 12%. We choose only to show test results from our first test because the others were very similar in results.

5.8 Performance Conclusion

After testing all our deobfuscation methods we conclude that for certain set of emails we can greatly improve the results of SpamAssassin. Even though SpamAssassin is a bayesian system which do a lot heuristic tests it still lacks the ability to handle word obfuscation as unicode, scrambling or by intentionally misspelling. As our unicode, scrambling and misspelling tests shows we can increase at average the score on obfuscated spam email by about

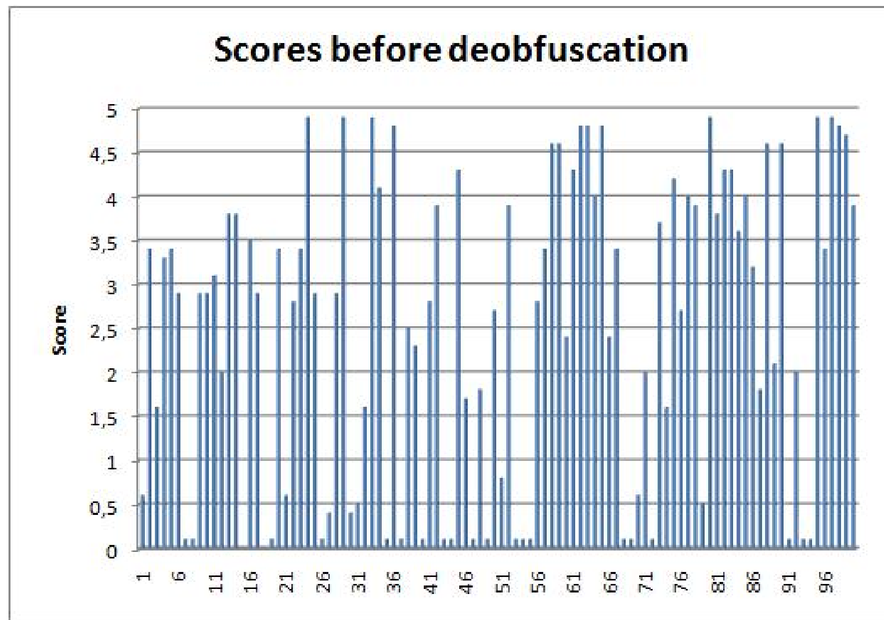

Figure 9: Illustration of 100 randomly picked spam emails with a score below 5, which means SpamAssassin wrongly marks these emails as ham.

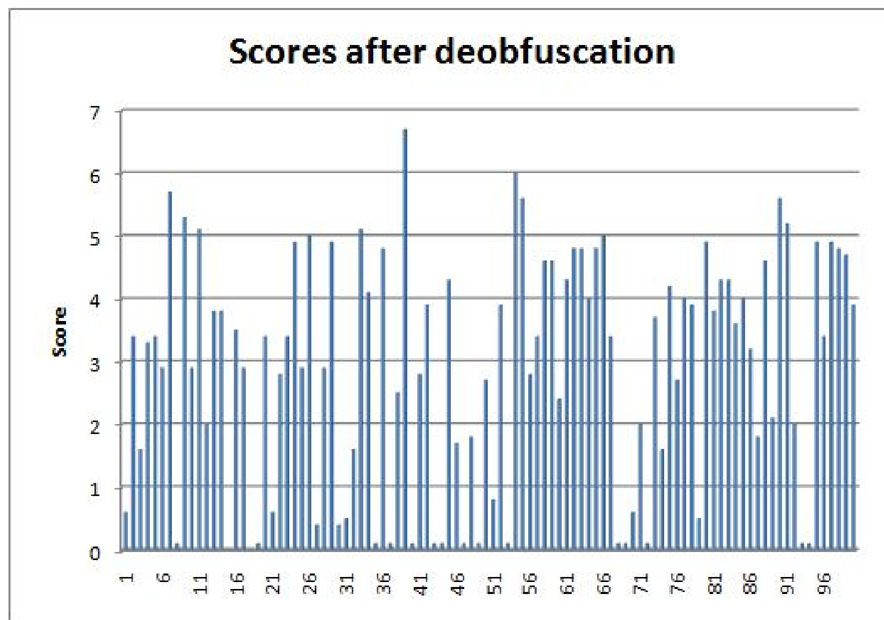

Figure 10: Illustration of the 100 randomly picked spam emails after WordAdjust have been applied. The score of 10% of the emails have risen to above 5 and will therefore be declared as spam by SpamAssassin.

