Real-time Private Membership Test using Homomorphic Encryption

Eduardo Chielle
Center for Cyber Security
New York University Abu Dhabi
eduardo.chielle@nyu.edu

Homer Gamil
Center for Cyber Security
New York University Abu Dhabi
homer.g@nyu.edu

Michail Maniatakos
Center for Cyber Security
New York University Abu Dhabi
michail.maniatakos@nyu.edu

***Abstract*—With the ever increasing volume of private data residing on the cloud, privacy is becoming a major concern. Often times, sensitive information is leaked during a querying process between a client and an online server hosting a database; The query may leak information about the element the client is looking up, while sensitive details about the contents of its database can leak on the server side. The ability to check if an element is included in a database while maintaining both the client's and the server's privacy is known as the *Private Membership Test*. In this context, we propose a method to privately query a database with computational complexity O(1) using Bloom filters and Homomorphic Encryption. The proposed methodology also enables post-encryption insertions and deletions without requiring a new setup. Experimental results show that our proposed solution has practical setup, insertion and deletion times for databases of up to a few million entries, with constant query time less than $0.3s$, considering a false positive rate lower than 10^{-3}. We instantiate our methodology for a URL denylisting service, and demonstrate that it can provide solid security guarantees without affecting the user experience.**

I. Introduction

Databases are an essential tool for the rapidly growing number of online services utilized daily to store and access information. Querying online databases, however, has raised concerns regarding the privacy of parties interacting in this information transaction. *Private Membership Test* (PMT) is a method in which the client can privately evaluate whether a particular element is part of a third-party database. In PMT, the server does not learn about the client's query, while the client does not learn anything about the server's database except for the query result (i.e., whether the queried element is part of the database or not). It is important to note that many applications are interactive, therefore when designing PMT solutions, real-time performance capabilities have to be taken into consideration in order not to sacrifice usability for privacy.

Current PMT methods exhibit practical querying times [1]–[4], but this comes at the cost of flexibility. In other words, existing solutions with practical querying time become inefficient when it comes to updating the contents of a database. This problem stems from the fact that these methods have not been designed to support insertions and deletions, and therefore, every single update requires the re-initiation of the costly preprocessing protocol for the entire database.

URL denylisting services are motivating examples of the requirement of frequent deletions and insertions. A URL denylisting service is an add-on to an email service, which rewrites all URLs of all incoming emails to pass them through the service website first. The service will deny forwarding to the URL if it matches its malicious URL database. If the URL is not in the database, the client will be automatically redirected to the target URL. URL denylisting services are used to thwart phishing attacks, and their adoption has increased recently. A URL denylist provider maintains a database with a few millions URLs where new entries are added almost daily [5], [6], thus support for frequent insertions/deletions is essential.

Apparently, the URL denylisting service can effortlessly see and track all websites visited by the users of the service, which is a major privacy concern. At the same time, the service needs to respond to the client without noticeable delay, in order not to affect usability, as users would probably be unhappy waiting for several seconds/minutes to access a website. Therefore, there is a need for a solution that: 1) Protects the privacy of the users querying online databases, 2) Allows the database providers to insert/delete entries without having to recreate everything from scratch, and 3) Offers acceptable querying time.

To this end, in this work we combine Bloom filters [7], a common data structure used by online servers for performance improvement, with homomorphic encryption [8], a type of encryption that enables computation on encrypted data. This combination allows real-time private membership tests with a query complexity of O(1), reaching querying latency under $0.3s$. In addition, our solution provides flexibility, since it enables insertions and deletions in practical times with complexity O(n). Our proposal is suitable for databases of up to a few million entries, making it ideal for URL denylist applications. Our contributions can be summarized as follows:

- We propose a solution that enables PMT with complexity O(1) and query time under $0.3s$, and database insertions and deletions with complexity O(n);
- We instantiate our solution using the BFV homomorphic encryption scheme [8], where we apply a number of data-encoding optimizations to bring our setup protocol to practical times;
- We evaluate the proposed design in the context of a URL denylisting application.

