# Internship Report
# Pre-filtering solutions for private encrypted matching in publish/subscribe systems

**Nguyen Viet Sang**
viet-sang.nguyen@etu.unilim.fr

Master 1 Cryptis, Faculty of Sciences and Techniques, University of Limoges

**Abstract.** This report studies an application of Cuckoo Filter and Bloom Filter in publish/-subscribe systems using encrypted matching. We implement an encrypted matching protocol and successfully embed these types of filters into the broker. The results show that these filters save much time by early discarding costly matching operation. Besides, Cuckoo Filter's performance is better than Bloom Filter's with nearly 2x faster in inserting and searching items.

## 1 Introduction

Publish/subscribe systems allow us to spread information from publishers to different subscribers. Data produced by publishers is in the form of publications. Subscribers express their interests for receiving publications by issuing subscriptions composed some constraints. The matching operation can be topic-based or content-based and is executed by a set of dedicated machines call the brokers. Topic-based is an equality condition which is simply an exact comparison between topic of a publication and topic of a subscription. Content-based is more complex which can consist of several inequality conditions as well as equality conditions. Publishers and subscribers exchange messages through publish/subscribe systems, hence these untrusted systems can try to learn information of publications and subscriptions. This violates the confidentiality of information and appeals a solution for privacy-preserving.

To avoid the leakage of information in untrusted systems, it requires to perform matching operation on encrypted data. In a traditional broker, the matching involves a publication $p$ and a subscription $s$. Encrypted matching must ensure that the operation on their ciphertexts $(E(p), E(s))$ returns the same result as the matching on $p$ and $s$. However, this encrypted matching normally costs a considerable amount of time because it involves many complicated computations. For that reason, pre-filtering is proposed to reduce number of calls to this costly matching operation. By applying this technique, the executions of matching which are surely known to be unsuccessful are early discarded.

In this work, we analyse and implement an encrypted matching protocol in order to preserve the confidentiality of information exchanged in publish/subscribe systems. Then, we apply pre-filtering technique to quickly discard unnecessary encrypted matching operation. Our problem is limited to a matching operation between two topic names. We also experiment with different types of filters such as Bloom Filter, Cuckoo Filter. The results show that encrypted matching operation is very costly and pre-filtering technique helps to save much time by its early ignorance. Besides, Cuckoo Filter performs exceptionally well compared to Bloom Filter with nearly 2x faster.

This report is organised as follows:

- Section 2 recalls the ideas of Bloom Filter and Cuckoo Filter. We show the comparison of their performances.
- Section 3 analyses the details of an encrypted matching protocol which is implemented in this work.
- Section 4 analyses the way that we embed the idea of pre-filtering with Bloom Filter and Cuckoo Filter into publish/subscribe systems.

- Section 5 shows the details of implementation and our main results. We also give some discussion about the results.
- Section 6 concludes our work.

## 2 Bloom Filter and Cuckoo Filter

Pre-filtering aims to reduce number of calls to the encrypted matching operation which costs so much time. The principle of this idea is to embed several filters into the broker. There are two types of filters that can be considered: Bloom Filter [4] and Cuckoo Filter [7]. Both are data structures designed to tell us, rapidly and memory-efficiently, whether an item is present in a set. They can return two possible results: the item is *definitely* not in the set or *may be* in the set. The second possibility can raise a true positive or false positive. True positive means that the item is actually in the set. False positive means that the item is actually not in the set, however the filtering algorithm returns an answer of existing in the set.

In this section, we present the principles of Bloom Filter and Cuckoo Filter. We also use figures to depict how it works. Moreover, results of an experiment are shown to compare the performances of these two filters.

### 2.1 Principle of Bloom Filter

Bloom Filter is probabilistic data structure that allow for efficient testing of whether or not an item belongs to a set. Essentially, a Bloom Filter is a bit array. When adding an item, one or several hash functions are used to identify bits in the filter that must be set to 1. To test whether an item belongs to a set, it is also hashed. If the corresponding bits are 1, the set likely includes it. Otherwise, it is guaranteed not to be in that set. The accuracy of Bloom Filter can be tuned by properly choosing the size of bit array and the number of hash functions.

Bloom Filter is based on some simple hash functions such as MurmurHash [2], FNV [8]. Because of the simplicity of hash functions, this filter is extremely efficient in terms of time complexity which is $\mathcal{O}(k)$ for both adding and checking an item, where $k$ is number of hash functions. Besides, Bloom Filter only requires a very small space in memory (an array of bits). Figure 1 shows an example of how this type of filter works.

In usage, we need to provide two parameters, including estimated number of maximum items ($n$) and false positive rate ($p$), in order to be able to create a new Bloom Filter. Then, the size of bit-array ($m$) can be computed by

