Digital Communications and Networks 6 (2020) 321–332

Contents lists available at ScienceDirect

Digital Communications and Networks

journal homepage: www.keaipublishing.com/dcan

Private membership test protocol with low communication complexity

Sara Ramezanian [a,*], Tommi Meskanen [a], Masoud Naderpour [a], Ville Junnila [b], Valtteri Niemi [a]

[a] Department of Computer Science, University of Helsinki, P.O. Box 68 (Pietari Kalmin katu 5), 00014 Helsinki, Finland
[b] Department of Mathematics and Statistics, University of Turku, 20014 Turku, Finland

ARTICLE INFO

Keywords:
Privacy enhancing technologies
Applied cryptography
Private information retrieval
Private membership test
Homomorphic encryption

ABSTRACT

We introduce a practical method to perform private membership tests. In this method, clients are able to test whether an item is in a set controlled by the server without revealing their query item to the server. After executing the queries, the content of the server's set remains secret. One use case for a private membership test is to check whether a file contains any malware by checking its signature against a database of malware samples in a privacy-preserving way. We apply the Bloom filter and the Cuckoo filter in the membership test procedure. In order to achieve privacy properties, we present a novel protocol based on some homomorphic encryption schemes. In our protocol, we rearrange the data in the set into $\mathcal{N}$-dimensional hypercubes. We have implemented our method in a realistic scenario where a client of an anti-malware company wants to privately check whether a hash value of a given file is in the malware database of the company. The evaluation shows that our method is feasible for real-world applications. We also have tested the performance of our protocol for databases of different sizes and data structures with different dimensions: 2-dimensional, 3-dimensional, and 4-dimensional hypercubes. We present formulas to estimate the cost of computation and communication in our protocol.

1. Introduction

Utilizing publicly accessible databases to access information is an essential part of everyday life. Usually, Internet users query through databases by clearly stating their search terms. However, this makes it possible for the database holders to gain personal information about their users [1,2], which can be explored further, e.g., for advertisement purposes. Moreover, these queries may leak sensitive and private information about the users, such as their political views, ethnicity, etc. [3–5]

Membership test is a query with an outcome of *True* or *False*, determining whether an item is in a given set or not. The procedure includes two parties: a server that possesses a set of values, and a client that wants to query this set for a certain value.

Private Membership Test (PMT) protocols empower users to perform membership tests without revealing their search values to the database holders. To illustrate the importance of PMT, consider the following real-life scenario: A server holds a database consisting of malware hash values. A client wants to check whether a certain file is clean or not. The contents of the file may be privacy sensitive. In that case, giving away even the hash of the file is not acceptable. Although it is not possible to derive contents of arbitrary files from their hash values, the server may make **an educated guess** about the contents of the file and may be able to check whether the guess is correct.

A trivial approach to this problem is delivering a copy of the server's database to the client. However, in our case, due to the high bandwidth usage, this solution is infeasible in practice.

Our contributions in this paper are as follows:

- We present a novel PMT protocol with lower communication complexity compared with protocols known before. Our protocol is built on top of the protocol of [6].
- We analyse both theoretically and experimentally the choice of parameters for the new protocol. Moreover, we evaluate the performance of our protocol to achieve minimal communication complexity.
- We show that our PMT protocol is a solution that could be used in real-world cases, and it is significantly more feasible than the earlier solutions.
- We compare the performance of our protocol with the previous state of the art, from points of view of privacy, time complexity, and communication complexity.

* Corresponding author.
E-mail addresses: sara.ramezanian@helsinki.fi (S. Ramezanian), tommi.meskanen@helsinki.fi (T. Meskanen), masoud.naderpour@helsinki.fi (M. Naderpour), ville.junnila@utu.fi (V. Junnila), valtteri.niemi@helsinki.fi (V. Niemi).

https://doi.org/10.1016/j.dcan.2019.05.002
Received 26 October 2018; Received in revised form 8 January 2019; Accepted 7 May 2019
Available online 13 May 2019

After executing our PMT protocol, the server does not have enough information to make a guess about the search item of the client. Moreover, our PMT protocol keeps the contents of the server's database secret. The rest of the paper is constructed as follows: the required preliminaries are presented in Section 2. We state the problem in Section 3. Then, in Section 4, we explain the related work. Next, we present our protocol to solve the problem of PMT in Section 5. The complexity analysis of the protocol is presented in Section 6. Then, in Section 7, we show how to implement the protocol and **compare its performances based on a real-life scenario.** In Section 8, we present a security and privacy analysis of the protocol. Finally, we conclude the paper in Section 9.

2. Preliminaries

In this section, we provide the necessary background of the techniques that our protocol is based on.

2.1. Bloom filter

A Bloom filter is a probabilistic data structure that is commonly used to store big databases [7]. If the Bloom filter represents a set X, then a query for an item x through this filter shows whether x belongs to X or not. More specifically, a Bloom filter is an array of m bits that are all initially set to 0.

Each Bloom filter has l independent hash functions $H_i(x)$, where $i = 1,\ldots,l$. To insert an element of set X to the Bloom filter, we should feed the element to every hash function separately. The output of each hash function is assumed to be an index from the set $\{1,2,\ldots,m\}$ and therefore, this step will result in l positions of the filter (some of which may be equal to each other). We then set bits in those l positions to 1. This means that each element of X maps to l random-looking positions in the filter. To query an element from the Bloom filter, one should feed that element to the hash functions $H_i(x)$ and check the corresponding l positions of the filter. If the values are equal to one, then the query result is positive. Otherwise, we know that the element does not belong to the set.

A query from a Bloom filter can never result in a false negative. However, a query from the Bloom filter may cause a false positive. This means that an element does not belong to set X, but the hash values of the element point to indices in the array such that each index is set to one because of some other elements. If the number of elements in set X is n, then the false positive probability is known to be

$$\varepsilon = \left(1 - e^{-ln/m}\right)^{l}. \tag{1}$$

A critical property of the filter is *the number of bits per item*. We denote this measure of space efficiency by C, and here $C = m/n$.

It can be shown that the length of the Bloom filter is $m = -\frac{\ln\varepsilon}{(\ln 2)^2}\cdot n$, where n is the number of elements in set X, and ε is the target false positive rate.

2.2. Cuckoo filter

Fan et al. introduced a data structure more efficient than the Bloom filter, called *Cuckoo filter* in 2014 [8]. In order to store elements of set X in this filter, one should first calculate the *fingerprints* of the elements and store the fingerprints in a Cuckoo filter. The fingerprint of an element is a short bit string that has been obtained by computing the hash value of that element for a given hash function. Several different items may have the same fingerprint. Therefore, a query on the Cuckoo filter may result in a false positive.

The Cuckoo filter is an array of *buckets*, where each bucket consists of several *entries*. For efficiency reasons, there are a limited number of entries. The fingerprints will be stored in these entries. There are two candidate buckets for inserting the fingerprint of an element x into a Cuckoo filter. The addresses of the two buckets are computed as follows:

$$\begin{cases} h_1(x) = hash(x) \\ h_2(x) = h_1(x) \oplus hash(x\text{'s fingerprint}). \end{cases}$$

The hash function *hash* maps the elements to one of the buckets in the filter. The fingerprint of element x will be stored in either of the buckets that have enough space. If both buckets are full (there are no empty entries in either bucket), the Cuckoo filter chooses one of them randomly (let us call it bucket i) and displaces a fingerprint from i to make space for x. The displaced fingerprint is stored in its alternative bucket j utilizing the following formula:

$$j = i \oplus hash(\text{fingerprint}). \tag{2}$$

Fan et al. suggested repeating the procedure of relocating the displaced fingerprint for 500 times. If all the 500 buckets were full, the filter is considered too full to insert. Looking up an element in this filter is simply done by checking both alternative buckets for the element's fingerprint.