56%. Unfortunately we additionally have to conclude that our system is not capable of deobfuscating words that have been obfuscated by both scrambling and misspelling. The deobfuscation without obfuscation test shows that we can remove approximately 10% of the spam emails that slips through SpamAssassin.

6 Conclusion

In this project we aimed to create a system with a purpose to increase efficiency on existing anti-spam tools. The problem consisted in spammers taking the advantage of anti-spam tools lack of ability to recognize distorted emails. We aimed to solve these three obfuscation techniques within the area of obfuscation: unicode, scrambling and misspell obfuscation.
We solved the unicode obfuscation problem by implementing an idea already proposed by [13]. We shown by test that unicode obfuscation is capable of dramatically decreasing the efficiency of the anti-spam tool SpamAssassin. We showed in Section 5.3 that parsing the unicode obfuscated email through unicode deobfuscation algorithm on average increased the score given by SpamAssassin by 56%. However, if the email is additionally obfuscated by the technique of either scrambling or misspelling obfuscation, the unicode deobfuscation obviously has no effect. This claim is supported by performance test 5.6. We additionally showed that our system in average is able to catch 10% of the spam emails that normally parses through SpamAssassin.

But if we apply our scrambling/misspelling deobfuscation to an email obfuscated by these techniques our performance studies show that close to all obfuscated words get translated to the correct word, thereby increasing the resulting score from SpamAssassin. This support our claim that if a email is obfuscated by unicode combined with scrambling/misspell obfuscation, our system corrects the obfuscated words into the correct spelling. This enables SpamAssassin to assign a higher, and correct, score to the email thereby increasing the chance of correctly marking the email as spam. This enables us to conclude that using our system will generally decrease the number of spam mails that slips through SpamAssassin.

7 Future Work

Our system deobfuscate unicode/scrambling/misspell, but a common problem which we discovered while benchmarking our system were another way spammers distort emails. The technique is to omit spaces between words and capitalize the beginning character of each word. An example of this could be:"buyCheapWatchesHere". As the example shows it is easy for humans to read this text, but spam filters as SpamAssassin do not read this as spam. It would be interesting to find a way to eliminate the problem with scrambled and misspelled words together aswell.

WordAdjust is a generic frontend so seen from realization point of view, it is a potential addition to more spamfilters than just SpamAssassin. It is easy to implement, since it is a frontend, and need therefore no interaction with the the spamfilter.

References

[1] D. Fallows. How women and men use the internet. *Pew Internet and American Life Project*, 2005. `http://www.pewinternet.org/pdfs/PIP_Women_and_Men_online.pdf` Checked: 9'th of June, 2008

[2] B. Laurie and R. Clayton. Proof-of-work proves not to work. In *The Third Annual Workshop on Economics and Information Security, May 2004.*, 2004. `http://www.dtc.umn.edu/weis2004/clayton.pdf` Checked: 9'th of June, 2008

[3] ACMqueue.com. Dns - although it contains just a few simple rules, dns has grown into an enormously complex system. *Network and Distributed System Security Symposium*, page 29, 2007.

http://mags.acm.org/queue/20080102/?u1=texterity Checked: 9'th of June, 2008

[4] Whitelist - wikipedia http://en.wikipedia.org/wiki/Whitelist. Checked: 9'th of June, 2008

[5] A. Khorsi. An overview of content-based spam filtering techniques. *Informatica 31*, pages 269–277, May 2007. http://www.informatica.si/PDF/31-3/12_Khorsi-AnOverviewofContent-BasedSpam...pdf Checked: 9'th of June, 2008

[6] J. Ioannidis. Fighting spam by encapsulating policy in email addresses. *Network and Distributed System Security Symposium*, pages 17–24, 2003. "http://www.isoc.org/isoc/conferences/ndss/03/proceedings/papers/1.pdf" Checked: 9'th of June, 2008

[7] Mauro Andreolini, Alessandro Bulgarelli, Michele Colajanni, and Francesca Mazzoni. Honeyspam: honeypots fighting spam at the source. In *SRUTI'05: Proceedings of the Steps to Reducing Unwanted Traffic on the Internet on Steps to Reducing Unwanted Traffic on the Internet Workshop*, pages 11–11, Berkeley, CA, USA, 2005. USENIX Association. "http://www.usenix.org/events/sruti05/tech/andreolini.html" Checked: 9'th of June, 2008