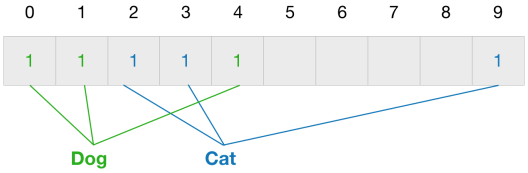
$$m = -\frac{n \ln p}{(\ln 2)^2}$$

And the number of hash functions ($k$) can be computed by

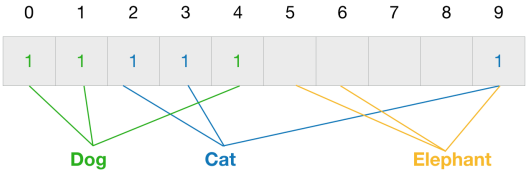
$$k = \frac{m}{n} \ln 2$$

### 2.2 Principle of Cuckoo Filter

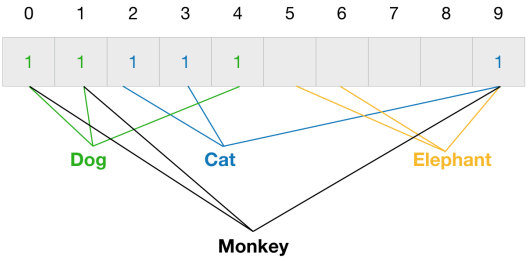
Although Bloom Filter is widely used for membership tests, it does not permit deletion of items from the set without rebuilding the entire filter. There are several versions of improvements, such as Counting Bloom Filter and Quotient Filter, based on a standard Bloom Filter in order to support deletion. However, these approaches need significant space or have worse performances. Meanwhile, Cuckoo Filter also supports deleting operation besides inserting and searching. It is reported to have a higher performance than Bloom Filter in terms of searching operation. Besides, according to the




(a) Adding items “Dog” and “Cat” to the filter. After hashing the item “Dog” with 3 hash functions, the results are (0, 1, 4). Bits at those positions are set to be 1. Similarly, the item “Cat” is added to the filter.




(b) Searching item “Elephant” in the filter. This item is definitely not present in the filter because there exists a bit which is not equal to 1 (at positions 5 and 6).




(c) Searching item “Monkey” in the filter. The result is that this item may be in the filter since the corresponding bits are 1, but it is actually not present in the filter. This is a false positive.

Fig. 1: An example of adding and searching items with Bloom Filter. This uses 3 hash functions and the size of bit array is 10.

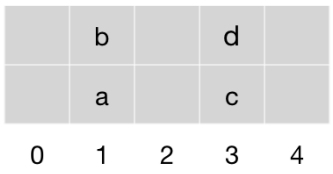
authors, Cuckoo Filter is easier to implement than alternatives such as the quotient filter, and it uses less space than Bloom Filter in many practical applications if the target false positive rate is less than 3%. In this work, we mainly focus on the advantage of speed.

For each item inserted, this kind of filter only stores its fingerprint which is a bit string derived from the item using a hash function. A membership query for item $x$ simply searches the hash table for the fingerprint of $x$, and returns true if an identical fingerprint is found. When constructing a Cuckoo Filter, its fingerprint size is determined by the target false positive rate. The smaller value this rate is, the longer fingerprint we need. The substantial difference between Cuckoo Filter and regular hash table is that only fingerprints are stored in the filter and it is impossible to retrieve the original key as well as value bits of each item.

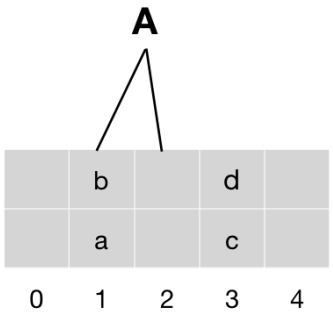
***Data structure:*** This filter uses a basic hash table consisting of an array of buckets. Each bucket has one or several entries. A fingerprint is stored in an entry. For example, figure 2 shows a Cuckoo Filter with 5 buckets (capacity) and 2 entries per each bucket (bucket size). An item has two candidate buckets determined by two hash functions $h_1(x)$ and $h_2(x)$. Both of these two buckets are checked when looking for the existence of a certain item.

There are three main operations of Cuckoo Filter that we describe in this section: Insert, Lookup and Delete. Practically, It just uses a hash function ($h_2(x)$ is derived from $h_1(x)$) and an array of buckets as the data structure. Fingerprint $f$ of an item is stored in a bucket.

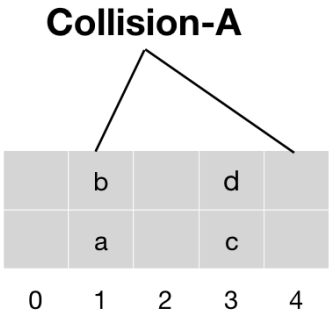
– **Insert:**



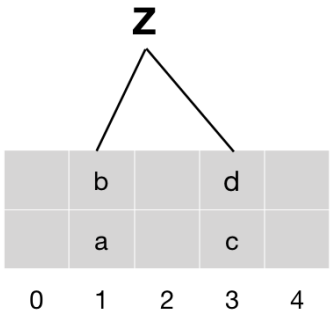

(a) Adding items “A”, “B”, “C”, “D” to the filter. $i_1(A) = i_1(B) = 1$, $i_1(C) = i_1(D) = 3$, $i_2(A) = i_2(B) = 2$, $i_2(C) = i_2(D) = 4$.



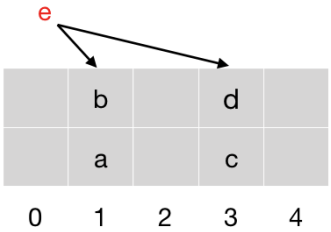

(b) Searching item “A”. Since $i_1(A) = 1, i_2(A) = 2$, it lookups in buckets of elements 1, 2 and compares the fingerprint of “A” with all fingerprints in these buckets.