One of the advantages of Cuckoo filters over Bloom filters is their ability to delete an item. To delete a fingerprint F from a Cuckoo filter, one should find two possible buckets, check which one contains F and then remove F from the bucket.

Each Cuckoo filter has seven parameters: ε is the target false positive rate, f is the length of a fingerprint in bits, b is the number of entries per bucket, m' is the number of buckets, n is the number of items, C is the average number of bits per item, and finally, α is the *load factor*, where $\alpha \in [0,1]$ shows how full the Cuckoo filter is. Fan et al. showed that $b = 4$ and $\alpha = 95.5\%$ give the best space efficiency for $\varepsilon \in (0.00001, 0.002]$. They also computed an upper bound for the false positive rate to be: $\varepsilon \le 1 - (1 - 1/2^f)^{2b}$. The minimum fingerprint size to return the above ε is

$$f \ge \lceil \log_2(2b/\varepsilon) \rceil = \lceil \log_2(1/\varepsilon) + \log_2(2b) \rceil \tag{3}$$

bits. The value C of the Cuckoo filter has the property of

$$C \le \lceil \log_2(1/\varepsilon) + \log_2(2b) \rceil / \alpha. \tag{4}$$

In order to reduce the space complexity of a Cuckoo filter, Fan et al. used the *semi-sorting buckets* optimization in Ref. [9] and saved one bit per item. Table 1 shows the space complexities of Bloom filters and Cuckoo filters, as discussed in Ref. [8].

It is shown that the length of the Cuckoo filter is $(\lceil \log_2(1/\varepsilon) + 2 \rceil / \alpha) \cdot n$, where n is the number of elements in set X.

2.3. Additively homomorphic encryption

An encryption method that allows computations to be performed on ciphertext without first decrypting it is called a homomorphic encryption [10]. Let E_k be an encryption function with a key k, and x be a message in a set M. The scheme is additively homomorphic if the following holds:

$$E_k(a) \odot E_k(b) = E_k(a+b) \quad \text{for all} \quad a,b \in M. \tag{5}$$

Note that the operation on the left side of Equation (5) could be very different from the one on the right side. Also, note that E_k does not have to be a deterministic function, i.e., there may be several valid encryptions

Table 1
Space cost comparison of Bloom filters, when the optimal number of hash functions is used, and of Cuckoo filters, when $b = 4$.

Type of the Filter	Bits Per Item	When $\alpha = 95.5\%$ and $\varepsilon = 0.001$
Bloom filter	$1.44\log_2(1/\varepsilon)$	14.3 Bits Per Item
Cuckoo filter	$\lceil \log_2(1/\varepsilon) + 3 \rceil / \alpha$	13.5 Bits Per Item
Cuckoo filter optimized by semi-sorting buckets	$\lceil \log_2(1/\varepsilon) + 2 \rceil / \alpha$	12.5 Bits Per Item

of x with the key k.

2.4. Paillier's cryptosystem

Let p and q be two different safe primes (primes of the form $2P+1$ where P is also a prime number) of the same size, $N = pq$, and g is an element of the multiplicative group $\mathbb{Z}^*_{N^2}$ with the order that is a non-zero multiple of N (for example, we may choose $g = N+1$). Paillier and Pointcheval [11] proposed the following encryption scheme:

$$w = E_g(\mathrm{c}, d) = g^{\mathrm{c}} d^N \mod N^2, \tag{6}$$

where $\mathrm{c} \in \mathbb{Z}_N$ is a plaintext, $d \in \mathbb{Z}^*_N$ is a random number, and ciphertext $w = g^{\mathrm{c}} d^N \mathrm{mod} N^2$ is in $\mathbb{Z}^*_{N^2}$. The private key is (p, q), which is the factorization of N. The ciphertext w is in $\mathbb{Z}^*_{N^2}$ and can be decrypted in the following way:

$$\mathrm{c} = D_g(w) = \frac{L(w^\lambda \bmod N^2)}{L(g^\lambda \bmod N^2)} \mod N, \tag{7}$$

where $\lambda = lcm(p-1, q-1)$ and $L(u)$ is the integer quotient of $(u-1)/N$, where $u \in \mathbb{Z}^*_{N^2}$. This cryptosystem is additively homomorphic.

The state of the art in factorization dictates that in order to have a secure cryptosystem, the length of N should be at least 2048 bits.

2.5. Chang's cryptographic scheme

Chang [6] proposed a protocol to privately retrieve an element of a 2-hypercube as follows: A server possesses a two-dimensional database DB of the size $h \times h$. A client wants to query this database to learn the value of the item that is located on the i^*-th row and j^*-th column of DB, which is denoted as $x(i^*, j^*)$. Let us consider the identity matrix I:

$$I(t, t') = \begin{cases} 1 & \text{if } t = t' \\ 0 & \text{otherwise.} \end{cases}$$

Chang's protocol uses Paillier's cryptosystem and proceeds as follows:

1. The client $\mathscr{C}$ computes

$$\alpha_t = E_g(I(t, i^*), r_t) \text{ and } \beta_t = E_g(I(t, j^*), s_t), \tag{8}$$

where $t \in \{1, 2, \ldots, h\}$, r_t and s_t are random numbers that have been uniformly chosen from $\mathbb{Z}^*_N$. The client $\mathscr{C}$ sends α_t and β_t to the server.

2. For $i = 1, 2, \ldots, h$, the server computes

$$\sigma_i = \prod_{t=1}^{h} (\beta_t)^{x(i,t)} \mathrm{mod}\ N^2. \tag{9}$$

3. The server $\mathscr{S}$ computes $u_i, v_i \in \mathbb{Z}_N$ such that $\sigma_i = u_i N + v_i$.
4. The server $\mathscr{S}$ computes

$$u = \prod_{t=1}^{h} (\alpha_t)^{u_t} \mathrm{mod}\ N^2\ ,\ v = \prod_{t=1}^{h} (\alpha_t)^{v_t} \mathrm{mod}\ N^2 \tag{10}$$

and sends u and v to the client.

5. The client $\mathscr{C}$ retrieves the value of the wanted item as:

$$x(i^*, j^*) = D_g\big(D_g(u)N + D_g(v)\big). \tag{11}$$

3. Problem statement

We formulate the PMT problem as follows: A server $\mathscr{S}$ holds a set X with the cardinality of n. A client $\mathscr{C}$ wants to perform a membership test for a search item x in a privacy-preserving way. This query results in *positive* if $x \in X$, and *negative* otherwise. After executing this protocol, $\mathscr{S}$ cannot learn x. Also, the client $\mathscr{C}$ is prevented from learning too much about the contents of X.

