Encrypted Search

Jimmy Swanbeck
Endicott College

Thesis I

December 12, 2014

Abstract

One potentially useful feature missing from most email services today is the ability to encrypt your emails. The main reason no email provider seems to offer this service is because it presents an issue with organizing and searching through the database of your email messages. This issue arises when you try to perform a search on the encrypted information: an encryption algorithm performs a private set of operations on a character encoding of the plaintext to generate the ciphertext, so that whatever you want to encrypt is done so in a case-sensitive manner. Once the text is encrypted and stored on the email server there is no way to change that text without having the client download it first, as the client is the only one with the ability to decrypt it. This makes normalization impossible, and therefore the typical cloud-based search algorithm that involves normalizing both the search term and the email text in order to remove case-sensitivity will not work.

Our proposed solution to this problem will attempt to remove the need to search through the body of a message by creating an index of unique words that is attached to each email. This index will be created and normalized before it is encrypted. The same normalization and encryption rules will then be applied to any search text before being executed, thus making it possible to effectively search through an encrypted email. This search algorithm will also take into account other important aspects of any modern-day search engine such as ranked, fuzzy, and multi-keyword searches, and will do so while staying mindful of security concerns.

Table of Contents

Introduction …………………………………………………………… 1

Related Work …………………………………………………………. 3

Conclusion ……………………………………………………………….. 9

Proposed Solution ……………………………..……………………… 10

Algorithms ………………………………………………………………….. 12

Architecture ……..……………………………………………………….. 16

Timeline ……...……………………….……………………………….. 16

Bibliography ………………………………………………………... 18

Introduction

One of the many important features of any web service today is cloud storage. Your email provider likely stores all of your emails on its own servers, allowing you to access them from any location. Unfortunately, cloud storage can complicate certain important security features. There are not really any email providers that offer a service to encrypt your emails, and an important reason why is because once you have encrypted a message it becomes much more difficult to search through its contents.

When you encrypt a message it is done so in a case-sensitive manner. This happens because the uppercase and lowercase versions of every letter each have a unique code point. This is a number that is used to identify the character in a character set (such as ASCII or Unicode). For instance, the ASCII code point for “a” is 97, but the code point for “A” is 65. This number is the starting point for any encryption algorithm, so that the ciphertext will always depend on the plaintext’s code points. Therefore, any change in capitalization will result in an entirely different encrypted message. If you were to try to perform a search on a database of encrypted emails, you would likely just encrypt the search term and scan through the emails to find a match. The problem here is that this will only return exact matches, as the results are all case-sensitive. However, this would also be the case if you were searching a database of non-encrypted emails; the only reason that it is possible to search through plaintext while ignoring case-sensitivity is because it is possible for us to normalize it to match the search term. Because there is nothing in common with the ciphertext generated by encrypting the capital and lowercase versions of the same letter, it is not possible to normalize ciphertext. Therefore, the standard method of normalizing the body of the email on the fly will not work in this situation.

To simplify it, the way a normal non-encrypted email provider allows you to search through your emails is by capitalizing every character in both the email and the search term so that random capitalization will not affect your search results, then matching them against each other. This is not possible with an encrypted email because once the text has been encrypted you cannot change capitalization (or anything else) without decrypting and re-encrypting it. This being a cloud service, decryption should only happen on the client side, as only the client has access to the private key. By that logic you would have to download every email in the database, decrypt, and then perform a search on it to do it the standard way. However, downloading emails onto your computer defeats the purpose of a cloud service.

Related Work

Much research has already been done on encrypted search. Many existing applications already offer to encrypt your stored data, and it is not rare to find a service with a foundation similar to this project. That is to say, the concept of an encrypted index is not unique. However, it is uncommon enough that we feel there is a lot left to be desired of an application utilizing this method of encrypted search. While fundamentally our goal is to make encrypted search (without client-side decryption) compatible with a cloud server, we would also like to incorporate other modern search features as outlined in the analysis below (*Figure 1*).

