PEKS with Bloom Filter

COM S 453 Project Report

Sumon Biswas

May 5, 2017

1 Introduction

Public Key Encryption with Keyword Search (PEKS) is one of the most used method to search keywords over encrypted data. Suppose, Bob is sending email with specific keywords to Alice. Encrypted emails are stored in the server. Alice wants to search emails with keywords from email server but does not want to allow the server decrypt any email. The paper on PEKS [`http://crypto.stanford.edu/~dabo/papers/encsearch.pdf`] described two algorithms to achieve that goal. The first algorithm takes less time and space compared to the second. However, the first one can not guarantee semantic security. The second one is semantically secure. But dictionary attack can help attackers to guess keywords and pose serious damage. I have resolved that issue using a Bloom Filter. The false positives of a bloom filter does not allow to make it susceptible to dictionary attack. In this project, I have implemented the second algorithm of PEKS that originates form trapdoor permutations. Then I have implemented Bloom Filter that is used to search keywords over the hashmap.

2 PEKS

In this implementation I used a set of public and private keys. Each keyword is associated with one public key and private key. Alice generates set of keys. While sending email Bob acquires the public keys for his keywords from Alice. Then he encrypts the keywords and send it to the server. Alice uses her private key as trapdoor. Then she encrypts a random word for each keyword and send it to the server. The server uses the trapdoor to decrypt the words and matches each one with the given random word. If matches found then it is sent to Alice. The server decrypts the encrypted word but it is not the actual keyword. Rather it is one random word selected by Alice which corresponds to the keyword. By this method, Alice can search over any previously specified keyword without revealing it. The algorithm is given by four methods:

- **KeyGen:** For each $W \in \sum$ run $G(s)$ to generate a new public/private keypair $PK_w/Priv_w$ for the source indistinguishable encryption scheme. The PEKS public key is $A_{pub} = PK_w \mid W \in \sum$. The private key is $A_{priv} \mid W \in \sum$.
- **PEKS**(A_{pub}, W)**:** Pick a random $M \in 0, 1^s$ and output $PEKS(A_{pub}, W) = (M, E[PK_w, M])$, i.e., encrypt M using the public key PK_w
- **Trapdoor**(A_{priv}, W)**:** The trapdoor for word W is simply $T_w = Priv_w$
- **Test**(A_{pub}, S, T_w)**:** Test if the decryption $D[T_w, S] = 0^s$. Output 'yes' if so and 'no' otherwise.

3 Bloom Filter

Bloom filter is used to check membership in a set. In this case, every keyword has a corresponding word which is encrypted in the server. When Alice wants to search some keyword she sends the *dummy* word to the server. The sever stores the decrypted keywords in the map of the filter. The server checks the membership of the word using Bloom Filter.

This filtering technique need some work in the server. When PEKS send the encrypted words in the server it decrypts it and stores it to the filter's map. To build the map the keyword is hashed a number of times and then the hash is indexed to a position of the vector. Initially the bit vector is set to false. When inserting one keyword the index position of each hash value is set to true. While membership checking the server performs check to each index of the bit vector and outputs the result.

Since Bloom Filter allows false positives, we will get some 'yes' even if the keyword is not present in the server. The number of hashes and the size of the bit-vector changes the false positive rate. Generally, the number of hashes increase the accuracy while the bit vector is big enough to hold all the value. Similarly, the increase of the bit-vector also reduces the number of false positives. The steps we followed to work bloom filter work with PEKS:

1. Create a bit vector initializing all bits to *false*.
2. Receive the encrypted keyword store which might be searched.
3. Decrypt each one and perform insertion to the vector.
4. When a request is sent with a *dummy* word, the membership is checked in the filter.
5. The 'yes'/'no' result is sent to the client.

4 Implementation

I have implemented the whole project in python. I used 256-bit elliptic curve cryptography to generate public and private key-pair for each word. Then I used Diffie-Hellman key exchange protocol to obtain a shared-key for Alice and Bob. Then each party can encrypt and decrypt using the secret key efficiently. Since, encryption and decryption is linear to the number o ftotal number of keywords, so the shared key helps to reduce the time for the PEKS.

After obtaining a shared key by Alice and Bob, then we use AES algorithm to encrypt and decrypt keywords. I used the *Crypto* module from python library for AES.

To implement the Bloom filter, I used the JHASH hash function originally created by Bob Jenkins. Then I randomized the hash function and used it to build the Bloom Filter. Finally, a python script generates the plot using the stored results. To run the project just the requirements.txt and run peks.py.