[8] N. Jindal, B. Liu. Review spam detection. *WWW 2007*, pages 1189–1190, May 2007. "http://www2007.org/posters/poster930.pdf" Checked: 9'th of June, 2008

[9] The apache spamassassin project - the powerful number 1 open-source spam filter. The official homepage of Anti-Spam tool SpamAssassin. http://spamassassin.apache.org/. Checked: 9'th of June, 2008

[10] Spam filter for outlook and express, windows mail and servers. The official homepage of Anti-Spam tool Spamfighter. http://www.spamfighter.com/ Checked: 9'th of June, 2008

[11] Razor - spam should not be propagated beyond necessity. The official homepage of Anti-Spam tool Razor. http://razor.sourceforge.net/. Checked: 9'th of June, 2008

[12] S. Ahmed, F. Mithun. Word stemming to enhance spam filtering. In *the Conference on Email and Anti-Spam*, 2004. http://www.ceas.cc/papers-2004/167.pdf Checked: 9'th of June, 2008

[13] Changwei Liu and Sid Stamm. Fighting unicode-obfuscated spam. In *eCrime '07: Proceedings of the anti-phishing working groups 2nd annual eCrime researchers summit*, pages 45–59, New York, NY, USA, 2007. ACM. http://apwg.org/ecrimeresearch/2007/proceedings/p45_liu.pdf Checked: 9'th of June, 2008

[14] R. Cockerham. Different ways to spell viagra. http://cockeyed.com/lessons/viagra/viagra.html. Checked: 9'th of June, 2008

[15] R. Shillcock, P. Monaghan. An anatomical perspective on sublexical units: The influence of the split fovea. 2003. "http://www-users.york.ac.uk/~pjm21/papers/LCP.pdf" Checked: 9'th of June, 2008

[16] Can you raed tihs? http://www.snopes.com/language/apocryph/cambridge.asp. Checked: 9'th of June, 2008

[17] A. Y. Fu, X. Deng, W. Liu, G. Little. The Methodology and an Application to Fight Against Unicode Attacks. *Proceedings of the Second Symposium*, 2006. http://cups.cs.cmu.edu/soups/2006/proceedings/p91_fu.pdf Checked: 9'th of June, 2008

[18] B. Guenter. Spam archive. We use the spam archive from march 2008. http://untroubled.org/spam/ Checked: 9'th of June, 2008