	Secure Search	Case Insensitive Search	Ranked Keyword Search	Fuzzy Keyword Search	Conjunctive Keyword Search	Modern Security Context
Li, Jin, et al. [1]	🟩	🟩		🟩		
Wang, Cong, et al. [2]	🟩	🟩	🟩			
Cao, Ning, et al. [3]	🟩	🟩	🟩		🟩	
Golle, Philippe, et al. [4]	🟩	🟩	🟩		🟩	
Li, Ming, et al. [5]	🟩	🟩				🟩
Boneh, Dan, et al. [6]	🟩					
Lu, Wenjun, et al. [7]	🟩	🟩				
Song, Dawn, et al.. [8]	🟩	🟩				
Swaminathan, Ashwin, et al. [9]	🟩	🟩	🟩			🟩
Goh, Eu-Jin. [10]	🟩					

Figure 1: Bibliography analysis

Goh lays the framework for what our project would like to accomplish by discussing normal methods for searching keyword indexes such as hash tables and how those are unsuitable for encrypted indexes [10]. It introduces the concept of a “trapdoor” which exists for a particular word and is only able to test the index for the existence of that word. Trapdoors can only be generated with a private key. The authors of this article consider how it would be inefficient to download all of the files you wish to search onto your own machine; instead you must do the searching on the server by generating a trapdoor which can only test for the existence of a particular word in the database (boolean search).

This form of encrypted search is only necessary in a multi-user setting where the database is constantly changing. If it exists as a static database it would make much more sense to have a hash table with pointers to the information. Since this index would exist locally, there is no need for security.

Boneh expounds upon this topic by explaining how it can work in conjunction with a public/private key system [6]. In this context, a public key makes it possible for an email server to identify certain keywords without learning anything about the rest of the email contents (this is the trapdoor talked about in [10]). This is similar to our topic in that it allows for searching through the encrypted contents of an email message, but different in that the cloud server is doing the searching instead of the user. This may be an interesting feature to consider for additional functionality (e.g. forwarding or categorizing emails automatically based on a specific keyword) but is not how we plan to handle encrypted searching.

Similar to [10] and [6], Song et al. explore using a boolean method to determine the presence of a particular keyword in a given document. This is essential to our topic because it involves matching an encrypted search term against an indexed search term, returning true or false. It also assumes that the server is attempting to learn information about the plaintext document, and can offer provably secure methods of hiding information from the honest-but-curious server.

We can build upon this concept of an encrypted index by incorporating other search features to improve functionality for the user. Jin Li et al. describe an additional feature we plan to include in our final product; adding support for “fuzzy” search terms [1]. This means including common misspellings, wildcard searches, and other small factors that make search algorithms more intuitive in utilizing the index.

Consider what it would take to add an exception for every letter in every word in the body of the message. For each letter in a given word you would need to add 25 additional words to the index where you replace that letter with each letter in the alphabet. The result of this can be seen in *Figure 2*.

Obviously if you do this for every letter of every word in a message you are going to greatly increase the size of the email. For every word of length n, you will add $n(25n + 1)$ letters to the index. For a word of length 6 that would add 906 characters, which is a substantial amount when you consider how many words are in an entire email. Jin Li et al. also bring up that this is not the only form of fuzzy search that you can account for: aside from substitution, there are also insertion and deletion [1].

CASTLE:

{AASTLE, BASTLE, DASTLE, …, YASTLE, ZASTLE},
{CBSTLE, CCSTLE, CDSTLE, …, CYSTLE, ZYSTLE},
…
{CASTLA, CASTLB, CASTLC, …, CASTLY, CASTLZ}

Figure 2: Direct implementation (brute force) of fuzzy search for single-letter misspellings

The solution for this is to implement wildcard searches instead. The simplicity of wildcard search is that it would just involve altering your search algorithm to look for asterisks (*) in place of the dropped/added/replaced letter so that you can cover an entire alphabet's worth of added/dropped/replaced characters for one index of a word with only a single additional word rather than 25. See *Figure 3* for the same example as *Figure 2* using wildcard search instead of brute force.