II. Preliminaries

A. Bloom filters

A Bloom filter is a probabilistic data structure that is used for efficient Boolean searching. More specifically, a Bloom

978-3-9819263-5-4/DATE21/©2021 EDAA

TABLE I
NOTATION

Functions		Sets	
$D(\cdot)$	decryption function	β	Bloom filter
$E(\cdot)$	encryption function	Δ	database
$H(\cdot)$	hash functions	ι	indices from $H(\cdot)$
$R(\cdot)$	random permutation	θ	positions of ι in $\hat{\beta}$
$dec(\cdot)$	decrements indices	ρ	permutated indices
$inc(\cdot)$	increments indices	**Scalars**	
$set(\cdot)$	sets indices of $\hat{\beta}$	d	# of entries in Δ
$unset(\cdot)$	unsets indices of $\hat{\beta}$	h	# of hash functions
$sort(\cdot)$	sort function	m	# of elements in β
$\lvert\cdot\rvert$	size of a set	s	# of ciphertexts in $\hat{\rho}$
$\widehat{\cdot}$	encrypted element/set	e	element or entry
$\cdot[i]$	item of $\cdot$ at index i	n	polynomial degree
$\cdot[\omega]$	subset of $\cdot$ at indices ω	t	plaintext modulus

filter can be described as an m-bit array that represents all the elements e of a set Δ of size d.[1]

$$\Delta = \{e_1, e_2, ..., e_d\}$$

For the construction of a Bloom filter array β, an h number of hash functions H is required. When the array is initialized, all cells take a value of 0. The selected hash functions are utilized to modify the initialized array such that all elements of set Δ are represented. Each element of the set Δ is passed as an input to all hash functions. Each output is used as an index that sets the pinpointing location of β to 1.

$$\beta[H_i(e_1)] = 1, \forall\, i \in \mathbb{Z} : 1 \leq i \leq h$$

This process is repeated until all elements e of the set Δ are passed through every hash function.

B. Homomorphic encryption

Homomorphic encryption is a type of encryption that enables computation on encrypted data. For a given function $F_p(\cdot)$ on plaintexts, there is an equivalent function $F_c(\cdot)$ on ciphertexts. Consider the example in Eq. 1, where c_1 and c_2 denote ciphertexts and $D(\cdot)$ is the decryption function. In this case, F_c is homomorphic to F_p iff the equality is satisfied for any c_1 and c_2.

$$D\Big(F_c(c_1, c_2)\Big) = F_p\Big(D(c_1), D(c_2)\Big) \tag{1}$$

C. BFV crypto scheme

Brakerski/Fan-Vercauteren (BFV) is a fully homomorphic encryption scheme based on RLWE (Ring Learning With Errors), which is an extension of the scheme provided by Brakerski on standard LWE (Learning With Errors) [8]. The BFV scheme supports batching, which is a technique that allows packing multiple plaintexts into one ciphertext. This concept allows parallel processing of plaintexts in a Single-Instruction Multiple-Data (SIMD) fashion, significantly improving the performance of the scheme in supported applications [9].

D. Threat model

The threat model targeted in this work is the *semi-honest* model. In this model, the server will follow the established protocol as designed, but may attempt to learn more information about the client's data. In other words, our aim is to protect the client's query and ensure that the server does not learn anything about it.

[1]Table I summarizes the notation used throughout the paper.

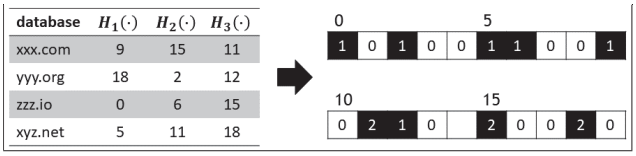

Fig. 1. **Database representation**. We hash the database using a set of hash functions and increment each positions of the Bloom filter accordingly.

III. PROPOSED PMT PROTOCOL

In our target scenario, a client wants to check if a particular element is part of a database without revealing the element to the server. At the same time, the server does not want to share the whole database with the client, and the client can only learn if a particular element is part of it.

1) Data representation: The first task is to transform the database into a Bloom filter. For that, the server applies a set of independent hash functions to each item of the database. This results into a list of indices that shall be set in the Bloom filter (Section II-A). However, unlike standard Bloom filters that set indices to either true or false whether an index is part of the database or not, we accumulate repeating values in order to consider collisions, as shown in Fig. 1. This process effectively results in a list of integers, where the value of each element signifies how many hashes resulted in that index. An accumulating Bloom filter will come in handy when deleting an element from the database.