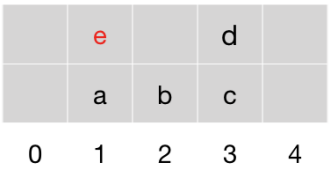

(c) Searching item “Collision-A”. Suppose this item has the same fingerprint with “A” and its results of hashing are $i_1 = 1, i_2 = 4$. This returns a false positive.




(d) Searching item “Z” which has $i_1(Z) = 1, i_2(Z) = 3$. Its fingerprint “z” is not the same with any fingerprint in corresponding buckets. Therefore, “Z” is definitely not in the filter.




(e) Adding item “E”. Suppose $i_1(E) = 1, i_2(E) = 3$ and buckets at these positions are full. It randomly choose a bucket in element $i_1(E)$. Suppose “b” is chosen to be kicked out.




(f) “b” is moved to a bucket in $i_2(B) = 2$. Then, “e” moves to the old bucket of “b”.

Fig. 2: An example of operations on Cuckoo Filter. This filter has capacity of 5 and bucket size of 2. Notice that “x” is fingerprint of “X”

In standard cuckoo hashing, when the two candidate buckets of an item are both occupied, it is necessary to access the original existing items in order to determine where to relocate them and hence, make room for the new ones. However, in this type of Cuckoo Filter, the only things we store are fingerprints of items and therefore, there is no way to restore or rehash the original keys to find their alternate positions. The authors of Cuckoo Filter proposed a technique called “partial-key cuckoo hashing” to derive an item’s alternative location based on its fingerprint. For

an item $x$, we firstly calculate the indexes of the two candidate buckets as follows:

$$\begin{aligned} i_1 &= h_1(x) = hash(x) \\ i_2 &= h_2(x) = i_1 \oplus hash(f) \end{aligned} \tag{1}$$

where $f$ is the $x$'s fingerprint.

We can obserse that $i_1$ can be calculated from $i_2$ and the fingerprint $f$ by using the same formula. This means that to kick out a fingerprint in bucket $i$, no matter if $i$ is $h_1(x)$ or $h_2(x)$, we can calculate its alternate bucket $j$ from the current bucket index $i$ and the fingerprint stored in this bucket by

$$j = i \oplus hash(f)$$

Hence, we never need to retrieve the original $x$ in an insertion. This important improvement in Cuckoo Filter is based on the property of x-or operation. Furthermore, the items are distributed uniformly in the hashing table since the fingerprint is hashed before being xor-ed with the index $i$. If the alternate index of bucket is directly calculated without hashing the fingerprint, the items which are kicked out from nearby buckets will locate close to each other in the table, if the fingerprint size is small compared to the size of table. This also ensures that items can be relocated to entries of buckets in different parts of the table. Hence, it reduces the problems of hash collisions.

Because the size of fingerprint is shorter than the ones of $h_1$ and $h_2$, the are two possible consequences. Firstly, the total number of possible values of $(i_1, i_2)$, as a result, is much smaller than the ones using a perfect hash as standard cuckoo hashing. This leads to more collisions. Secondly, there is no problem when adding two different items that have the same fingerprint. However, it can insert the same item maximum $2b$ times, where $b$ is number of entries in each bucket (bucket size).

Figure 2a shows an example of inserting items to a Cuckoo Filter. We notice that in this case, fingerprints are prioritised to store in the buckets $i_1$. We consider figures 2e and 2f which show an example of kicking operation.

– **Lookup:**

The lookup procedure is very simple. Given an item $x$, we firstly calculate its fingerprint and the two corresponding candidate buckets by the equation 1. If any entry of these two buckets contains the same fingerprint, the result will be true. Otherwise, we will receive the result of false. Notice that false positive can occur. It means an item is actually not present in the filter, but the returned result is true. However, false negative never happen in a Cuckoo Filter.

Figure 2b shows an example of lookup in a Cuckoo Filter. This is a normal case where an item is added before and hence, the result of searching that item is true. An example of false negative (the item is definitely not in the filter) is shown in figure 2d. This case returns false because the fingerprint "z" does not exist in the filter. Figure 2c shows an example of false positive. In this case, we suppose the item "Collision-A" has the same fingerprint ("a") with the item "A". They are definitely two different items, however, we still receive the result of true because of the collision.

– **Delete:**

The process of deletion is simple and similar to the lookup. Given an item $x$, we firstly calculate its fingerprint and the two corresponding candidate buckets by the equation 1. If any fingerprint in these two buckets matches the $x$'s fingerprint, it will be removed from that bucket. Notice that if there are other fingerprints that are the same as $x$'s, this procedure does not remove all of them. For example, items $x$ and $y$ reside in the bucket $i_1$ and collide on the fingerprint $f$. When $x$ is deleted, it does not matter if the process removes the fingerprint $f$ inserted when adding $y$ or $x$. After deleting $x$, $y$ is still perceived as a membership because there is one corresponding fingerprint left in bucket $i_1$. As a consequence, false positive will happen when we look for $x$ again.

This is an important point in the work of Cuckoo Filter. However, in this work, this situation is not helpful. We discuss this problem in the next sections.

### 2.3 Comparison between Bloom Filter and Cuckoo Filter

We also setup an experiment to compare performances of Bloom Filter and Cuckoo Filter. Table 1 shows time and memory taken by each type of filters. Maximum number of items that each filter can contain are fixed to be $400,000$. For the Bloom Filter, we configure the false positive rate to be 0.1. For the Cuckoo Filter, we experiment on different values of capacity and bucket size to see the difference. Capacity gradually decreases from $100,000$ to $25,000$, and bucket size increases from 4 to 16, respectively.