Note that this covers every case for substitution, insertion, and deletion. In this case, the number of additional letters added to the index for a word of length n is $n(2n + 2)$. This is only a moderate increase compared to the first method of implementation. For a word of length 6 that would be 84 additional characters added to the index; a relatively negligible amount in terms of data. In addition, this particular example also covers character insertion (which *Figure 2* does not).

Another feature we would like to consider in our algorithms is how to rank keyword searches on encrypted data, as discussed by Wang et al. [2]. Essentially, in this form of encrypted search, you are ranking hits by relevance. Relevance is determined by the number of matched keywords per hit.

CASTLE:

{CASTLE, *CASTLE, *ASTLE, C*ASTLE, C*STLE, · · · , CASTL*E, CASTL*, CASTLE*}

Figure 3: Wildcard implementation of fuzzy keyword search

The algorithm these authors came up with involved using term frequency TF (number of matched keywords in the message) and inverse document frequency IDF (number of messages that contain the matched keyword out of total number of messages). Out of hundreds of combinations of TF x IDF they did not find that any one particular formula performed better.

The article also addressed how to create a searchable index at the end of each message. To do this you must scan the file and extract all of the unique words, then calculate the score for each one and store it with the file is identifier in the index. Finally, encrypt everything.

Cao et al. go into some of the same topics as Wang et al. with regards to ranked search [3]. They also detail a few additional factors of encrypted searching. The authors' main priorities are to:

1. Ensure that their search scheme supports multi-keyword matching by relevance
2. To prevent any information leakage from the encrypted data set
3. To meet certain goals on computational overhead

The paper does go into depth on this last point, but hardware efficiency is not the focus of this project. The form of efficiency that the article discusses that does concern our form of encrypted search is how it takes into account storage space for the index. For instance, there are certain ways of building the index (discussed in [1]) that are more size efficient than others.

Golle et al. try to accomplish some of the same things as the previous two articles, but to do so securely. In addition, they also consider conjunctive keyword searches. When searching for multiple keywords (conjunctive keyword searching) you inevitably have to choose between revealing certain information about your encrypted emails to the server and increasing the size of the index exponentially.

This article also further discusses ways of categorizing search data. For instance, you may want to only search through emails marked as "urgent" or under some other category. You can also combine this with other search parameters and only get results for emails from a particular sender, in a particular category, sorted by relevance, etc.

Swaminathan et al. have a similar goal of attempting to provide rank-ordered results from a search on encrypted data [9]. What we found particularly useful about this article was that it does not go in-depth in creating the index and rather was mostly informative about ordering the search results. The authors identify two major stages in ordering data: pre-processing and searching. The pre-processing stage involves ranking and ordering keyword frequency in the index, and the searching stage orders documents in response to a particular query.

Ming Li et al. discuss integrating encrypted search methods over different media than email databases [5]. They use Personal Health Record (PHR) as a case study and examine methods of Authorized Private Keyword Search (APKS) over encrypted cloud data.

In particular, we value the way that this article puts into context the necessity of this topic. It discusses how the cloud database will glean personal information from private health records, and encrypting whatever data that you store on the cloud will prevent this. It assumes that a cloud host is "honest-but-curious" in that it is curious to learn whatever personal information it can about data stored on its servers but that it will follow protocol honestly.

Lu et al. do not so much discuss encrypted search in the capacity that we examine in in this project; rather, they look at indexing multimedia such as videos and images before encrypting and searching [7]. However, this still encompasses several features we hope to include in our encryption schemes. The method used in this article involves a hash table, which is somewhat similar to our encrypted index. The main way that this relates to our project is in the boolean method of searching.

Conclusion

These articles all address our base topic of creating a searchable index, and many build on that by considering security concerns and various types of additional functionality for searching in general. However, nothing exists yet that can search encrypted information securely while maintaining intuitive usability and functionality. We would like to combine all of these unique facets into a superior form of encrypted search.

Proposed Solution

The foundation for our solution to this problem is in the creation of an index appended at the end of each email containing every unique word in the address, subject line, and message. Building this index involves normalizing each word's case and removing all punctuation (hyphens, apostrophes, etc.). In this scenario, when you perform a search on the email database it will perform the same normalization rules to your search term, then encrypt it and scan through each email's own normalized/encrypted index to find any words that match the search term.