2) Setup: Fig. 2 illustrates the setup phase. The server converts the database Δ to its Bloom filter representation β using hash functions $H(\cdot)$. Then, it sends the number of elements in the filter m to the client. The client creates a list of m unique integers in the interval $[0, m-1]$ and permutates them in a random order using function $R(\cdot)$, resulting in the list ρ. $E(\cdot)$ encrypts ρ into a list of ciphertexts $\hat{\rho}$, which is sent to the server. Finally, the server applies the data-oblivious algorithm $sort(\cdot)$ passing the plaintext filter β converted to Boolean and the encrypted list of permuted indices $\hat{\rho}$ as parameters, resulting in an encrypted Bloom filter $\hat{\beta}$ sorted in a way only known to the client.

3) Query: The client hashes the query e using a set of hash functions $H(\cdot)$ provided by the server, resulting in a list ι of up to h unique indices, where h is the number of hash functions, as one can see in Fig. 3. Based on the list of randomly permuted indices ρ, defined during setup, the client identifies the positions θ of the shuffled indices ι, which are sent to the server in plaintext. The server simply returns the ciphertexts at positions θ in the encrypted filter $\hat{\beta}$. Finally, the client decrypts the ciphertexts. If any resulting plaintext is zero, the query is not part of the database, otherwise, it most probably is, with a probability of false positives that depends on h, m, and d, where $d = |\Delta|$ [7].

4) Insertion: Fig. 4 depicts the insertion process. The server wants to insert an element e in the Bloom filter representation β of the database. For this, it first hashes the value using a set

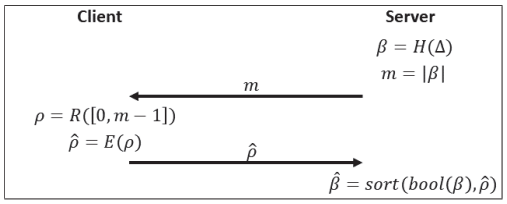

Fig. 2. **Setup**. The Bloom filter is encrypted and shuffled in a random order only known to the client.

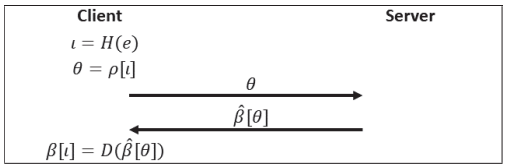

Fig. 3. **Query**. The client hashes the query e and calculates the positions θ of the indices ι using the permutations ρ defined during setup, which is sent to the server. The server simply returns ciphertexts at positions θ. Finally, the client decrypts $\hat{\beta}[\theta]$ and finds if the query is in the database or not.

of hash functions $H(\cdot)$, resulting in a list ι of h indices (not necessarily unique), where h is the number of hash functions. Function $inc(\cdot)$ increments the indices ι of filter β. At the same time, it creates a subset ι_λ of ι containing only the indices that made β become non-zero at their respective indices. Since the encrypted Bloom filter $\hat{\beta}$ is in a Boolean representation, a change from anything other than zero to one is irrelevant. This operation reduces execution time and noise consumption when inserting in the encrypted Bloom filter because fewer indices need to be set. Lastly, function $set(\cdot)$ traverses list $\hat{\rho}$ and sets the indices of $\hat{\beta}$ where ι_λ matches $\hat{\rho}$.

5) Deletion: Deletion, shown in Fig. 5, works in a similar way to insertion. First, the server generates the list of indices ι of value e using hash functions $H(\cdot)$. Then, function $dec(\cdot)$ decrements the filter β at indices ι and creates a subset ι_λ of ι. This subset contains only the indices that turned β into zero at their respective indices, since a change into zero is the only relevant change to the encrypted Bloom filter $\hat{\beta}$. Lastly, function $unset(\cdot)$ traverses list $\hat{\rho}$ and unsets the indices of $\hat{\beta}$ where ι_λ matches $\hat{\rho}$.

IV. Implementation

We optimize our implementation for the BFV scheme without bootstrapping, i.e., we treat it as a Somewhat Homomorphic Encryption (SHE) scheme. This means that our method works within a maximum computation depth, since each homomorphic operation (mainly multiplication) increases the ciphertext noise until its plaintext gets corrupted. BFV also provides batching, which lets us pack many plaintexts into a ciphertext in a Single-Instruction Multiple-Data fashion and significantly improve performance. Avoiding bootstrapping and using batching enable the real-time querying times presented in this work.