We motivate the problem of PMT with a real-life scenario, where a server $\mathscr{S}$ possesses a database with a malware. The server $\mathscr{S}$ stores the hash values of the malware in a set X. A client $\mathscr{C}$ has a hash value x of a file and wants to check whether the file contains any malware or not while keeping the contents of the file private. However, if the result of the PMT shows that the file is malicious, $\mathscr{C}$ is willing to reveal x to the server to get instructions on how to handle the malicious file. Hash functions are one-way functions, and therefore, a client or any third party cannot find a malware sample from its hash value. This means, in our scenario, keeping the contents of set X private is not necessary for $\mathscr{S}$.

In order to have realistic numbers, we apply the following setting in our implementations: Set X contains 2^{21} SHA-1 values. Each of the hash values consists of 160 bits, and therefore the size of X is 40 MB. To make the protocol more memory efficient, we apply a Bloom filter or a Cuckoo filter. The Bloom filter consists of 2^{25} bits with 10 hash functions. Based on Equation (1), these numbers result in a false positive rate of 0.001. Utilizing the Bloom filter reduces the size of the database to 4 MB. The Cuckoo filter with a false positive rate of 0.001 consists of 12.5×2^{21} bits. This will reduce the size of the database to 3.2 MB.

In our scenario, client $\mathscr{C}$ wants to make sure that his/her file does not have any malware. If the outcome of the PMT protocol is positive, $\mathscr{C}$ sends the hash value of the file to the server for further action. However, in some cases, the file is clean, and the result of PMT could be false positive. Fortunately, depending on false positive probability, this happens rarely and randomly. Note that when handling suspicious files, it is better to make an error on the safe side. Therefore, a small number of false positives is a less serious issue than even a smaller number of false negatives.

4. Related work

Private Information Retrieval (PIR) is a well-known topic in cryptography that was introduced by Chor et al. in 1995 [12]. Extensive work has been done on PIR, such as [13–15]. Single-server PIR protocols involve two parties: Bob holds a database X of n records, and Alice holds an index i of a record where $1 \le i \le n$. Alice wants to retrieve the i^{th} record of X without revealing her index to Bob. Bob wants to respond to Alice's query in such a way that the communication complexity is much less than $\mathscr{O}(n)$. The problem of PMT can be solved by using PIR in the following way: database X is stored in a Bloom or Cuckoo filter. Then, the PIR is used to fetch specific items from the filter. The values of these items are sufficient to perform the PMT. Section 2.5 gives an example of the PIR protocol.

In a *Private Set Intersection* (PSI) protocol, the server and the client compute the intersection of their private sets together, i.e., they only learn their joint inputs in the end. The PMT can be considered as a special case of the PSI, where the client's set has one single element. In this case, the emptiness or non-emptiness of the intersection of sets $\mathscr{S}$ and $\mathscr{C}$ determines the result of a membership test. In a recent article by Pinkas et al. [16], a PSI protocol based on *Oblivious Transfer* [17] has been presented. Although the results of [16] are promising for general use cases of the PSI, they are nevertheless infeasible in our case scenario. This is because these PSI protocols have too high communication complexity for practical on-line malware checking. We later show that the communication complexity of our protocol is significantly lower than that of the protocol in Ref. [16].

Utilizing *Trusted Hardware* (TH) is another method to solve the problem of PMT, e.g., in Ref. [18]. However, in this paper, we do not want to assume any special properties of the used computing platform.

In Ref. [19], Meskanen et al. presented three unique protocols with three different cryptosystems to perform the PMT with Bloom filters. The

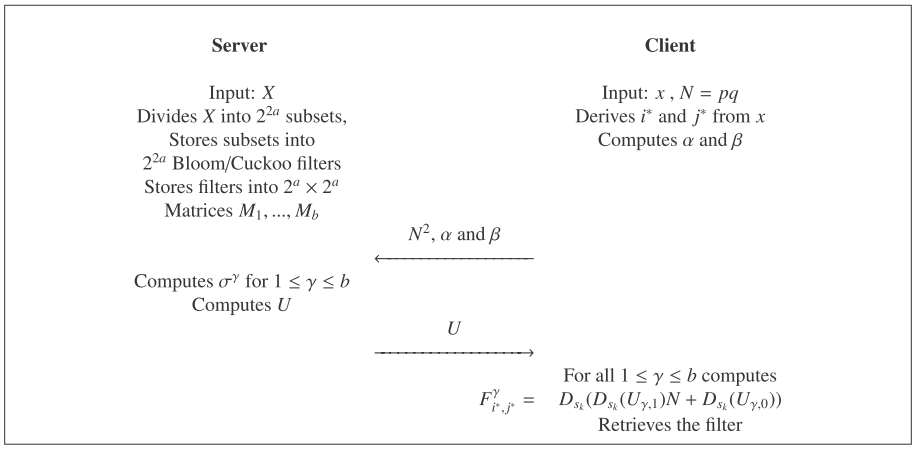

Fig. 1. An overview of our protocol on 2-dimensional data structures utilizing Paillier cryptosystem [11].

main idea in Ref. [19] is to deliver an encrypted version of the database to the client, so that the query can be done independent of the server. In the protocols, the server $\mathscr{S}$ either stores database X into B, encrypts it, and sends the encrypted Bloom filter EB to the client $\mathscr{C}$, or stores an encrypted database into B and sends the filter B to $\mathscr{C}$. Finally, $\mathscr{C}$ computes l positions of the filter corresponding to x, and with the server's help, decrypts these l bits of EB or B. The protocols have communication complexity of $\mathscr{O}\left(-\frac{n\ln\varepsilon}{(\ln 2)^2}\right)$ because of the size of the (encrypted) Bloom filter.

In this paper we present a protocol to perform PMT using homomorphic encryption. Our protocol improves the results of [19] and has a communication complexity that is significantly smaller than $\mathscr{O}\left(-\frac{n\ln\varepsilon}{(\ln 2)^2}\right)$, which makes it feasible in practice. The improvement is necessary because the server needs to respond to many clients at the same time. Our protocol has a much lower communication complexity, and reduces the bandwidth usage on the server's side significantly. Moreover, the preprocessing speed in our protocol is at least four orders of magnitude faster than the preprocessing in protocols of [19]. Table 8 in Sect. 8 also compares the performance of the protocols in Refs. [16,19] with our protocol.

5. The protocol

In this section, we present our protocol to solve the PMT problem. We assume that a server $\mathscr{S}$ has a database X that consists of n records. Each record is an s-bit string where s is an integer such that $2^s > n$. Let $X = \{h_1, h_2, \ldots, h_n\}$. We also assume that a client $\mathscr{C}$ possesses an item x and wants to perform PMT for this item through database X.

The server picks a public-key cryptosystem $(\mathbf{P}, \mathbf{C}, \mathbf{K}, E_k, D_{s_k})$ where $\mathbf{P}$, $\mathbf{C}$ and $\mathbf{K}$ are plaintext, ciphertext and key space, respectively. Moreover, $E_k : \mathbf{P} \to \mathbf{C}$ is the encryption function and $D_{s_k} : \mathbf{C} \to \mathbf{P}$ is the decryption function, where k is the public key and s_k is the secret key in space $\mathbf{K}$. For simplicity, we assume that the elements of $\mathbf{P}$ and $\mathbf{C}$ have fixed lengths $l_\mathbf{P}$ and $l_\mathbf{C}$, respectively. The lengths are known for both $\mathscr{S}$ and $\mathscr{C}$. Moreover, we assume that the elements of $\mathbf{P}$ are integers $0, 1, 2, \ldots, N-1$, when $N = |\mathbf{P}|$; and that the elements of $\mathbf{C}$ are non-negative integers. $\mathscr{S}$ computes an integer e such that $N^e < \max(\mathbf{C}) \le N^{(e+1)}$. We require the cryptosystem to be additively homomorphic, i.e., for all the plaintexts $x_1, x_2 \in \mathbf{P}$:

$$E_k(x_1) \cdot E_k(x_2) = E_k(x_1 + x_2).$$

Here, on the right side, '+" is an operation inside $\mathbf{P}$. On the left hand side, '$\cdot$' is an operation inside $\mathbf{C}$.