We would furthermore like to build on this topic to include support for ranked, fuzzy, and conjunctive keyword searches, while also minimizing loss of security. Adding support for ranked keyword search entails recording the number of matched keywords present in each email that contains at least one match and sorting search results by relevance. Conjunctive keyword search simply means that you can search for a string of words that will each be searched for independently (unless enclosed by quotation marks) and return whichever results contain the greatest number of your search terms, kind of like an extension of ranked keyword search. Fuzzy keyword search is a little bit trickier, and will require finding a balance between functionality and the index's increase in size. This involves adding variations on each unique word in the index to support single- or even multi-character spelling errors. For each level of functionality you add for fuzzy keyword search, the amount of variations of the word you are storing in the index increases exponentially. Fuzzy keyword search will also simultaneously add support for wildcard searching (e.g. searching "Cas*" will return "Castle" as well as any other words that start with “cas”) and minimize certain security leaks. For instance, assuming you do not know the fuzzy keyword indexing algorithms, you will not immediately know the number of unique words in the email.

Finally, we would contextualize this by considering modern network security implications. Google freely admits to collecting data on its users to improve its targeted advertisements. One way that they do this is by automatically scanning your emails for keywords, which they funnel into their algorithms to target ads. If you want to keep your personal information safe from Google, it might be prudent to hide the contents of your emails from Gmail. Likewise, you may want to keep sensitive information safe from your employer or anyone else who may have access to your personal emails. With this form of email encryption, all decryption happens on your computer and no third party will have access to your information.

In terms of setup, we would like to have a mock web server and client server in two separate virtual machines, allowing the client to encrypt and store files on the web server. The client would then be able to run searches to retrieve and decrypt encrypted data using HTTP. It should be noted that the purpose of this project is not to build a fully-functional email server, but rather as a proof of concept. Therefore, success will be measured based on the criteria outlined in *Figure 4* (in order of importance).

1. Encrypted data can be searched without adhering to case-sensitivity
2. Encryption schemes minimize information leakage
3. The search algorithms incorporate additional functionality to increase usability. This functionality will include:
    a. Fuzzy keyword support (common misspellings; add, drop, replace)
    b. Conjunctive keyword support (multiple keywords)
    c. Ranked keyword support (sort by relevance)

Figure 4: Metrics for assessment

Algorithms

It is important to consider the extent to which we will take our index-building algorithms. We do not want to waste space, but at the same time it is important to demonstrate the potential to which wildcard search terms can improve functionality. Therefore, the support for fuzzy-keyword search will only include single-character variations and not full wildcard support.

There are also many common words that will be purposefully excluded from the index in order to save space and improve usability. For instance, a user will most likely never search for the word “it” and if they do they will receive far too many matches to be of any use. So, our algorithm will exclude any word that is less than three characters from the index. We will also match words against a list of common words that nobody would search for. This list will mostly be comprised of three-character words such as “the” but we will not rule out any longer words just yet. See *Figure 5* for a look at the code that will handle building the wildcard index for character replacement and deletion search terms.