A. Data representation in the encrypted domain

When batching, each ciphertext contains n slots that can hold independent values. Each slot contains a plaintext in modulo t. Since we only need one bit to represent an index of the Bloom

Fig. 4. **Insertion**. The server hashes the value e, increments the plaintext Bloom filter β at indices ι, and set the encrypted Bloom filter $\hat{\beta}$ at the indices ι_λ where β became non-zero.

Fig. 5. **Deletion**. The server hashes the value e, decrements the plaintext Bloom filter β at indices ι, and unset the encrypted Bloom filter $\hat{\beta}$ at the indices ι_λ where β became zero.

filter, we can use the plaintext to represent several indices. Thus, a ciphertext can hold $n \cdot \lfloor \log_2 t \rfloor$ indices of the Bloom filter. As a result, the number of ciphertexts necessary to represent the Bloom filter is $|\hat{\beta}| = \lceil \frac{m}{n \lfloor \log_2 t \rfloor} \rceil$. For example, given a Bloom filter with $m = 2^{24}$ indices, the polynomial degree $n = 2^{14}$, and the plaintext modulus $t = 2^{16} + 1$, we have $|\hat{\beta}| = 2^8$ ciphertexts.

B. Element-wise shuffle setup

In our initial configuration, we treat every element of the Bloom filter as an independent index for the setup protocol. The setup phase is by far the slowest process due to the sort algorithm, which must be data-oblivious since it works on encrypted data. The client generates a random permutation of unique numbers in the interval $[0, m-1]$. These m numbers must be encrypted before sending to the server. Since the sort algorithm requires comparison, the numbers must be broken down into their bit constituents to allow homomorphic gate operations to be performed, creating a total of $m \cdot \lceil \log_2 m \rceil$ bits. Each ciphertext can hold n plaintexts, thus, the total number of ciphertext for the setup phase is $s = \lceil \frac{m}{n} \rceil \cdot \lceil \log_2 m \rceil$. For example, given $m = 2^{24}$ and $n = 2^{14}$, $s \approx 2^{14.58}$ ciphertexts.

The sort algorithm is depicted in Alg. 1. In homomorphic encryption, the multiplication is the slowest and noisiest operation. In the algorithm, only the comparison operation at line 7 uses homomorphic multiplication. An efficient implementation of the equality between a ciphertext and a plaintext requires $\lceil \log_2 m \rceil - 1$ homomorphic multiplications and has a multiplicative depth equal to $\lceil \log_2 \lceil \log_2 m \rceil \rceil$. However, the equality operation is only required for the indices of the Bloom filter set to true. In the worst case (in case the whole Bloom filter is set to true), the number of homomorphic multiplications is given by $\#muls = (\lceil \log_2 m \rceil - 1) \cdot m \cdot \lceil \frac{m}{n} \rceil$. Considering our example with $m = 2^{24}$ and $n = 2^{14}$, the number of homormorphic multiplications would be $\#muls \approx 2^{38.52}$. In reality, the Bloom filter occupation is much lower and the actual numbers of multiplications is given by $\#muls \leq (\lceil \log_2 m \rceil - 1) \cdot h \cdot d$, where $h = |H|$ and $d = |\Delta|$. For $d = 2^{20}$ and a false positive rate $f \leq 10^{-3}$, we need $h = 6$. Therefore, $\#muls \leq 2^{27.17}$.

It is worth noting that this algorithm creates an encrypted Bloom filter with $\lceil \frac{m}{n} \rceil$ ciphertexts. Since these ciphertexts are either the encryption of zero or one, it is possible, although not

Algorithm 1 Element-wise shuffle setup: sort algorithm

```
1: function SORT(β, ρ̂)
2:     vector<Ciphertext> β̂
3:     for ( ĉ : ρ̂ ) do
4:         vector<Ciphertext> τ̂
5:         for ( i : β ) do
6:             if ( i ) then
7:                 τ̂.append( ĉ == i )
8:             end if
9:         end for
10:        β̂.append( add_many(τ̂) )
11:    end for
12:    return β̂
13: end function
```

required to use each bit of the plaintext in each of the ciphertext slots for an index. In this case, the resulting encrypted Bloom filter $\hat{\beta}$ is composed of $\lceil \frac{m}{n \cdot \lceil \log_2 t \rceil} \rceil$ ciphertexts.

When the client wants to access a specific index, they check in which position that index is stored, and then request the position from the server. Once the request has been completed, the client can decrypt and check the index at that position.