As we can see from table 1, Bloom Filter seems to perform exceptionally well in terms of space. It only needs 0.47MiB while Cuckoo Filter needs 26.43MiB corresponding to its capacity of $100,000$ and bucket size of 4. For Cuckoo Filter, experiment shows that the larger bucket size, the smaller space it costs. When the bucket size is 16, the Cuckoo Filter occupies 6.68MiB. In terms of time, Cuckoo Filter takes nearly 0.3s to insert $50,000$ items, while Bloom Filter takes over 0.5s. In the lookup operation, the same thing happens. Cuckoo Filters consumes over 0.25s while Bloom Filter needs an amount of nearly 0.5s. In summary, Cuckoo Filter is nearly 2x faster than Bloom Filter. This suggests us to apply Cuckoo Filter in publish/subscribe system if time is considered as an important factor.

| Parameter | Bloom Filter | Cuckoo Filter | | |
|---|---|---|---|---|
| Capacity | / | 100,000 | 50,000 | 25,000 |
| Bucket size | / | 4 | 8 | 16 |
| Fingerprint size (bytes) | / | 2 | 2 | 2 |
| Max elements | 400,000 | / | / | / |
| False positive rate | 0.1 | / | / | / |
| Memory (MiB) | 0.47 | 26.43 | 13.15 | 6.68 |
| Time of insertion (s) | 0.504 | 0.292 | 0.288 | 0.290 |
| Time of searching (s) | 0.492 | 0.263 | 0.259 | 0.268 |

Table 1: Performances of Bloom Filter and Cuckoo Filter. For comparisons of inserting and searching operations, we experiment on $50,000$ items on each operation and measure time taken by the filters.

## 3 Encrypted matching in publish/subscribe systems

In a publish/subscribe system, the communication between publishers and subscribers is outsourced to a third party which is an untrusted server called "broker". By this way, the broker can try to retrieve information exchanged among clients without permissions. It leads to information leakage and appeals a solution for this risk. Therefore, it is necessary to encrypt messages before they are sent to the broker. This also requires the broker to be able to match publications and subscriptions based on their ciphertexts instead of normally exact comparisons. In short, we need to replace the traditional method of matching by a new protocol involving encryption in order to preserve the privacy of clients.

### 3.1 Overview

There are some solutions that have been proposed to solve the problem of privacy-preserving in publish/subscribe systems. Barazzutti et al. [3] proposed a mechanism which reduces number of calls to encrypted matching operation based on pre-filtering. The authors applied Bloom Filter for pre-filtering stage and solved the general problems of containment conditions and content-based matching. For the encrypted matching scheme, they used the idea of Asymmetric Scalar-product Preserving

Encryption (ASPE) proposed by Choi et al. [5]. This aims to route publications to the appropriate subscribers without letting the brokers learn any information about publications and subscriptions. Besides, the encrypted matching protocol of Dong et al. [6] is an interesting scheme. They proposed an encryption scheme where each authorised user in the system has his own keys to encrypt and decrypt data. The scheme supports keyword search which enables the server to return only the encrypted data that satisfies an encrypted query without decrypting it. In our work, we utilise this solution to implement the encrypted matching operation. The details of this protocol are presented in the next section.

### 3.2 Encrypted matching scheme in usage

In this work, we follow the work of Ion et al. [9] which uses the scheme of Dong et al. [6]. Suppose that client $i$ firstly subscribes to the topic $w_i$. Then, client $j$ publishes messages on the topic $w_j$. Normally, if $w_i$ is equal to $w_j$, the broker will match these publication and subscription. We now consider a protocol in which these two clients only send the ciphertexts of $w_i$ and $w_j$, then the broker will match the encrypted messages instead of $w_i$ and $w_j$. There are four party involving in a publish/subscribe system, including two clients, a key management server and a broker, as shown in figure 3. The procedure of encrypted matching operation is as follows.