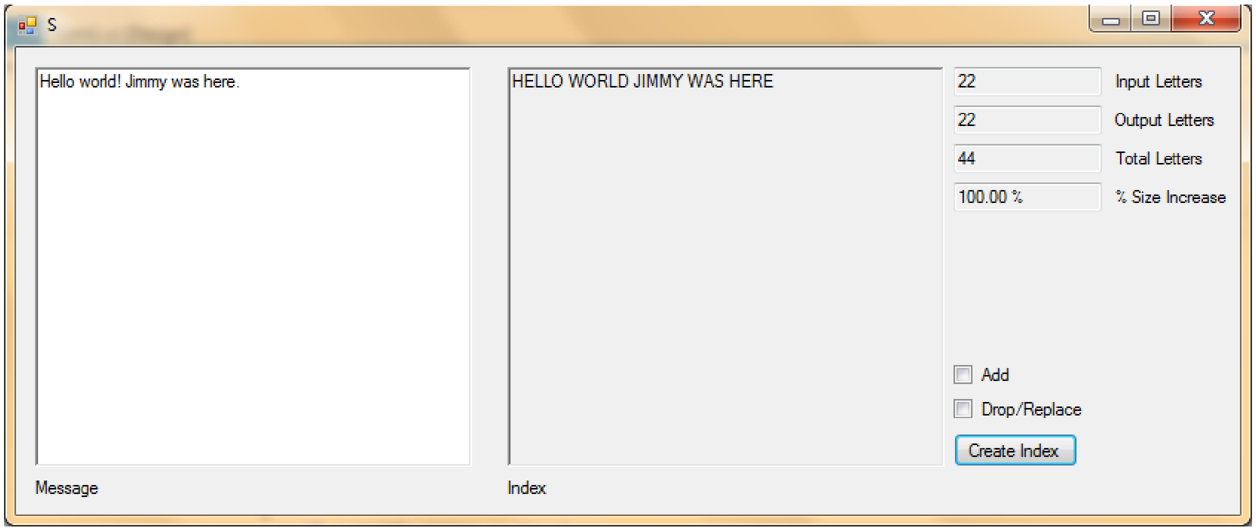
```
private static string[] wildcardReplace(string[] input)
{
    int length = 0; // Size of the array

    // Calculate the size of the array:
    // There will be one new word for each letter in the input, so the
    // array will contain one slot (each slot containing one string) for
    // each character in the input
    foreach (string item in input)
    {
        length += item.Length;
    }

    string[] returnArr = new string[length];    // Will contain the wildcard index terms for
character replacement
    int prevLength = 0;     // Contains the length of the previous word in the index (used
to navigate through the array)
    foreach (string item in input)
    {
        for (int i = 0; i < item.Length; i++)
        {
            // Convert one item in the input to a character array,
            // replace one character with an asterisk, then convert
            // it back into a string and insert in into returnArr
            char[] temp = item.ToCharArray();
            temp[i] = '*';
            foreach (char character in temp)
            {
                returnArr[i + prevLength] += character;
            }
        }
        prevLength += item.Length;  // Increment prevLength by the length of the previous
item
    }
    return returnArr;
}
```

Figure 5: Wildcard character deletion/replacement algorithms

The function for building the index for character insertion search terms is pretty similar except with more temporary variables so it is a bit harder to understand on paper. To observe that code as well as the delete/replace code in action, see *Figures 6 - 9*.



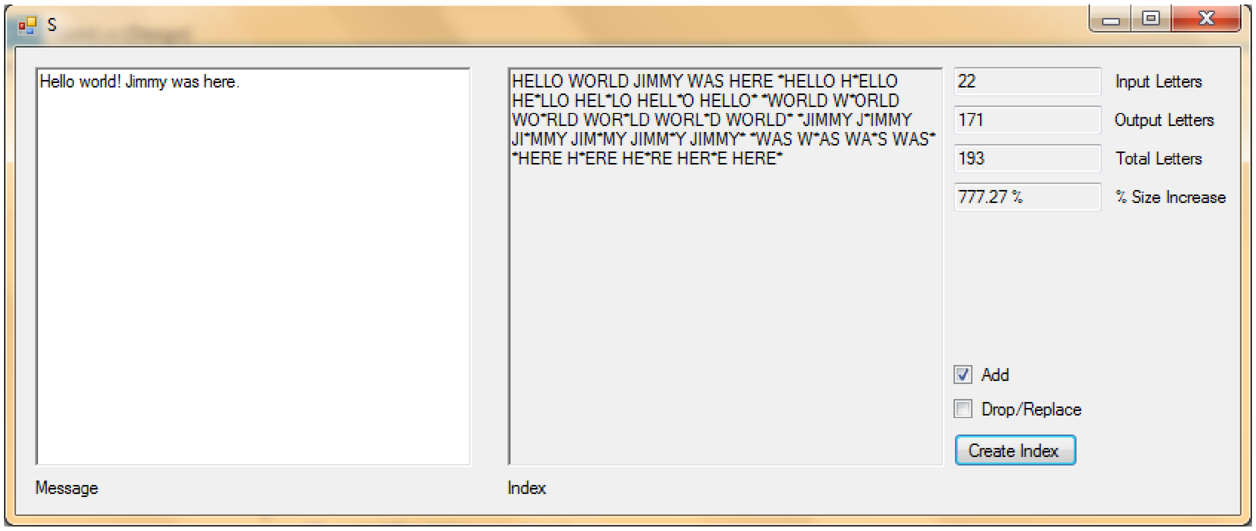
```
Hello world! Jimmy was here.
```

```
HELLO WORLD JIMMY WAS HERE
```