C. Group-wise shuffle setup

In case the homomorphic encryption scheme employed uses batching, i.e. can pack multiple plaintexts in one ciphertext, we can use this property to improve the setup time of the protocol by grouping indices and shuffling only the groups.

When the client needs a position of the encrypted Bloom filter, they are actually receiving the ciphertext that contains it. Therefore, all indices in that ciphertext are referenced by requesting the same ciphertext. A possible optimization is to only permutate the ciphertext indices. In this case, instead of permutating m unique numbers, the client has to permutate only $\lceil \frac{m}{n \cdot \lfloor \log_2 t \rfloor} \rceil$ unique numbers, leading to $s = \lceil \log_2 m \rceil \cdot \lceil \frac{m}{n \cdot \lfloor \log_2 t \rfloor} \rceil$ ciphertexts due to the bit expansion necessary to compare encrypted data. Thus, the indices are random among ciphertexts, but in-order inside a ciphertext. For example, for $m = 2^{24}$, $n = 2^{14}$, and $t = 2^{16} + 1$, we have $s \approx 2^{10.58}$.

Within a ciphertext, each subsequent bit represents an index distant by s positions, as shown in Eq. 2, where $i = [0, s-1]$, $j = [0, n-1]$, $k = [0, \lfloor \log_2 t \rfloor - 1]$, $\hat{\beta}[i]$ is the ciphertext at position i of the encrypted Bloom filter, $\hat{\beta}[i][j]$ is the slot j of ciphertext $\hat{\beta}[i]$, and $\hat{\beta}[i][j][k]$ is a bit of the plaintext at slot $\hat{\beta}[i][j]$. We do that to balance the access rate of ciphertexts when $\frac{m}{n \cdot \lfloor \log_2 t \rfloor}$ is not an integer. Nevertheless, ideally, m should be a multiple of $n \cdot \lfloor \log_2 t \rfloor$.

$$\hat{\beta}[i][j][k] = \beta[\hat{\rho}[i] + (j \cdot \lfloor \log_2 t \rfloor + k) \cdot s] \tag{2}$$

The sort algorithm is shown in Alg. 2. In this case, we have a comparison between a ciphertext and a scalar, followed by a multiplication between a ciphertext and a plaintext vector, which can be efficiently encoded as a plaintext polynomial. As before, the comparison requires $\lceil \log_2 m \rceil - 1$ homomorphic multiplications and has a multiplicative depth equal to $\lceil \log_2 \lceil \log_2 m \rceil \rceil$. Meanwhile, the multiplication consists of one homomorphic multiplication with depth equal to one. That

Algorithm 2 Group-wise shuffle setup: sort algorithm

```
1: function SORT(β, ρ̂, n, ⌊log2 t⌋)
2:     s = |ρ̂|
3:     β.resize(n · s · ⌊log2 t⌋, 0)
4:     vector<vector<integer>> β_u
5:     for ( i = 0; i < s; i++ ) do
6:         vector<integer> ψ
7:         for ( j = 0; j < n; j++ ) do
8:             p = 0
9:             for ( k = 0; k < ⌊log2 t⌋; k++ ) do
10:                l = i + (j · ⌊log2 t⌋ + k) · s
11:                p = (p << 1)+ bool( β[l] )
12:            end for
13:            ψ.append(p);
14:        end for
15:        β_u.append(ψ)
16:    end for
17:    vector<Ciphertext> β̂
18:    for ( ĉ : ρ̂ ) do
19:        vector<Ciphertext> τ̂
20:        for ( i = 0; i < s; i++ ) do
21:            τ̂.append( (ĉ == i) · β_u[i] )
22:        end for
23:        β̂.append( add_many(τ̂) )
24:    end for
25:    return β̂
26: end function
```

gives us $\lceil \log_2 m \rceil$ multiplications per iteration with depth equal to $\lceil \log_2 \lceil \log_2 m \rceil \rceil + 1$, which can be slightly optimized to $\lceil \log_2 \lceil \log_2 m \rceil + 1 \rceil$ by combining both equality and multiplication operations within one arithmetic circuit. The total number of multiplications is given by $\#muls = \lceil \log_2 m \rceil \cdot \lceil \frac{m}{n \cdot \lfloor \log_2 t \rfloor} \rceil^2$. For example, for $m = 2^{24}$, $n = 2^{14}$, and $t = 2^{16} + 1$, we have $\#muls \approx 2^{16.58}$. The encrypted database is composed of $\lceil \frac{m}{n \cdot \lceil \log_2 t \rceil} \rceil$ ciphertexts.