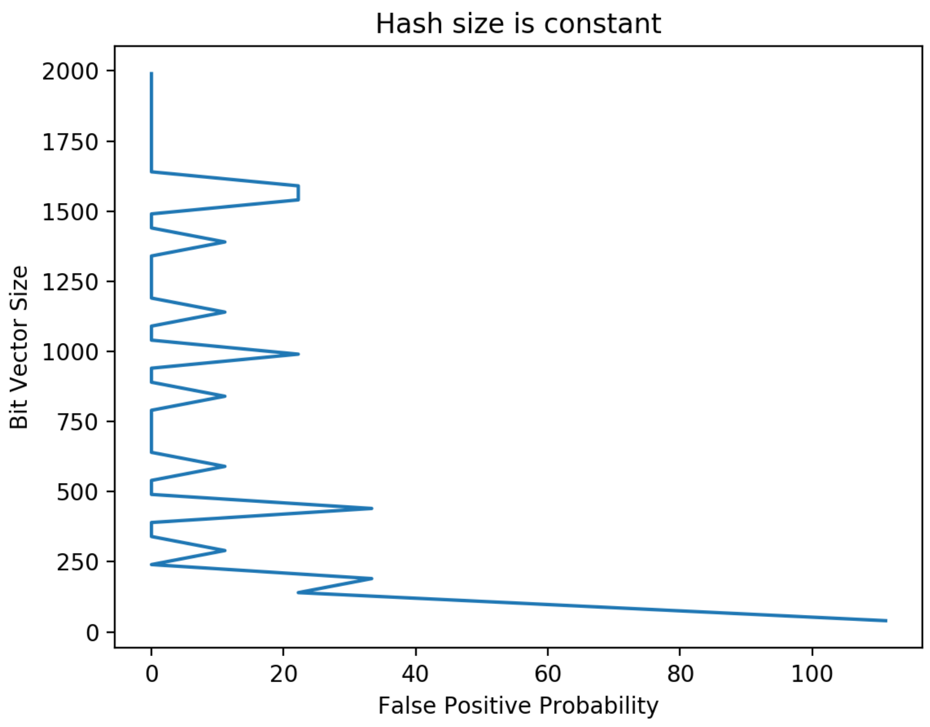

Figure 1: False positive rate while changing bit vector size

5 Result

I used two text files as inputs:

1. Keyword dictionary: This files stores all the keywords that Alice wants to search. She publishes key-pair for each of the keywords in this file.

2. Keyword store: This files contains a subset of the keywords of the dictionary. Bob send emails containing these keywords. So, only these keywords search will return 'yes'.

We can update this files to specify more keywords. Note that, no repetition should be used in this files. That will refer to generate more than one set of keywords. The result is stored in another text file (result.txt). I have executed several iterations with different values of vector size and number of hashes. The number of false positive outputs with other parameters are printed in this file.

First, we fixed the number of hashes to a moderate number so that the false positive rate is not very large. We observed that if the bit-vector size is very low then the false positive rate reaches about 100%. However, after a certain increase the false does not tend to reduce any more. In fact, if we will not want to reduce the false positive rate very low either.

Second, we fixed the size of the bit vector and increased the number of hashes gradually. The result is plotted in the following figure. It suggests that the if we use a high number of hashes the false positive rate may reach 100%. Therefore, the client will decide the number of hashes depending upon how much false positive she allows. Note that, a very low number of hashes might make the filter susceptible to frequency attacks.

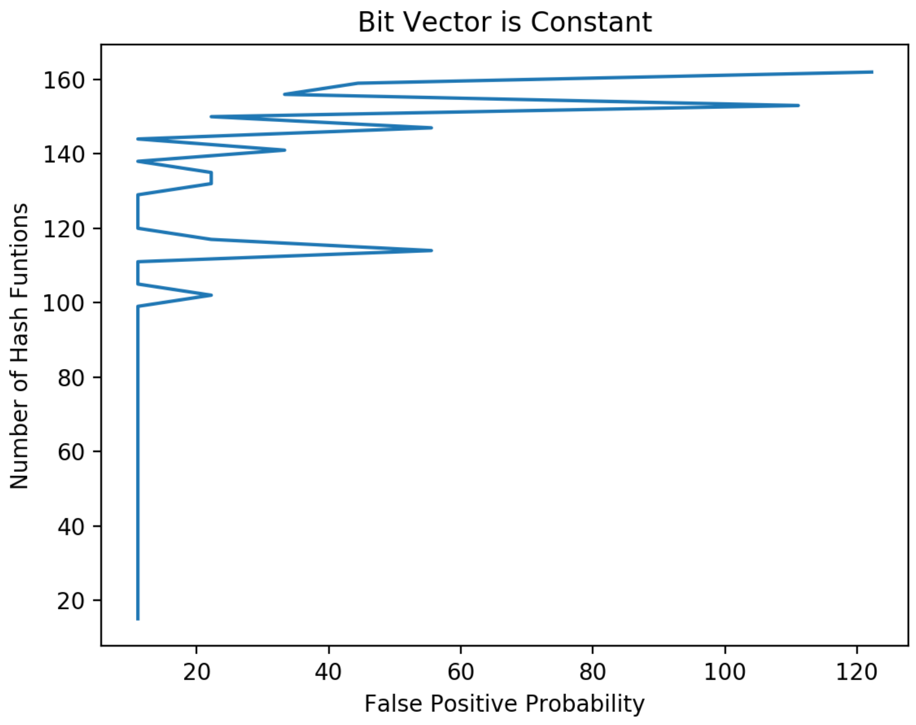

Figure 2: False positive rate while changing number of hashes

6 Conclusion

The main goal of the project was to extend PEKS. I assumed that space is not a problem to trade security. Therefore, I implemented the version of PEKS which is semantically secure and used comparatively simple proof of security. However, by dictionary attack one can guess the frequency of keywords searched by Alice. To combat that issue I designed a Bloom Filter which works with PEKS. After sending the trapdoor to the server, instead of searching the whole dictionary the server keeps track of the filter and outputs the result. The number of false positive results are important because thus we prevent attacker from frequency attack. The number of bit vector size and number of hashes affect the false positive rate. Finally, I analyzed the result to show the change of false positives with the size of bit-vector and number of hashes.