22	Input Letters
22	Output Letters
44	Total Letters
100.00 %	% Size Increase

Add [ ]
Drop/Replace [ ]

Figure 6: Base indexing formula



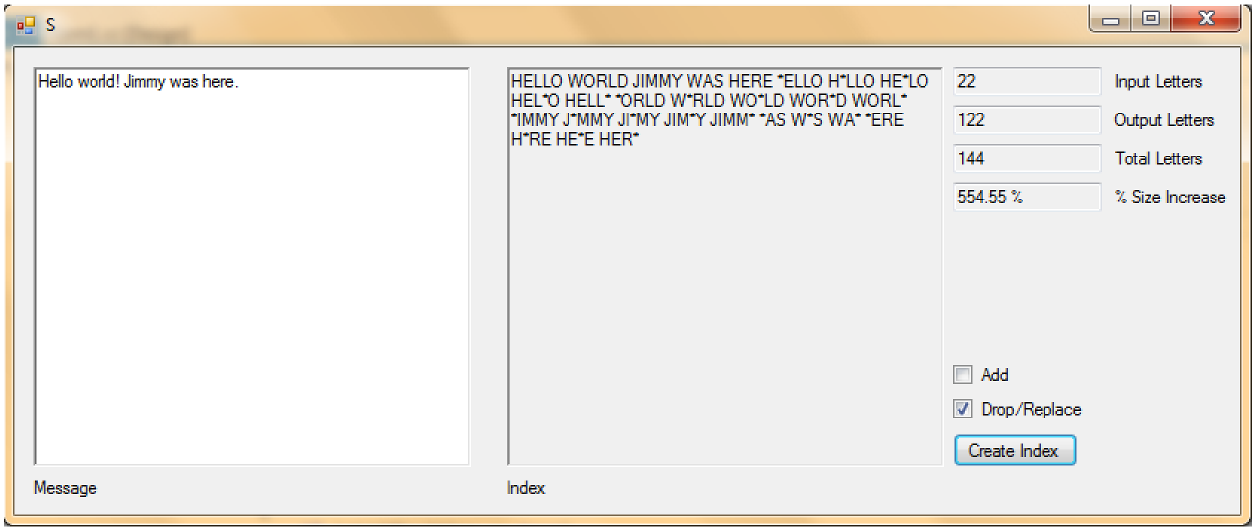
```
Hello world! Jimmy was here.
```

```
HELLO WORLD JIMMY WAS HERE *HELLO H*ELLO
HE*LLO HEL*LO HELL*O HELLO* *WORLD W*ORLD
WO*RLD WOR*LD WORL*D WORLD* *JIMMY J*IMMY
JI*MMY JIM*MY JIMM*Y JIMMY* *WAS W*AS WA*S WAS*
*HERE H*ERE HE*RE HER*E HERE*
```

22	Input Letters
171	Output Letters
193	Total Letters
777.27 %	% Size Increase

Add [✓]
Drop/Replace [ ]

Figure 7: Base + add character indexing formula



Message: Hello world! Jimmy was here.