5.1. The protocol on 2-dimensional approach

The server $\mathscr{S}$ divides X into 2^{2a} subsets, where $2a \le s$. Each subset holds the elements of X that start with the same prefix of $2a$ bits. The server $\mathscr{S}$ picks l hash functions for a Bloom filter or one hash function for a Cuckoo filter depending on which data structure the server uses. The server $\mathscr{S}$ stores each subset into a Bloom/Cuckoo filter with a false positive rate of ε. Each Bloom filter is an array of $-\frac{n\ln\varepsilon}{(\ln 2)^2} \cdot \frac{1}{2^{2a}}$ bits and each Cuckoo filter is an array of $(\lceil \log_2(1/\varepsilon) + 2 \rceil/\alpha) \cdot \left(\frac{n}{2^{2a}}\right)$ bits. The server $\mathscr{S}$ and the client $\mathscr{C}$ agree on the filter and its hash function(s) before executing the protocol. The server constructs a $2^a \times 2^a$ matrix M and inserts the filters into M, as it is shown in Equation (12). In this matrix, $F_{i,j}$ is a filter that contains all elements of X that begin with the binary representation of $i||j$.

$$M = \begin{pmatrix} F_{0,0} & F_{0,1} & \cdots & F_{0,2^a-1} \\ F_{1,0} & F_{1,1} & \cdots & F_{1,2^a-1} \\ \vdots & \vdots & \ddots & \vdots \\ F_{2^a-1,0} & F_{2^a-1,1} & \cdots & F_{2^a-1,2^a-1} \end{pmatrix} \tag{12}$$

We call this a 2-dimensional approach, because the matrix is a two dimensional object. In the next subsection, we explain the $\mathscr{N}$-dimensional approach.

In order to be able to encrypt elements of M individually, those elements should belong to $\mathbf{P}$. Therefore, the matrix M is sliced into b matrices $M_1, \ldots, M_b$, by dividing elements of M into smaller pieces in such a way that the elements in these new matrices can be read as integers smaller than N. Thus $F_{i,j} = F^1_{i,j} \| F^2_{i,j} \| \cdots \| F^b_{i,j}$, where $F^\gamma_{i,j}$ is an element of matrix M_γ. Equation (13) shows how the matrices M_γ look like.

The server $\mathscr{S}$ considers each element $F^\gamma_{i,j}$ as a binary number. Therefore, each filter $F_{i,j}$ is interpreted as an ordered set of b integers.

$$M_1 = \begin{bmatrix} F^1_{0,0} & \cdots & F^1_{0,2^a-1} \\ \vdots & \ddots & \\ F^1_{2^a-1,0} & \cdots & F^1_{2^a-1,2^a-1} \end{bmatrix}, \ldots, M_b = \begin{bmatrix} F^b_{0,0} & \cdots & F^b_{0,2^a-1} \\ \vdots & \ddots & \\ F^b_{2^a-1,0} & \cdots & F^b_{2^a-1,2^a-1} \end{bmatrix} \tag{13}$$

The client $\mathscr{C}$ wants to perform the PMT for an item x. The client extracts a$2a$-bit prefix from x, divides the prefix into two halves and computes the decimal representation of these halves as i^* and j^*. Then, $\mathscr{C}$ computes vectors α and β as follows:

$\alpha = (E_k(I(1,i^*)),\ldots,E_k(I(2^a,i^*)))$ and

$\beta = (E_k(I(1,j^*)),\ldots,E_k(I(2^a,j^*)))$,

where $I(t,t') = 1$, if $t = t'$; and $I(t,t') = 0$ otherwise.

Now, $\mathscr{C}$ has $2^a + 2^a$ encrypted values. The client $\mathscr{C}$ sends vectors α and β and the public key k to the server.

For every matrix M_γ, $\gamma = 1,\ldots,b$, the server $\mathscr{S}$ performs the following calculations:

1. $\mathscr{S}$ computes a vector $\sigma^\gamma = (\sigma_1^\gamma,\ldots,\sigma_{2^a}^\gamma)$, where each element σ_i^γ equals to $\prod_{t=1}^{2^a}(\beta_t)^{F_{i,t}^\gamma}$.
2. For each element σ_i^γ of the vector σ^γ, the server computes $a_{i,e},\ldots,a_{i,0} \in \mathbf{P}$, such that σ_i^γ is equal to $a_{i,e}N^e + a_{i,e-1}N^{e-1} + \ldots + a_{i,0}N^0$.
3. $\mathscr{S}$ computes vector $U_\gamma = (U_{\gamma,0},\ldots,U_{\gamma,e})$, where each $U_{\gamma,i}$ equals to $\prod_{t=1}^{2^a}(\alpha_t)^{a_{t,j}}$.

The server $\mathscr{S}$ repeats steps 1–3 for every matrix M_γ and sends $U = \{U_1,\ldots,U_b\}$ to the client.

For every U_γ in U, the client computes:

$$D_{s_k}\left(D_{s_k}(U_{\gamma,e})\cdot N^e + D_{s_k}(U_{\gamma,e-1})\cdot N^{e-1} + \ldots + D_{s_k}(U_{\gamma,0})\cdot N^0\right) \quad (14)$$

and retrieves $F_{i^*j^*}^\gamma$. The client $\mathscr{C}$ calculates the binary representation of F_{i^*,j^*}^γ for $1 \le \gamma \le b$, concatenates them together, and gets a smaller Bloom/Cuckoo filter to perform the query by himself/herself. An overview of this protocol is shown in Fig. 1.

The proof of the correctness of our scheme is similar to the reasoning that is given by Chang in Ref. [6]. In our protocol, we use an additively homomorphic encryption scheme and therefore, for every $1 \le \gamma \le b$, the components σ_i^γ of the vector σ^γ are the encryptions of F_{i,j^*}^γ, where $1 \le i \le 2^a$, because all components of β_t, $1 \le t \le 2^a$, of the vector β are encryptions of zeros, except β_{j^*}, which is an encryption of one. Similarly, every $U_{\gamma,i}$ is an encryption of $a_{i^*,j}$, where $0 \le j \le e$. Therefore, when the client uses Eq. (14), first he/she computes all $D_{s_k}(U_{\gamma,i}) = a_{i^*,j}$ and gets $(a_{i^*,e})\cdot N^e + (a_{i^*,e-1})\cdot N^{e-1} + \ldots + (a_{i^*,0})\cdot N^0$, which is equal to $\sigma_{i^*}^\gamma$. Consequently, $D_{s_k}(\sigma_{i^*}^\gamma) = F_{i^*,j^*}^\gamma$.