V. Experiments

We ran all our experiments on a single thread of an Intel i7-4790 CPU @ 3.60GHz, 16 GB of RAM, running Ubuntu 16.04.3 with 64 GB of swap area. We use GCC 7.5.0, SEAL 3.3.2 [10], and E3 #8555dd7 [9]. The encryption parameters are selected to provide at least 128 bits of security [11].

As for our setup parameters, we selected $t = 2^{16} + 1$, as it is the smallest plaintext modulus that allows batching with our encryption parameters (a smaller t is desirable since it translates to a higher noise budget), and $n = 2^{14}$, where a smaller n means higher performance but lower noise budget. We defined m according to Eq. 3, where e is the Euler's number. For the element-wise shuffle (`ES`), we round m up to the next multiple of n, and for the group-wise shuffle (`GS`) we set m as the next multiple of $n \cdot \lfloor \log_2 t \rfloor$, as shown in Eq. 4; Thus, we fully utilize all slots of the ciphertexts. We set the number of hashes functions $h = 6$ as it provides the best trade-off between a false positive rate $f \leq 10^{-3}$ and performance for the value of m selected.

$$m = \left\lceil -d \cdot \frac{h}{\log(1 - e^{\log(f)/h})} \right\rceil \tag{3}$$

$$m_{ES} = n \cdot \left\lceil \frac{m}{n} \right\rceil \quad m_{GS} = n \cdot \lfloor \log_2 t \rfloor \cdot \left\lceil \frac{m}{n \cdot \lfloor \log_2 t \rfloor} \right\rceil \tag{4}$$

Fig. 6 shows the execution time for the setup, query, and insertion/deletion algorithms for the proposed methods, ES and GS, with a false positive rate $f \leq 10^{-3}$.

Setup: We can see that the element-wise shuffle setup time grows with the increase of the database size d, as discussed in Section IV-B, and it becomes prohibitive for d in the thousands. Regarding the group-wise shuffle setup, its execution time is proportional to m, which stays constant until $d = 2^{14}$ and $m \approx 2^{18}$ since the number of indices per ciphertext is $n \cdot \lfloor \log_2 t \rfloor$. Then, the execution time grows linearly to m, making this method still suitable for database sizes in the millions.

Query: We tested the query processing time with 100 queries, where half of the queries were part of the database, and report the average time per query. For each query, the client sends h plaintexts with the positions of $\hat{\beta}$ they want to access. The server returns the requested ciphertexts, then the client decrypts them and checks if the query is part of the database. The only computationally heavy operation here is decryption, which the client can preempt once a zero has been found. Since the query does not depend on the database size, its execution time stays constant under $0.3s$. With regards to communication, the query protocol requires the server to send h ciphertexts to the client. In $\hat{\beta}$, each ciphertext has a size of 2 MB for $n = 2^{14}$. These large ciphertexts are required to provide sufficient noise budget for homomorphic computation during setup, insertions, and deletions. Since there are no homomorphic operations when querying, we can use a technique called *modulus switch* to reduce the size (and consequently noise budget) of ciphertexts. This results in a ciphertext size of 256 KB for our encryption parameters. Therefore, by employing modulus switching the server has to send $h \cdot 256$ KB to the client per query.

Insertion/deletion: The insertion and deletion time presented is average since not all indices ι resulting from H need to be set/unset in $\hat{\beta}$, only ι_λ, but for the general case, where f is small, $|\iota| \approx |\iota_\lambda|$, which means that the average insertion/deletion time is close to the worst case, i.e., when the server has to set/unset ι indices. In Fig. 6, one can notice that the insertion/deletion time stays constant for $d \leq n$ for the element-wise shuffle, and $m \leq n \cdot \lfloor \log_2 t \rfloor$ for the group-wise shuffle method. As the number of ciphertexts in the Bloom filter increases, so does the insertion/deletion time, proportionally.

All proposed algorithms are highly parallelizable; therefore, their performance can be vastly improved by employing threads, since servers tend to have many cores.

VI. Use case: URL denylist

We apply our solution to a URL denylist service, which is actually a service purchased by our institution and motivated this work. In this scenario, our institution forces the users to use the service of a 3rd-party (the server) to verify if the links that the employees (the clients) click on their emails are malicious. The institution mail servers replace all links in emails with links to the service, as shown in Fig. 7. Once a user clicks on the link, the server checks if the link is part of their database, which consists of a list with a few million malicious websites [5]. If it belongs to the database, the server blocks the access to the link and alerts the user about the malicious website; otherwise, the server redirects the user to the desired website.