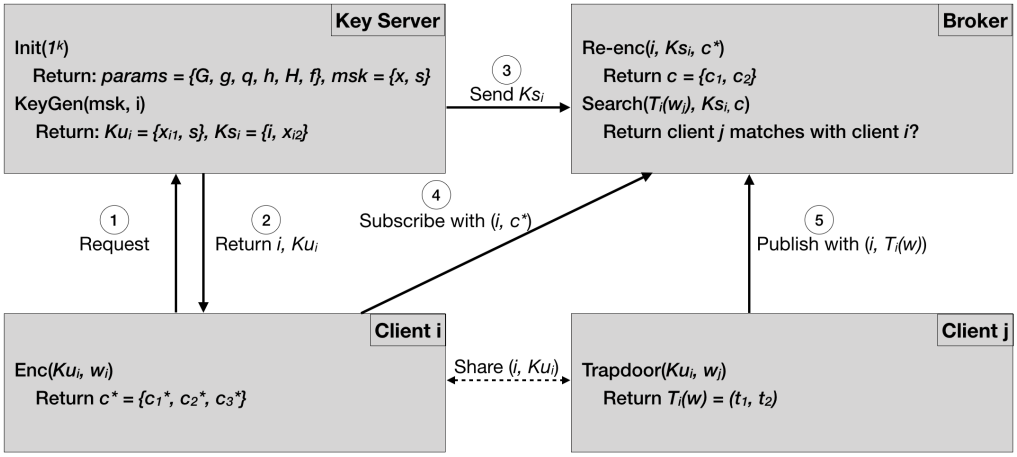



Fig. 3: Procedure of encrypted matching

1. Client $i$ sends a request of key along with a security parameter $(1^k)$ to the key server. Then, this server runs the following two functions:
   - $\mathsf{Init}(1^k)$: initialise a El Gamal encryption scheme. Choose two prime numbers $p$, $q$ such that $q$ divides $p-1$, a cyclic group $\mathbb{G}$ with a generator $g$ such that $\mathbb{G}$ is the unique order $q$ subgroup of $\mathbb{Z}_p^*$. Then, choose $x \xleftarrow{R} \mathbb{Z}_q$ randomly and compute $h = g^x$. Choose a collision resistant hash function $H$, a pseudorandom function $f$ and a random key $s$ for $f$.
     The outputs are $params = (\mathbb{G}, g, q, h, H, f)$ and master key $MSK = (x, s)$.
   - $\mathsf{KeyGen}(MSK)$: generate and assign an identifier $i$ for this client. Choose a random number $x_{i1} \xleftarrow{R} \mathbb{Z}_q$, then compute $x_{i2}$ as follows:

     $$x_{i2} = x - x_{i1}$$

     The outputs are $Ku_i = (x_{i1}, s)$ and $Ks_i = (i, x_{i2})$ which are the two packets that need to be sent to user $i$ and server of broker, respectively.

2. Key server sends the identifier $i$ and $Ku_i$ to the client $i$. Then, this client encrypts the keyword $w_i$ describing the topic that he wants to subscribe by the following function:

   – $\mathsf{Enc}(Ku_i, w_i)$: The user chooses a random number $r \xleftarrow{R} \mathbb{Z}_q$, the keyword $w_i$ is encrypted as follows:

   $$c^*(w_i) = (c_1^*, c_2^*, c_3^*)$$

   where

   $$\begin{aligned} c_1^* &= g^{r+\sigma_i} \\ \sigma_i &= f_s(w) \\ c_2^* &= (c_1^*)^{x_{i1}} \\ c_3^* &= H(h^r) \end{aligned}$$

   The output is $c^* = (c_1^*, c_2^*, c_3^*)$.

3. Key server also sends $Ks_i$ to the broker and the broker keeps this key unique for client $i$.

4. Client $i$ sends $(i, c^*)$ as his subscription to the broker. This information is used in matching operation whenever a new publication arrives to the broker.

5. Client $j$ sends $(i, T_i(w_j))$ as his publication to the broker in order to express his desire to match with client $i$. Note that the client $i$ shares the information of $(i, Ku_i)$ with the client $j$. Then, client $j$ can compute $T_i(w_j)$ by the function below:

   – $\mathsf{Trapdoor}(Ku_i, w_j)$: Choose a random number $r \xleftarrow{R} \mathbb{Z}_q$ and compute $T_i(w_j)$ such that

   $$\begin{aligned} t_1 &= g^{-r} g^{\sigma_j} \\ t_2 &= h^r g^{-x_{i1} r} g^{x_{i1}\sigma_j} \end{aligned}$$

   where

   $$\sigma_j = f_s(w_j)$$

   The output is $T_i(w_j) = (t_1, t_2)$.

   Once receiving the publication from client $j$, the broker firstly retrieves $Ks_i$ of the client $i$ whom the client $j$ wants to connect with. Then, it executes the encrypted matching operation by the two functions as follows:

   – $\mathsf{Re\text{-}enc}(Ks_i, c^*)$: Compute $c = (c_1, c_2)$ such that

   $$\begin{aligned} c_1 &= (c_1^*)^{x_{i2}} c_2^* = (c_1^*)^{x_{i1}+x_{i2}} = (g^{r+\sigma_i})^x = h^{r+\sigma_i} \\ c_2 &= c_3^* = H(h^r) \end{aligned}$$

   The output is $c = (c_1, c_2)$

   – $\mathsf{Search}(T_i(w_j), Ks_i, c)$: Compute $T = t_1^{x_{i2}} t_2 = g^{x\sigma_j}$. Check if $c_2 = H(c_1 T^{-1})$ or not. If yes, the publication matches with the subscription, and otherwise.

The procedure presented above is for the case in which the subscription arrives to the broker before the publication. For the case of conversion, we can easily swap the role of client $i$ and $j$ in the figure 3. In summary, the encrypted matching procedure above can replace the traditionally equality comparisons. Although this ensures the preservation of privacy for users, it needs so much time to compute. We discuss this problem in the next section.

## 4 Pre-filtering in publish/subscribe systems

Because of the considerable amount of time taken by the encrypted matching operations, it is better to pre-discard the execution of these operations whenever we know that they definitely do not match. In other words, when the broker receives a new request (a new publication or a new subscription), it firstly pre-filters with the existing corresponding lists of subscriptions or publications in order to check whether this request might match or definitely do not match. The broker goes to the encrypted matching operations if and only if the first possibility happens. Otherwise, it discards the request. This idea suggests us to utilise a kind of filter such as Bloom Filter or Cuckoo Filter.