The protocol of Chang [6] is a special case of the PIR phase of our protocol, where the cryptosystem is Paillier [11] and there is only one $h \times h$ matrix with n entries of equal length of s bits. Table 2 lists the computation complexities of our protocol and the protocol of [6]. When $n = 2^{20}$, $h = 2^{10}$, $b = 32$, $a = 4$ and utilizing Bloom filters in our protocol, it can be derived from Table 2 that a server utilizing Chang's protocol performs 114 times more modular exponentiations than a server using our protocol.

The size of the filter retrieved by the client depends on the value of a. A larger a leads to a smaller filter. On the other hand, the larger a makes the homomorphic encryption computationally expensive. We implemented this protocol for a database of more than two million elements. In order to make the protocol fast enough for industrial use cases, we assume that the client is willing to reveal a small prefix of x. Let us assume that x is a string of 160 bits and $\mathscr{C}$ reveals a 4-bit prefix of that to $\mathscr{S}$. The only information $\mathscr{S}$ obtains is that x belongs to a subset of size 1 /16 of the database. On the other hand, this makes the database smaller and reduces the amount of computation and communication significantly.

Table 2
Computation cost of the PIR phase of our protocol and the protocol of [6]. The size of the two-dimensional database in Chang's protocol is $h \times h$. We utilize b matrices of size $2^a \times 2^a$ in our protocol.

Protocol	Computation on client-side	Computation on server-side
Chang's Protocol	$2h$ encryptions	$h^2 + h$ mod. exp.
Our Protocol	$2\cdot 2^a$ encryptions	$b(2^{2a} + 2\cdot 2^a)$ mod. exp.

5.2. The protocol on $\mathscr{N}$-dimensional approach

In this subsection, we show that our protocol can be generalized in the form that utilizes more than two dimensions to build the data structure.

For the sake of simplicity, we explain the 3-dimensional case in details. And at the end of this subsection, we explain how this approach can be used with even higher dimensions.

As shown in the previous subsection, the 2-dimensional approach is built on top of the 1-dimensional case, i.e., we consider each $2^a \times 2^a$ matrix as a data structure of 2^a rows. We use the same logic to perform the PIR on a 3-dimensional data structure (a cube). Thus, each $2^a \times 2^a \times 2^a$ cube consists of 2^a matrices, and each of those matrices consists of 2^a rows. This approach is detailed in the following subsection.

The server $\mathscr{S}$ divides X into 2^{3a} subsets, such that each holds the elements of X that begin with the same prefix of $3a$ bits. If $\mathscr{S}$ uses a Bloom filter as the data structure, it picks l hash functions for this filter; otherwise, the server picks one hash function for a Cuckoo filter. Before executing the protocol, the server $\mathscr{S}$ and the client $\mathscr{C}$ make an agreement on the filter and its hash function(s). The server $\mathscr{S}$ picks a Bloom/Cuckoo filter with a false positive rate of ε and stores each subset into this filter. Each Bloom filter is an array of $-\frac{n\ln\varepsilon}{(\ln 2)^2}\cdot\frac{1}{2^{3a}}$ bits, and each Cuckoo filter is an array of $(\lceil\log_2(1/\varepsilon)+2\rceil/\alpha)\cdot\left(\frac{n}{2^{3a}}\right)$ bits.

The server $\mathscr{S}$ constructs a 3-dimensional hypercube $\mathscr{K}$ and inserts the filters into $\mathscr{K}$. In this 3-dimensional hypercube, each element $F_{i,j,k}$ is a filter that contains all elements of X that begin with the binary representation of $i||j||k$.

The server $\mathscr{S}$ slices the 3-dimensional hypercube $\mathscr{K}$ into b 3-dimensional hypercubes $\mathscr{K}_1,\ldots,\mathscr{K}_b$ by dividing elements of $\mathscr{K}$ into smaller pieces in such a way that the elements in these new 3-dimensional hypercubes can be read as integers smaller than N. Thus, $F_{i,j,k} = F_{i,j,k}^1\left|\right|F_{i,j,k}^2\left|\right|\cdots\left|\right|F_{i,j,k}^b$, where $F_{i,j,k}^\gamma$ is an element of 3-dimensional hypercube $\mathscr{K}_\gamma$.

The server $\mathscr{S}$ considers each element $F_{i,j,k}^\gamma$ as a binary number. Therefore, each filter $F_{i,j,k}$ is interpreted as an ordered set of b integers.

The client $\mathscr{C}$ wants to privately search the server's database for an item x. The client extracts a $3a$-bit prefix from x, divides the prefix into 3 parts, and interprets these as integers i^*, j^*, k^*. Then, $\mathscr{C}$ computes vectors α, β, η as follows:

$\alpha = (E_k(I(1,i^*)),\ldots,E_k(I(2^a,i^*)))$,

$\beta = (E_k(I(1,j^*)),\ldots,E_k(I(2^a,j^*)))$,

$\eta = (E_k(I(1,k^*)),\ldots,E_k(I(2^a,k^*)))$,

where $I(t,t') = 1$, if $t = t'$; and $I(t,t') = 0$ otherwise.

Now, the client $\mathscr{C}$ has $2^a + 2^a + 2^a$ encrypted values. The client $\mathscr{C}$ sends vectors α, β and η and the public key k to the server.

The server $\mathscr{S}$ computes the PIR on the 3-dimensional hypercube by considering the cube as 2^a matrices as shown below:

1. The server $\mathscr{S}$ first computes a matrix σ^γ for each cube $\mathscr{K}_\gamma$, such that the elements of the matrix σ^γ are $\sigma_{i,j}^\gamma = \prod_{t=1}^{2^a}(\eta_t)^{F_{i,j,t}^\gamma}$.
2. For each element $\sigma_{i,j}^\gamma$ of σ^γ, the server computes $a_{i,j,e},\ldots,a_{i,j,0} \in \mathbf{P}$ such that $\sigma_{i,j}^\gamma = a_{i,j,e}N^e + a_{i,j,e-1}N^{e-1} + \ldots + a_{i,j,0}N^0$.
3. $\mathscr{S}$ computes vectors $\tau_i^\gamma = (\tau_{i,0}^\gamma,\ldots,\tau_{i,e}^\gamma)$, where $\tau_{i,j}^\gamma = \prod_{t=1}^{2^a}(\beta_t)^{a_{i,t,j}}$ and $0 \le j \le e$.
4. For each element $\tau_{i,j}^\gamma$, the server computes $a'_{i,j,e},\ldots,a'_{i,j,0} \in \mathbf{P}$ such that $\tau_{i,j}^\gamma = a'_{i,j,e}N^e + a'_{i,j,e-1}N^{e-1} + \ldots + a'_{i,j,0}N^0$
5. $\mathscr{S}$ computes vector $U^\gamma = (U_{0,0}^\gamma,\ldots,U_{e,e}^\gamma)$, where each component of vector U^γ is calculated as $U_{i,j}^\gamma = \prod_{t=1}^{2^a}(\alpha_t)^{a'_{t,i,j}}$, where $0 \le i,j \le e$ and $1 \le \gamma \le b$.

The server $\mathscr{S}$ repeats steps 1–5 for every cube $\mathscr{K}_\gamma$ and sends $U = \{U^1, \ldots, U^b\}$ to the client.