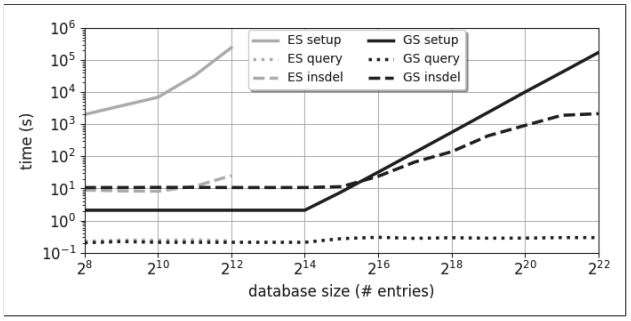

Fig. 6. **Performance evaluation**. Execution time for the element-wise shuffle (ES) and group-wise shuffle (GS) methods during setup, query, and insertion/deletion for different database sizes considering a false positive rate $f \leq 10^{-3}$.

Fig. 7. **URL denylist**. Link to google.com is replaced by a link to a 3rd-party that checks if the link is in the denylist.

Using our proposed method, the client and server perform the setup protocol, which is a one-time cost that takes a few hours. We use the database from [5] to demonstrate our end-to-end service. The database has 3.4M entries. When a user wants to access a particular link, the client performs a private query in under $0.3s$, thus adding negligible overhead to access time.

Updates to the database in this type of application are frequent [6]; therefore, the ability to insert and delete entries is paramount. This feature is a notable difference between our proposal and related work, which require re-setup for every database update, making them infeasible for such applications.

VII. Discussion

Limitations: Insertions and deletions increase the noise of ciphertexts. In order to support an infinite number of insertions or deletions, bootstrapping is necessary. For a predefined number of insertions/deletions, the server has to select encryption parameters that provide extra noise budget. When this number needs to be exceeded, the server needs to re-encrypt the Bloom filter in order to eliminate the noise of the ciphertexts.

Applicability to PIR: The proposed method also works for the Private Information Retrieval (PIR) problem. In PIR, the client informs the index of the element of the database they want to access, and the server returns the value at that index [12]. This case is similar to the one we presented, but instead of returning a flag (true or false), it returns a value (the content). If the value is greater than 1-bit, our method incurs an N times slowdown, where N is the number of bits allocated for each value.

VIII. Related work

Several works utilizing Bloom filters have been proposed in the literature for Private Membership Test. Nojima et al. [1] introduce two cryptographically secure Bloom filter protocols based on blind signatures and oblivious pseudo-random functions, where the Bloom filter encrypted by the server is sent to the client. Their proposal utilizes RSA-FDH blind signatures [13] and achieves query time complexity O(1). In [2], Meskanen et al. expand on [1] with a third protocol using the Goldwasser-Micali encryption scheme [14] as a building block. The authors report querying times of 0.12, 5.4, and 0.05 seconds per record for the respective protocols. Although two out of the three methods can perform in real-time, they do not support insertions and deletions. That is because the Bloom filter encrypted with the server's key is sent to the client during setup, and the client queries on this local Bloom filter with the server's help. In addition, contrary to our work where the server has an integer version of the Bloom filter to allow deletions, they use the classical Boolean Bloom filter, which does not allow deletions by design. Therefore, every database update requires repeating the costly setup protocol.

Ramezanian et al. [3] present solutions based on Cuckoo [15] and Bloom filters using SHA-1 for hashing, and the Paillier cryptosystem [16] for security. The server divides the filter into subsets that are accessed in plaintext using the query bits revealed by the client. In each query, the client also learns a significant portion of the filter. The efficiency of the protocol depends on the amount of information each party is willing to reveal. This solution provides setup and query times in the order of seconds. The fast setup makes updating the database practical, but the tens of seconds required for querying prohibit this method from performing in real-time.

Tamrakar et al. [4] propose a carousel approach for Private Membership Test. Their scheme utilizes the Cuckoo filter to represent the dictionary, which is circulated through a trusted hardware on a cloud server. The proposed method cycles the dictionary through a trusted application in a way that improves efficiency when processing batches of queries. The authors realize the design using two trusted execution environments, those of ARM TrustZone and Intel SGX. The proposed method achieves low query response times while maintaining high query arrival rates, however the solution is based on a hardware root-of-trust, which has suffered from several vulnerabilities such as SGXSpectre [17] and is also vulnerable to hardware trojans [18].