Recall that Bloom Filter and Cuckoo Filter have the same purpose. Their target is to check that if an item exists in a filter or not. We can receive two possible answers: that item may be in the filter or that item is definitely not in the filter. There are two essential operations on each type of these filters including inserting an item to the filter (1) and searching an item in the filter (2). For Cuckoo Filter, there is additionally an operation of deleting an item from the filter.

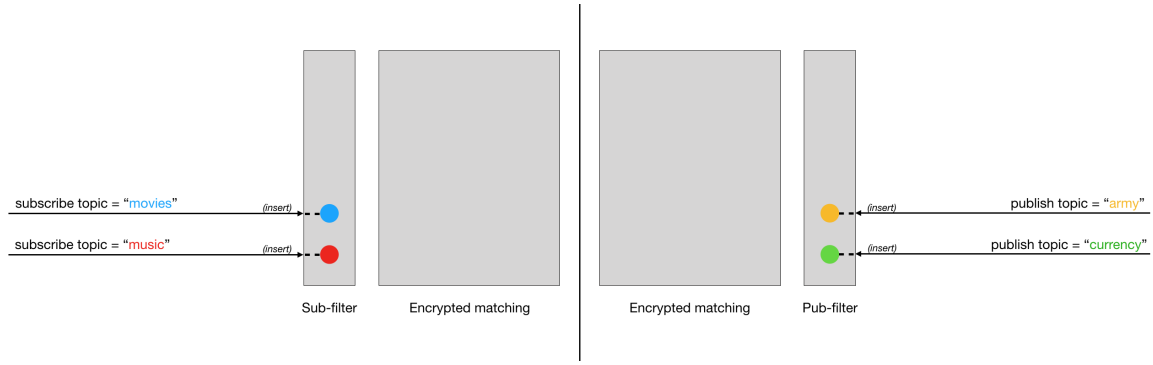
In this work, we use two filters in the broker for our pre-filtering purpose (pub-filter and sub-filter) as shown in figure 4. Pub-filter contains topics of different publications. Meanwhile, sub-filters contains topics of different subscriptions. These two filters operate in a similar way in which sub-filter is used to check topics with new publications and pub-filter is used to check topics with new subscriptions. We consider two possible cases that the broker can be faced with: a client publishes a message under a certain topic, then another client subscribes that topic and vice versa, the subscription is committed before the publication.

- **Matching a new subscription:** When the broker receives a new request of subscription, it firstly inserts the topic of this subscription into the sub-filter (figure 4a, left). Then, it searches in the pub-filter to find that if this topic is present or not. If yes, it means there maybe a publication which has the same topic in the pub-filter at that time and the broker will execute the encrypted matching operation (figure 4c, right). Otherwise, it means there is definitely no publication which has the same topic with that subscription, and hence the broker will discard the encrypted matching operation (figure 4b, right). Algorithm 1 shows the steps of this procedure.

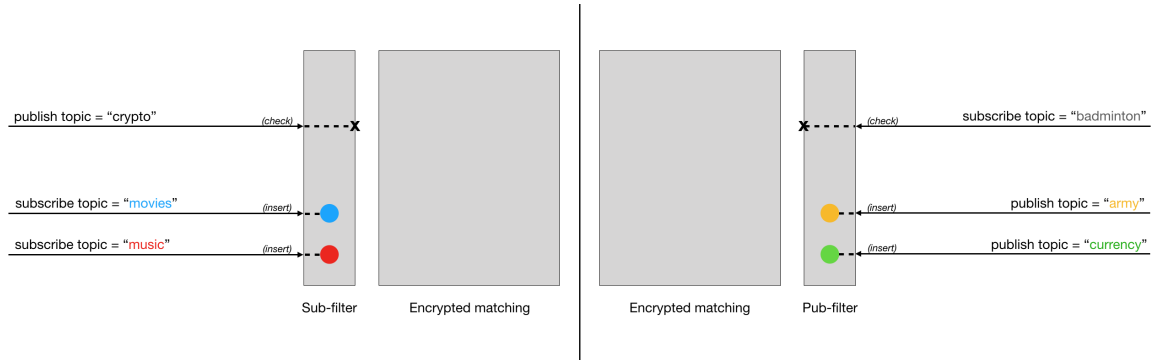
**Algorithm 1:** Pre-filter a new subscription $s$

```
Add s to sub-filter;
if s may be in pub-filter then
    for each p in list of publications do
        execute encrypted matching operation (p, s);
    end
else
    discard encrypted matching operation;
end
```

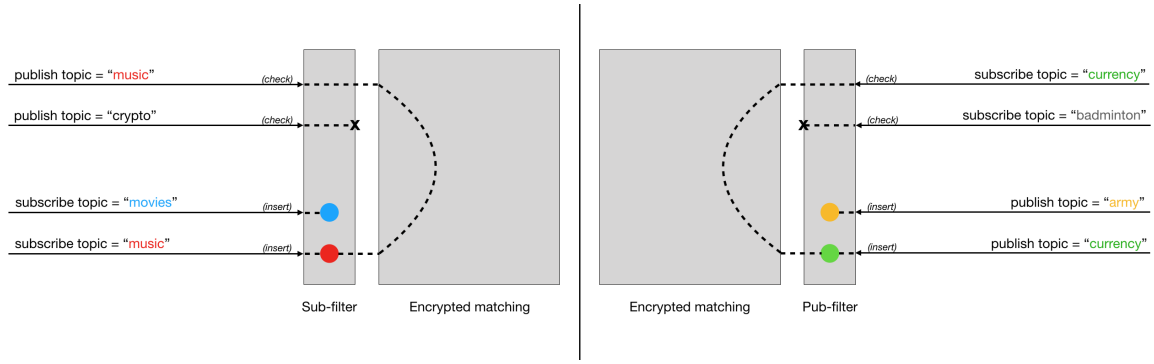
- **Matching a new publication:** On the reversed order, when the broker receives a new publication, it firstly inserts the topic of this publication into the pub-filter (figure 4, right). Then, it searches in the sub-filter to find that if this topic is present or not. If yes, it means there maybe a subscription which has the same topic at that time and the broker will execute the encrypted matching operation (figure 4c, left). Otherwise, it means there is definitely no subscription which