For every U^γ in U, the client computes:

$$\begin{aligned} &D_{s_k}\left(D_{s_k}\left(D_{s_k}\left(U^\gamma_{e,e}\right)\cdot N^e + D_{s_k}\left(U^\gamma_{e-1,e}\right)\cdot N^{e-1} + \ldots + \right.\right.\\ &D_{s_k}\left(U^\gamma_{0,e}\right)\cdot N^0\Big)\cdot N^e + \ldots + D_{s_k}\left(D_{s_k}\left(U^\gamma_{e,e}\right)\cdot N^e + \right.\\ &\left.\left.D_{s_k}\left(U^\gamma_{e-1,0}\right)\cdot N^{e-1} + \ldots + D_{s_k}\left(U^\gamma_{0,0}\right)\cdot N^0\right)\cdot N^0\right) = F^\gamma_{i^*,j^*,k^*} \end{aligned} \tag{15}$$

The client $\mathscr{C}$ calculates the binary representation of $F^\gamma_{i^*,j^*,k^*}$ for $1 \le \gamma \le b$, concatenates them together, and gets a smaller Bloom/Cuckoo filter to perform the query on by himself/herself.

Now, we use the same logic to show how our protocol can be formalized using an $\mathscr{N}$- dimensional data structure.

The server $\mathscr{S}$ performs PIR on the $\mathscr{N}$-dimensional hypercube in a recursive manner, such that each $\mathscr{N}$-dimensional $2^{\mathscr{N}a}$ hypercube consists of 2^a ($\mathscr{N}$-1)-dimensional hypercubes, each of those ($\mathscr{N}-1$)-dimensional hypercubes consists of 2^a ($\mathscr{N}$-2)-dimensional hypercubes, etc. Therefore, each $\mathscr{N}$-dimensional $2^{\mathscr{N}a}$ hypercube consists of $2^{\mathscr{N}a-2}$ matrices.

The communication complexity of our protocol on an $\mathscr{N}$-dimensional database is as follows:

- The client $\mathscr{C}$ sends the encrypted values $\alpha_1, \alpha_2, \ldots, \alpha_{\mathscr{N}}$ of the total size $\mathscr{N}\cdot 2^a \cdot l_C$ to the server.
- The server $\mathscr{S}$ generates $(e+1)^{\mathscr{N}-1}$ encrypted values of size l_C per data structure and there are b smaller data structures. Therefore, $\mathscr{S}$ sends $b\cdot l_C\cdot(e+1)^{\mathscr{N}-1}$ bits of information to the client.

Section 7 presents the implementations of our protocol on 2-dimensional, 3-dimensional and 4-dimensional approaches.

6. Complexity analysis

In this section, we analyse the computation and communication complexities of our protocol. In our computations, we first assume that b, which is the number of hypercubes, is equal to 1. This is done because the same amount of computation and communication happen for each hypercube. Then, we can simply multiply the results with b to get the formulas in the general case.

In our analysis, we use Paillier cryptosystem in the PIR phase of our protocol. This means that $e = 1$ and cryptotexts are twice the size of the plaintexts. We assume that the size of the Bloom/Cuckoo filter is 2^c and it is inserted into an $\mathscr{N}$-dimensional data structure. If the size of the public key N in Paillier is 2^{11} (as mentioned earlier, this is an appropriate size according to the current state of art), then the capacity of the data structure in $\mathscr{N}$-dimensional approach is $2^{11+a\mathscr{N}}$. In the following analysis, we find the optimal choices of $\mathscr{N}$ and a in order to minimize the communication complexity and the computation complexity.

In order to optimize both computation and communication complexities, the parameters a and $\mathscr{N}$ should be chosen in such a way that there is no emptiness in the data structure. This means $2^{11+a\mathscr{N}} \le 2^c$, and therefore $11 + a\mathscr{N} \le c$. Also $11 + (a-1)\mathscr{N} < c$ and $11 + a(\mathscr{N}-1) < c$, because otherwise we could fit the filter into a data structure with smaller dimensions.

The value of parameter b is 1, if $c \le 11 + a\mathscr{N}$; and $b = 2^{c-(11+a\mathscr{N})}$ otherwise.

6.1. Computation complexity

In order to calculate the computation costs of our protocol, it is enough if we only count the number of modular exponentiations because their cost is dominant in our computations.

We claim that the number of modular exponentiations on the server side can be obtained by the following formula:

$$\begin{aligned} C^s_{comp} = b\Big(&2^{\mathscr{N}a} + 2\cdot 2^{(\mathscr{N}-1)a} + 2^2\cdot 2^{(\mathscr{N}-2)a} + \\ &2^3\cdot 2^{(\mathscr{N}-3)a} + \ldots + 2^{(\mathscr{N}-1)}\cdot 2^a\Big) \end{aligned} \tag{16}$$

Let us first assume that $b = 1$. For that case, we next prove the above equation by induction.

Base case: We show that Equation (16) holds for $\mathscr{N} = 2$. Assuming $\mathscr{N} = 2$, the server needs to perform 2^{2a} modular exponentiations to compute vector σ^1, then 2^a modular exponentiations to compute $U_{1,0}$, and 2^a modular exponentiations to compute $U_{1,1}$ (see section 5.1). Therefore, the server needs to perform $2^{2a} + 2\cdot 2^a$ computations, and thus we showed that the base case holds.

Inductive step: we show that if Equation (16) holds for $\mathscr{N} = k$, then it holds for $\mathscr{N} = k+1$. We assume that the server needs to perform

$$2^{ka} + 2\cdot 2^{(k-1)a} + 2^2\cdot 2^{(k-2)a} + 2^3\cdot 2^{(k-3)a} + \ldots + 2^{(k-1)}\cdot 2^a$$

computations to compute one k-dimensional hypercube. As we explained in section 5.2, each $(k+1)$-dimensional hypercube consists of 2^a k-dimensional hypercubes. For each of these 2^a hypercubes, the server needs to perform

$$2^a \times \left(2^{ka} + 2\cdot 2^{(k-1)a} + 2^2\cdot 2^{(k-2)a} + 2^3\cdot 2^{(k-3)a} + \ldots + 2^{(k-1)}\cdot 2^a\right)$$

computations.

The server gets 2^{k-1} encrypted values from recursively processing the 2^a k-dimensional hypercubes, each of the size of the modulus N^2 (and therefore, each encrypted value has 2^{12} bits). The server computes $U_{1,0}$ and $U_{1,1}$ for each encrypted value and therefore obtains $2\cdot 2^{k-1}$ values. Then, the server performs 2^a computations for each of these values, as explained in step 5 of section 5.2. Therefore, the server performs

$$2^a\left(2^{ka} + 2\cdot 2^{(k-1)a} + 2^2\cdot 2^{(k-2)a} + \ldots + 2^{(k-1)}\cdot 2^a\right) + 2\cdot 2^{k-1}\cdot 2^a$$

modular exponentiations to compute a $(k+1)$-dimensional hypercube. So, the Equation (16) holds in the case $b = 1$. The Equation (16) also holds for an arbitrary value of b, because in that case the computation contains b different parts, each of which has the cost equal to the cost of the case of $b = 1$. Therefore, the Equation (16) is proven.