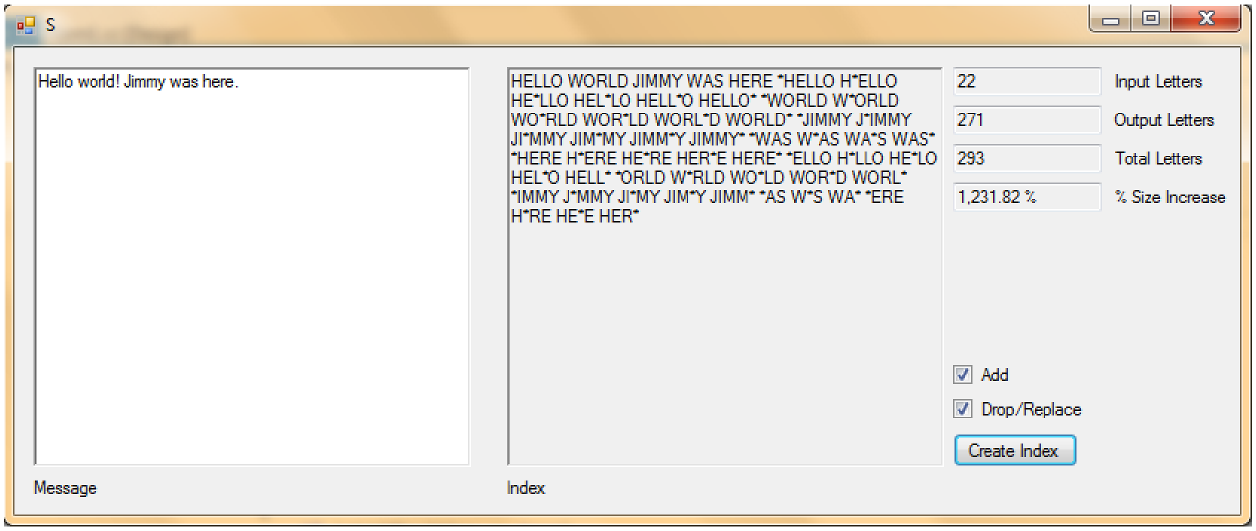
Index:

```
HELLO WORLD JIMMY WAS HERE *ELLO H*LLO HE*LO
HEL*O HELL* *ORLD W*RLD WO*LD WOR*D WORL*
*IMMY J*MMY JI*MY JIM*Y JIMM* *AS W*S WA* *ERE
H*RE HE*E HER*
```

22	Input Letters
122	Output Letters
144	Total Letters
554.55 %	% Size Increase

[ ] Add
[✓ Drop/Replace]

Figure 8: Base + drop/replace indexing formula



Message: Hello world! Jimmy was here.

Index:

```
HELLO WORLD JIMMY WAS HERE *HELLO H*ELLO
HE*LLO HEL*LO HELL*O HELLO* *WORLD W*ORLD
WO*RLD WOR*LD WORL*D WORLD* *JIMMY J*IMMY
JI*MMY JIM*MY JIMM*Y JIMMY* *WAS W*AS WA*S WAS*
*HERE H*ERE HE*RE HER*E HERE* *ELLO H*LLO HE*LO
HEL*O HELL* *ORLD W*RLD WO*LD WOR*D WORL*
*IMMY J*MMY JI*MY JIM*Y JIMM* *AS W*S WA* *ERE
H*RE HE*E HER*
```

22	Input Letters
271	Output Letters
293	Total Letters
1,231.82 %	% Size Increase

[✓ Add]
[✓ Drop/Replace]

Figure 9: Base + add + drop/replace indexing formula

Architecture

To start off, we will build two vagrant virtual machines. The mock web server will be hosted on the first VM using IIS (Internet Information Services) or Apache. The mock client will then be able to perform operations (store & search through data) on the web server using HTTP (Hypertext Transfer Protocol). There is an HTTP extension called WebDAV (Web Distributed Authoring and Versioning) that we would like to utilize here. This will make it possible to upload data to a web server, encrypt it, and search through it from a client.

Timeline

Stage 1: Planning / Design

September-November (complete)

The first phase will begin with an assessment of what the project will include. This means coming up with a definition of the problem and what we will do to address that problem. In outlining how we plan to solve this issue we will reference academic sources to see what has already been done to address it, then try to develop a unique solution that combines ideas from those sources.

Once we have a solution defined, we will plan out how to implement that solution. This will involve developing algorithms to build an index and writing some code to test how those algorithms will work.