(a) Subscribe topics "movies" and "music" and these topics are inserted to sub-filter (left). Publish messages on topics "army" and "currency" and these topics are inserted to pub-filter (right).




(b) Publish message on topic "crypto" and encrypted matching operation is ignored because this topic definitely does not exist in sub-filter after checking (left). Subscribe topic "badminton" and encrypted matching operation is ignored because this topic definitely does not exist in pub-filter after checking (right).




(c) Publish message on topic "music" and encrypted matching operation is executed because this topic may exist in sub-filter after checking (left). Subscribe topic "currency" and encrypted matching operation is executed because this topic may exist in pub-filter after checking (right)

Fig. 4: Operations of filters in pre-filtering. There are two filters in the system: a filter containing topics of subscriptions (sub-filter) and a filter containing topics of publications (pub-filter).

has the same topic with that publication, and hence the broker will discard the encrypted matching operation (figure 4b, left). Algorithm 2 shows the steps of this procedure.

**Algorithm 2:** Pre-filter a new publication $p$

```
Add p to pub-filter;
if p may be in sub-filter then
    for each s in list of subscriptions do
        execute encrypted matching operation (p, s);
    end
else
    discard encrypted matching operation;
end
```

***Modification in Cuckoo Filter:*** We know that a Cuckoo Filter has the ability to store multiple times of the same fingerprint. However, we can observe that this feature is not useful in our work. It is unnecessary to insert the same fingerprint more than once. Even when there are two different items that have the same fingerprint (since the size of fingerprints is quite smaller than the results of hashing), we just need to store the fingerprint once. For example, two items $x$ and $y$ exactly have the same fingerprint and the $x$'s is inserted to the filter first. Then, it is not necessary to insert the $y$'s. After that, when searching with the item $y$, we will receive the result of true. It is a false positive because the fingerprint belongs to the item $x$. Therefore, we modify the original implementation of Cuckoo Filter to adapt this idea.

We know that there is a case called false positive in Bloom Filter as well as Cuckoo Filter. This term describes a possibility in which an item is actually not in the filter, but the searching function returns a result of true. Applying these two types of filters means we must accept that there are some false positives. In other words, encrypted matching operation is sometimes executed even when the involved publication surely does not match with any subscription and vice versa. The probability of this unexpected situation depends on the way we configure parameters of the filters.

## 5 Experiment

### 5.1 Implementation details

We use the library Charm-Crypto [1] to implement the encrypted matching scheme. More specially, we utilise the packet of Elliptic Curve [10] for the implementation of El Gamal encryption system. The security parameter is set to be 1024. We also use the hash function $H$ embedded in Charm-Crypto with the group of Elliptic Curve. The pseudoramdom-function $f$ is constructed by a HMAC function involving SHA1 as the hash function. All random function and computation are also in the group of Elliptic Curve. Bloom Filter and Cuckoo Filter from the Python library are embedded into the HBMQTT Broker. Client's side is implemented by Paho MQTT. For the Cuckoo Filter, three parameters including capacity, bucket size and fingerprint size are set to be 100, 8 and 2, respectively. In theory, this filter can contain up to $100 \times 8 = 800$ elements. For the Bloom Filter, false positive rate is 0.01 and number of maximum elements is 800.

To embed a filter into the broker, we implement four functions: two functions for adding an item to filters of publications and subscriptions, two functions for searching an item in filters of publications and subscriptions. Before the starting point of encrypted matching function, we check that if the current item is present in the corresponding filter. If yes, we jump into that costly function. If no, we ignore the matching.

### 5.2 Main results

The experiment is set up as follows. We have different numbers of subscriptions (1, 5, 10, 15, 20 - column (1) in table 2) and they are waiting for a publication to match. Then, we send to the broker a new publication with a topic name which is not overlapped with any subscription. This aims to measure the time of traversing all list of subscriptions by the broker to do comparisons in the action of matching with non-pre-filtering. And this also aims to proves the advantage of pre-filtering because the encrypted matching operation is early ignored when the topic of publication is definitely not in the sub-filter.