The value in the parenthesis on the right side of Equation (16) is a geometric series. Therefore, we can simplify Equation (16). The common ratio is $q = 2^{a-1}$ and the first term of the series is $a_0 = 2^{(\mathscr{N}-1)}\cdot 2^a$. Therefore, the sum of this geometric series is:

$$\begin{aligned} C^s_{comp} &= 2^{\mathscr{N}a} + 2\cdot 2^{(\mathscr{N}-1)a} + 2^2\cdot 2^{(\mathscr{N}-2)a} + 2^3\cdot 2^{(\mathscr{N}-3)a} + \ldots + 2^{(\mathscr{N}-1)}\cdot 2^a \\ &= 2^{(\mathscr{N}-1)}\cdot 2^a\cdot\frac{2^{\mathscr{N}a-\mathscr{N}}-1}{2^{a-1}-1} \end{aligned}$$

The server repeats the above number of computations b times. Thus, in the general case, where $b = 2^{c-(11+a\mathscr{N})}$, the computation complexity on the server side is:

$$C^s_{comp} = 2^{c-(11+a\mathscr{N})}\cdot 2^{(\mathscr{N}-1)}\cdot 2^a\cdot\frac{2^{\mathscr{N}a-\mathscr{N}}-1}{2^{a-1}-1} = \frac{\left(2^{\mathscr{N}a-\mathscr{N}}-1\right)\left(2^{c-11-(a-1)(\mathscr{N}-1)}\right)}{2^{a-1}-1}. \tag{17}$$

Moreover, the computation cost on the client side is $\mathscr{N}2^a$. This is because of the fact that the client computes 2^a encrypted values for each dimension. The computation complexity of the protocol is measured by the number of modular exponentiations, and it is:

$$C_{comp} = \frac{\left(2^{\mathscr{N}a-\mathscr{N}}-1\right)\left(2^{c-11-(a-1)(\mathscr{N}-1)}\right)}{2^{a-1}-1} + \mathscr{N}2^a. \tag{18}$$

6.2. Communication complexity

We measure the communication complexity by the number of

encrypted values. This is justified because all communication between the server and the client is encrypted. To get the communication complexity in bits, we need to multiply the number of encrypted values by 2^{12} (size of the modulus N^2).

The client sends 2^a encrypted values for each dimension to the server. This means the communication complexity of the protocol from the client side is $\mathcal{N}2^a$. Let us again assume that $b = 1$. We claim that the communication complexity from the server side is $2^{\mathcal{N}-1}$.

Proof by induction:

Base case: The statement holds for $\mathcal{N} = 2$, because the server generates 2 encrypted values for a matrix and therefore sends 2 encrypted values to the client.

Inductive step: We assume that the server generates 2^{k-1} encrypted values for a k-dimensional data structure. We know that a $(k+1)$-dimensional data structure consists of 2^a k-dimensional data structures. The server $\mathcal{S}$ computes $U_{1,0}$ and $U_{1,1}$ for each encrypted value and therefore generates $2 \times 2^{k-1}$ values. Thus, the statement holds.

The server generates the above number of encrypted values b times. Thus, in the general case where $b = 2^{c-(11+a\mathcal{N})}$, the communication complexity of the protocol measured by the number of encrypted values is

$$C_{comm} = 2^{c-(11+a\mathcal{N})} \cdot 2^{\mathcal{N}-1} + \mathcal{N}2^a. \tag{19}$$

6.3. Further discussion on relevant special cases

As it is explained above, $11 + a\mathcal{N} \le c$, and therefore, $a\mathcal{N} \le c - 11$. We have calculated the possible values for $(\mathcal{N}, a)$ when $15 \le c \le 55$. These choices of c give us a realistic range for the size of the Bloom/Cuckoo filter (from relatively small filters to large ones). For instance, when $c = 15$, the possible values for $(\mathcal{N}, a)$ are $(1,1), (1,2), (2,1), (1,3), (3,1), (1,4), (4,1)$ and $(2,2)$. However, we assume that $\mathcal{N} \ge 2$, because $\mathcal{N} = 1$ leads to the biggest possible computation cost from the server side. Therefore, we have omitted the pairs that have the value 1 for $\mathcal{N}$ and computed the communication and computation costs of the protocol with the remaining pairs. In other words, we numerically found the optimal values for $(\mathcal{N}, a)$ when $15 \le c \le 55$. Our calculations show that when $\mathcal{N} = 2$ and a is the maximum possible value among all possibilities explained above, the protocol has the minimal computation costs. We can calculate the maximal a by using the following equation:

$$\left\lceil \frac{c-11}{2} \right\rceil = a. \tag{20}$$

Equation (20) holds because by replacing $\mathcal{N} = 2$ in Equation (18), we get the following function for variable a:

$$C_{comp}(a) = 2^{c-11} + 2^{c-a-10} + 2^{a+1}.$$

In order to find the optimal value for a, we should solve:

$$\frac{dC_{comp}(a)}{da} = 0 + \ln(2)(-1)2^{c-a-10} + \ln(2)2^{a+1} = 0.$$

We get $2^{a+1} = 2^{c-a-10}$, and therefore, $\frac{c-11}{2} = a$. As a is always an integer, and $a - 1$ should be smaller than $(c - 11)/2$, it follows that $\left\lceil \frac{c-11}{2} \right\rceil = a$.

Similarly to what we've explained above, the optimum value for the communication complexity of the protocol can also be found numerically. Fig. 2 shows the optimum values for $\mathcal{N}$ and a to achieve the lowest bandwidth usage when $c \in \{15, 16, \ldots, 55\}$. We notice from Fig. 2 that when c increases, both $\mathcal{N}$ and a increase as well. The optimum point is that the dimension $\mathcal{N}$ is roughly 50% bigger than a. If $\mathcal{N}$ and a could be any real numbers, then the "optimal" line would be $a \approx 2/3\mathcal{N}$. Fig. 2 shows this optimal line.

We also studied the optimal values for $\mathcal{N}$ and a when $c \in \{15, 16, \ldots, 55\}$, with regards to the computation complexity. The result was that we always achieve the optimal computation complexity when $\mathcal{N} = 2$.

7. Implementation

In Sect. 5, we have presented a protocol to solve the problem of the PMT. In this section, we'll implement our protocol based on a realistic scenario. We assume that the server $\mathcal{S}$ has a database of 2^{21} (more than two million) malware samples. In order to prevent the malware from spreading, $\mathcal{S}$ computes the SHA-1 values of the database items and stores them in a set X. Each SHA-1 value has 160 bits; therefore, the size of X is 40 MB. A client $\mathcal{C}$ has a file with SHA-1 value x, and $\mathcal{C}$ wants to check whether the file is malicious or not.