Finally, we will lay out hardware specifications and outline the architecture that will be required to design a system to test our solution.

Stage 2: Development

December-March (in progress)

The second phase will include setting up virtual machines and installing web-hosting tools to realize the architecture designs from stage 1. This will also involve learning how to use HTTP and WebDAV. After completing the project's architecture, we will move into programming the actual encrypted search application. This will need to align with our metrics for assessment as outlined in *Figure 4*.

Stage 3: Testing / Implementation

April-May

We plan to leave about a month after completing the development stage to allow for testing and enhancing existing features of the application. At this point the minimum metrics for assessment should be met and the focus will be on improving the overall quality of the project (including architecture, which is not included in the metrics).

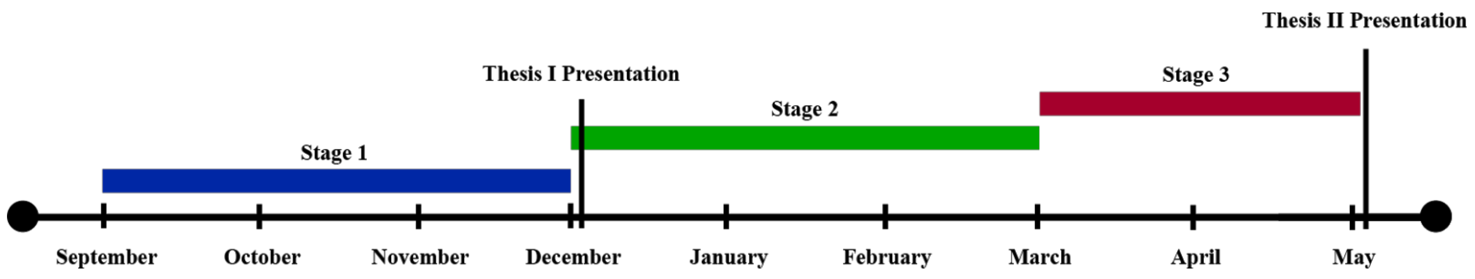


Figure 10: Timeline for completion

Bibliography

1. Li, J., Wang, Q., Wang, C., Cao, N., Ren, K., & Lou, W. "Fuzzy keyword search over encrypted data in cloud computing."*INFOCOM, 2010 Proceedings IEEE*. IEEE, 2010.

2. Wang, C., Cao, N., Li, J., Ren, K., & Lou, W. "Secure ranked keyword search over encrypted cloud data."*Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*. IEEE, 2010.

3. Cao, N., Wang, C., Li, M., Ren, K., & Lou, W. "Privacy-preserving multi-keyword ranked search over encrypted cloud data." *Parallel and Distributed Systems, IEEE Transactions on*25.1 (2014): 222-233.

4. Golle, P., Staddon, J., & Waters, B. "Secure conjunctive keyword search over encrypted data." *Applied Cryptography and Network Security*. Springer Berlin Heidelberg, 2004.

5. Li, M., Yu, S., Cao, N., & Lou, W. "Authorized private keyword search over encrypted data in cloud computing." *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*. IEEE, 2011.

6. Boneh, D., Di Crescenzo, G., Ostrovsky, R., & Persiano, G. "Public key encryption with keyword search." *Advances in Cryptology-Eurocrypt 2004*. Springer Berlin Heidelberg, 2004.

7. Lu, W., Swaminathan, A., Varna, A. L., & Wu, M. "Enabling search over encrypted multimedia databases."*IS&T/SPIE Electronic Imaging*. International Society for Optics and Photonics, 2009.

8. Song, D. X., Wagner, D., & Perrig, A. "Practical techniques for searches on encrypted data." *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*. IEEE, 2000.

9. Swaminathan, A., Mao, Y., Su, G. M., Gou, H., Varna, A. L., He, S., Wu, M., & Oard, D. W. "Confidentiality-preserving rank-ordered search." *Proceedings of the 2007 ACM workshop on Storage security and survivability*. ACM, 2007.

10. Goh, E. J. "Secure Indexes." *IACR Cryptology ePrint Archive* 2003 (2003): 216.