Table 2 shows the improvement in terms of time thanks to pre-filtering in our experiment. We consider columns (2), (3*) and (4*) to see the improvement. As we can see, when the number of subscriptions increases from 1 to 20, the time taken by encrypted matching operation also increases from 1.567ms to 57.497ms (column (2)). The larger number of subscriptions, the larger amount of time this operation takes. However, it still returns the result of unsuccessful matching. Meanwhile, the Bloom Filter needs less than 0.3ms to check and decide to ignore the encrypted matching operation (column (3*)). More impressively, the Cuckoo Filter needs only a little bit more than 0.1ms to do

| Number of subscriptions (1) | Non-pre-filtering (2) | Pre-filtering with Bloom Filter (3) | | Pre-filtering with Cuckoo Filter (4) | |
|---|---|---|---|---|---|
| | | Prefilter (*) | Matching (**) | Prefilter (*) | Matching (**) |
| 1 | 1.567 | 0.260 | 1.540 | 0.136 | 2.058 |
| 5 | 7.294 | 0.254 | 9.917 | 0.115 | 9.372 |
| 10 | 23.685 | 0.280 | 18.719 | 0.138 | 20.773 |
| 15 | 34.918 | 0.250 | 35.828 | 0.196 | 33.300 |
| 20 | 57.497 | 0.296 | 53.877 | 0.137 | 51.202 |

Table 2: Time of pre-filtering and matching when using filter and non-filter (millisecond). In this experiment, a publication tries to match with multiple subscriptions by encrypted matching. Column (2) shows the time taken by matching operations when there is no pre-filtering operation. Columns (3) and (4) show the time of pre-filtering using Bloom Filter and Cuckoo Filter, respectively, when the publication does not match with any subscription. And hence, encrypted matching operations are ignored in those cases (*). We also measure time of encrypted matching operations when pre-filtering and the publication matches with the last subscription in the list (**).

the same thing as the Bloom Filter (column (4*)). Therefore, applying pre-filtering in this such circumstance can save a considerable amount of time.

Besides, we adjust our experiment in which the publication matches with the last subscription in the list. And hence, the broker needs to traverse all list of subscriptions after pre-filtering (columns (3**) and (4**)). As we can see, the measured time in these two columns are nearly the same as the column (2) as expected, since they are the time needed to execute encrypted matching operation between the publication and all subscriptions in list.

### 5.3 Discussion

Despite we gain an improvement in time saving, this happens only with the false negative cases of the filter. In cases of false positive or true positive, there is no difference between pre-filtering and non-pre-filtering. The broker still sequentially executes the encrypted matching operation with the publication and each subscription. Furthermore, it additionally costs a small duration to pre-filter compared to the non-pre-filtering approach.

***Future work:*** A possible solution for the above drawback is to find a new encrypted matching scheme which can be computed faster than the current one. Besides, this work proposes a solution for the matching of two topic names which is quite simple. Based on this direction, we can expand the problem into the content-based publish/subscribe systems.

## 6 Conclusion

In this report, we have presented an application of pre-filtering in publish/subscribe systems using encrypted matching operation. This technique reduces number of calls to the function of encrypted matching which is very costly and hence, saves so much time. Our experiment shows that Cuckoo Filter performs better than Bloom Filter with nearly 2x faster. In summary, Cuckoo Filter is a better choice to pre-filter if we prioritise the factor of time.

## Acknowledgement

The author would like to wholeheartedly express his gratitude to Emmanuel CONCHON and Mathieu KLINGLER for their guidance during the two months of his summer internship at XLIM.

## References


1. Akinyele, J.A., Garman, C., Miers, I., Pagano, M.W., Rushanan, M., Green, M., Rubin, A.D.: Charm: a framework for rapidly prototyping cryptosystems. Journal of Cryptographic Engineering **3**(2), 111–128 (2013). https://doi.org/10.1007/s13389-013-0057-3, http://dx.doi.org/10.1007/s13389-013-0057-3
2. Appleby, A.: Murmurhash 2.0 (2008)
3. Barazzutti, R., Felber, P., Mercier, H., Onica, E., Rivière, E.: Thrifty privacy: Efficient support for privacy-preserving publish/subscribe. In: Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems. p. 225–236. DEBS '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2335484.2335509, https://doi.org/10.1145/2335484.2335509
4. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (Jul 1970). https://doi.org/10.1145/362686.362692, https://doi.org/10.1145/362686.362692
5. Choi, S., Ghinita, G., Bertino, E.: A privacy-enhancing content-based publish/subscribe system using scalar product preserving transformations (08 2010). https://doi.org/10.1007/978-3-642-15364-8$_3$2
6. Dong, C., Russello, G., Dulay, N.: Shared and searchable encrypted data for untrusted servers. In: Proceeedings of the 22nd Annual IFIP WG 11.3 Working Conference on Data and Applications Security. p. 127–143. Springer-Verlag, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70567-3$_1$0, https://doi.org/10.1007/978-3-540-70567-3_10
7. Fan, B., Andersen, D.G., Kaminsky, M., Mitzenmacher, M.D.: Cuckoo filter: Practically better than bloom. In: Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies. p. 75–88. CoNEXT '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2674005.2674994, https://doi.org/10.1145/2674005.2674994
8. Fowler, G., Noll, L.C., Vo, K.P., 3rd, D.E.E., Hansen, T.: The FNV Non-Cryptographic Hash Algorithm. Internet-Draft draft-eastlake-fnv-17, Internet Engineering Task Force (May 2019), https://datatracker.ietf.org/doc/html/draft-eastlake-fnv-17, work in Progress
9. Ion, M., Russello, G., Crispo, B.: Design and implementation of a confidentiality and access control solution for publish/subscribe systems. Computer Networks **56**(7), 2014 – 2037 (2012). https://doi.org/https://doi.org/10.1016/j.comnet.2012.02.013, http://www.sciencedirect.com/science/article/pii/S1389128612000771
10. Koblitz, N.: Elliptic curve cryptosystems. Mathematics of Computation **48**(177), 203–209 (Jan 1987)