IX. Concluding remarks

In this work, we propose a protocol for *Private Membership Test* using Bloom filters and homomorphic encryption considering semi-honest parties. Our method has query complexity O(1) and insertion/deletion complexity O(n). We implement and optimize our protocol for the BFV encryption scheme and its batching capabilities, allowing queries to perform under $0.3s$, while having practical insertion, deletion, and setup times for databases with up to a few million entries. The proposal is instantiated in the context of URL denylisting service, where a client wants to check with the service provider if a particular URL is malicious. URL denylisting requires frequent updates to the database, and contrary to related work, our solution does it in practical time without the need for re-initiating the protocol, while still maintaning real-time query times.

Resources

The source code of our work is publicly available at https://github.com/momalab/pmt.

References

[1] R. Nojima and Y. Kadobayashi, "Cryptographically secure bloom-filters." *Trans. Data Priv.*, vol. 2, no. 2, pp. 131–139, 2009.

[2] T. Meskanen, J. Liu, S. Ramezanian, and V. Niemi, "Private membership test for bloom filters," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1. IEEE, 2015, pp. 515–522.

[3] S. Ramezanian, T. Meskanen, M. Naderpour, V. Junnila, and V. Niemi, "Private membership test protocol with low communication complexity," *Digital Communications and Networks*, pp. 31–45, 2019.

[4] S. Tamrakar, J. Liu, A. Paverd, J.-E. Ekberg, B. Pinkas, and N. Asokan, "The circle game: Scalable private membership test using trusted hardware," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, vol. abs/1606.01655, 2017, p. 31–44.

[5] M. Studio, "Blacklist domains for squid-cache," https://github.com/maravento/blackweb, 2020, commit ab63933.

[6] notracking, "Automatically updated, moderated and optimized lists for blocking ads, trackers, malware and other garbage," https://github.com/notracking/hosts-blocklists, 2020, commit a7bbcc7.

[7] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, p. 422–426, Jul. 1970. [Online]. Available: https://doi.org/10.1145/362686.362692

[8] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption." *IACR Cryptol. ePrint Arch.*, vol. 2012, p. 144, 2012.

[9] E. Chielle, O. Mazonka, H. Gamil, S. Kastratovic, N. G. Tsoutsos, and M. Maniatakos, "E3: A framework for compiling c++ programs with encrypted operands," Cryptology ePrint Archive, Report 2018/1013, 2018, https://eprint.iacr.org/2018/1013.

[10] "Microsoft SEAL (release 3.3.2)," https://github.com/Microsoft/SEAL, 2019, microsoft Research, Redmond, WA.

[11] M. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. Lauter, S. Lokam, D. Micciancio, D. Moody, T. Morrison, A. Sahai, and V. Vaikuntanathan, "Homomorphic encryption security standard," HomomorphicEncryption.org, Toronto, Canada, Tech. Rep., November 2018.

[12] O. Mazonka, N. G. Tsoutsos, and M. Maniatakos, "Cryptoleq: A heterogeneous abstract machine for encrypted and unencrypted computation," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 9, pp. 2123–2138, 2016.

[13] M. Bellare and P. Rogaway, "Random oracles are practical: A paradigm for designing efficient protocols," in *Proceedings of the 1st ACM Conference on Computer and Communications Security*, ser. CCS '93. New York, NY, USA: Association for Computing Machinery, 1993, p. 62–73. [Online]. Available: https://doi.org/10.1145/168588.168596

[14] S. Goldwasser and S. Micali, "Probabilistic encryption & how to play mental poker keeping secret all partial information," in *ACM Symposium on Theory of Computing*, 1982, p. 365–377.

[15] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122 – 144, 2004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0196677403001925

[16] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Advances in cryptology–EUROCRYPT'99*. Springer, 1999, pp. 223–238.

[17] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 142–157.

[18] N. G. Tsoutsos and M. Maniatakos, "Fabrication attacks: Zero-overhead malicious modifications enabling modern microprocessor privilege escalation," *IEEE Transactions on Emerging Topics in Computing*, vol. 2, no. 1, pp. 81–93, 2013.