Figure 11: Unicode Obfuscated mail

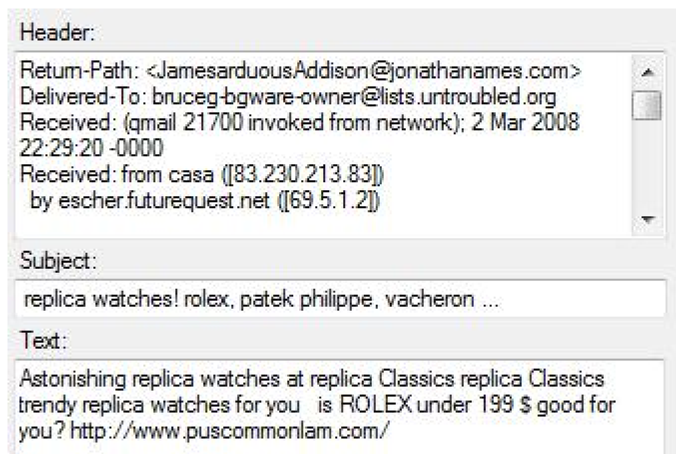

Figure 12: Unicode Deobfuscated mail

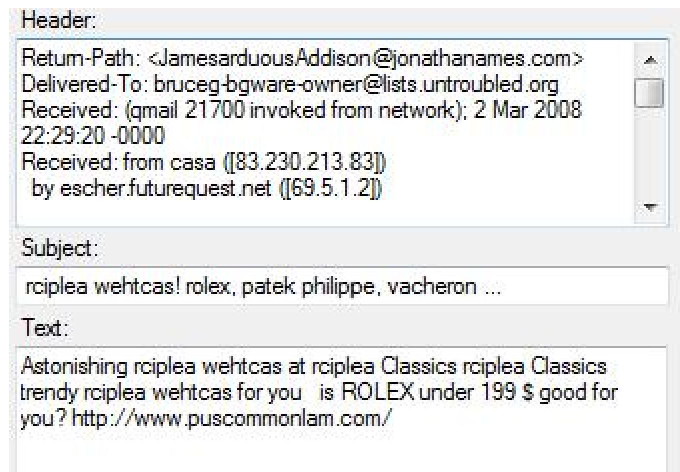

Figure 13: Scrambling Obfuscated mail

Figure 14: Scrambling Deobfuscated mail

Figure 15: Misspell Obfuscated mail

Figure 16: Misspell Deobfuscated mail

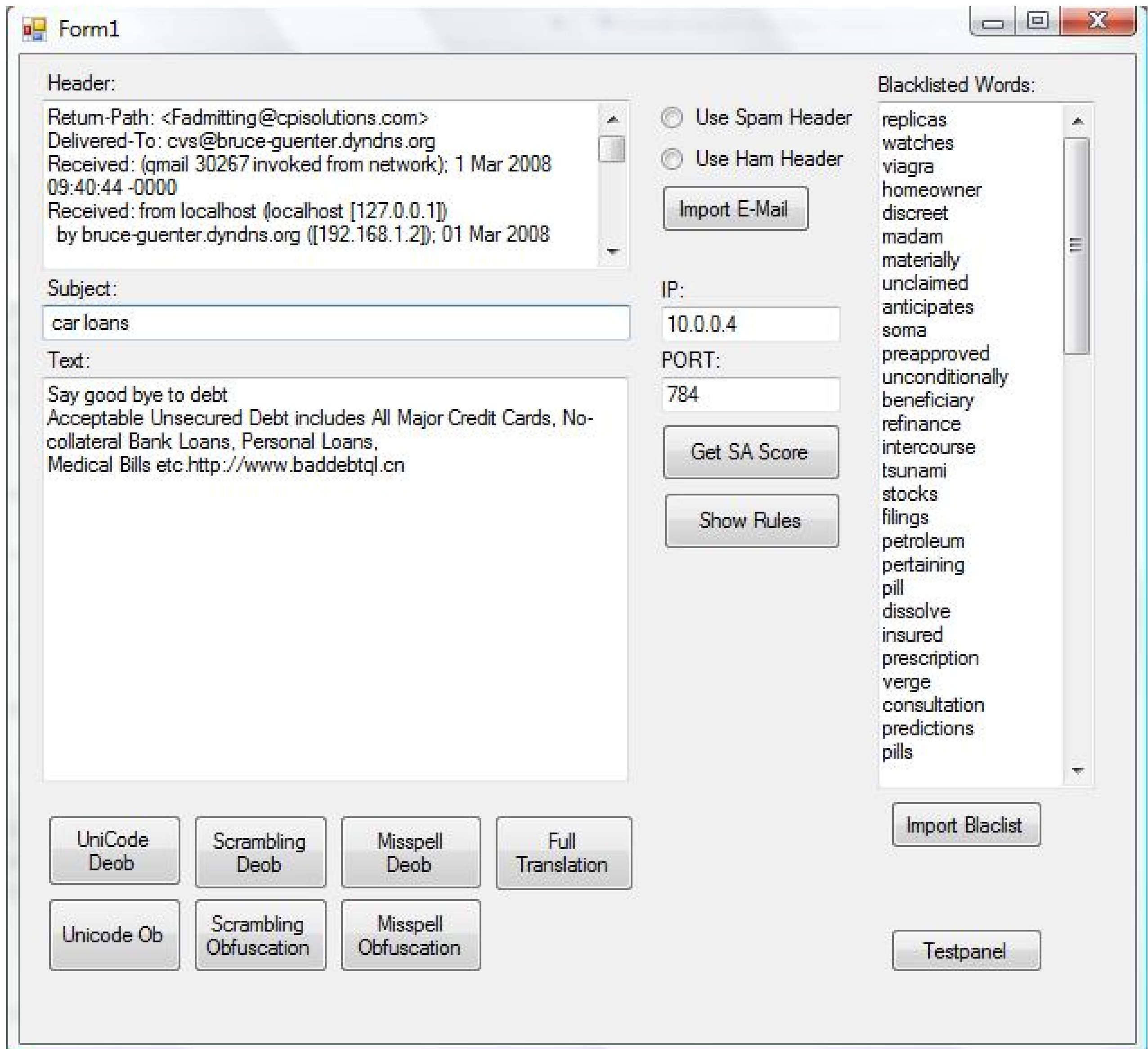

Figure 17: Screenshot of the GUI of WordAdjust