As explained before, the client $\mathcal{C}$ wants to be sure that his/her file is clean, and therefore a false negative (meaning that the file contains malware but the result of the query shows that the file is clean) is

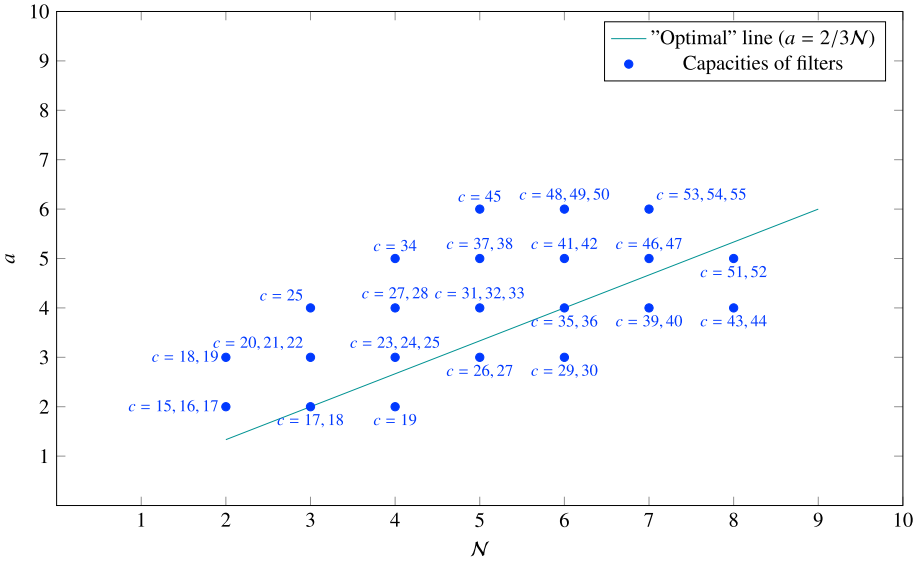

Fig. 2. In this plot, we present the values $\mathcal{N}$ and a when $c \in \{15, 16, \ldots, 55\}$ to achieve the minimal communication complexity. Parameter $\mathcal{N}$ is the number of dimensions in the data structure, 2^a is the number of entries per dimension in a hypercube, and 2^c is the size of the Bloom/Cuckoo filter. For each c, we found $\mathcal{N}$ and a such that the communication complexity is minimal.

considered to be unsafe in our setting. On the other hand, false positives are not as dangerous as false negatives. This asymmetry is due to the fact that, as explained earlier, in the case of positive response, the client sends the fingerprint of the file to the server. This means false positives would be noticed in this follow-up phase. The only harm is the loss of privacy with respect to the files that lead to false positives. Our proposed protocol has a small false positive rate but zero false negative rate. Therefore, among other reasons, utilizing our protocol is suitable for on-line malware checking.

We assume that revealing a small part of the database to $\mathscr{C}$ is acceptable to $\mathscr{S}$. Moreover, in order to make the protocol faster, in this section, we assume that the client reveals the first 4 bits of his/her SHA-1 value to the server. However, in the general case, the client does not need to reveal any bits of his/her SHA-1 value to the server.

7.1. Implementation on 2-dimensional approach

We implement the protocol as follows:

1. The client $\mathscr{C}$ reveals the first 4 bits of his/her SHA-1 value to the server.
2. The client $\mathscr{C}$ generates two distinct 2^{10}bit prime numbers p and q based on the setting of Paillier cryptosystem. Therefore, modulus $N = pq$ has 2^{11} bits. $\mathscr{C}$ picks g (for instance $g = N + 1$) and then defines $E_g(c, d)$ and $D_g(w)$ respectively as encryption and decryption functions.
3. The server $\mathscr{S}$ stores the 2^{21} SHA-1 values in 16 different subsets based on the first 4 bits of hash values. Therefore, each subset has approximately 2^{17} items. $\mathscr{S}$ knows which subset $\mathscr{C}$ is interested in.
4. $\mathscr{S}$ picks the value $a = 3$, and therefore divides the subset into 64 segments, each with approximately 2^{11} items. $\mathscr{S}$ inserts each segment into a Bloom filter with 10 hash functions or a Cuckoo filter. The false positive rate of the filters is 0.001. The value $\alpha = 95.5\%$ is picked for the Cuckoo filter. The size of the Bloom and Cuckoo filter are 2^{15} and 12.57×2^{11} bits, respectively.
5. $\mathscr{S}$ divides each filter into 16 parts ($b = 16$). Each part of the Bloom and Cuckoo filters has $2^{11} = 2048$ and 1608 bits, respectively. $\mathscr{S}$ arranges M_γ, where $1 \le \gamma \le 16$.
6. $\mathscr{C}$ calculates i^*, j^*, α and β, and sends N^2, α and β to $\mathscr{S}$.
7. For all matrices M_γ, $\mathscr{S}$ computes vectors σ^γ and U^γ.
8. The server $\mathscr{S}$ sends U and the hash functions of the filters to $\mathscr{C}$.
9. The client $\mathscr{C}$ decrypts all the sixteen results utilizing Equation (15), where $e = 1$, computes the binary representation of the results and concatenates them in the same order as the server has sent them and performs membership test without the server.

Communication complexities of this protocol are as follows: $\mathscr{C}$ sends $2^3 + 2^3$ encryptions to the server. These encryptions are of the size of modulus N^2 and therefore are of the size of 4096 bits. This means $\mathscr{C}$ sends 8 KB of data to $\mathscr{S}$. $\mathscr{S}$ generates 16×2 results, each of the size of the modulus N^2. Therefore, $\mathscr{S}$ sends 16 KB of data to $\mathscr{C}$.

The execution time of this setting depends on the processor which has been used to perform the computation. We used an x86-64 Intel Core i5 processor clocked at 2.7 GHz with a 4 MB L3 cache to implement this protocol. It takes 1.8 s for the client to encrypt 2×2^3 indices. The computation time on the server side depends on the filter which has been used. The server that utilizes a Bloom/Cuckoo filter does the computation on one matrix in 1.8 s/1.5 s. In order to calculate all 16 matrices, it is possible to run the protocol in parallel and keep the execution time as 1.8 s/1.5 s. If the server uses an Intel Xeon processor, the workload can be done with 32 threads and the total time for generating the set of responses will be 0.75 s. We assume that $\mathscr{C}$ first calculates the positions of the filter that correspond to value x and then decrypts just the responses that contain those positions. Therefore, decrypting all responses is not necessary for $\mathscr{C}$. The client $\mathscr{C}$ needs 0.5 s to decrypt one response and at most 5 s/1 s to retrieve 10 indices/2 indices of the Bloom/Cuckoo filter.

Let us now consider a modification where the client $\mathscr{C}$ makes a query for matrices M^γ one-by-one until he/she has enough information to decide whether $x \in X$. Then, the communication and computation complexities can be reduced on both sides on average. On the other hand, this modification would increase the average number of needed round-trips. In the best case, querying one matrix is enough to conclude that $x \notin X$. This is the case for both Bloom and Cuckoo filters. For Bloom filters, the worst case is that the client has to query for l matrices (where l is the number of hash functions in the Bloom filter). For Cuckoo filters, the client has to query at most two matrices. This modification gives, however, some extra information to $\mathscr{S}$ about x.

Actually, the communication complexity of our implemented protocol could be reduced in the case of the Cuckoo filter by storing the filter in 13 parts instead of 16 parts in step 5. This would imply similar reduction in communication complexity. However, the computational complexity would not be reduced significantly because bigger elements in the matrix would imply longer exponents in Equation (9).

Time complexities of our implementation are summarized in Table 3. If the client does not reveal the four bits of hash value, the processing time for $\mathscr{S}$ needs to be multiplied by 16. In this case, $b = 256$.

7.2. Implementation on 3-dimensional approach

To implement the protocol on a 3-dimensional database, we use the same parameters as explained in subsection 7.1. We assume that there are 2^{21} malware samples in the database of the server. The client reveals the first 4 bits of the hash value x. Now the server knows which subset the client is interested in. There are approximately 2^{17} samples in each subset. The server picks the value $a = 3$, divides the subset that the client is interested in into 512 segments and inserts each segment into a Bloom/Cuckoo filter. There are 4096 bits in each filter so the server gets 2 cubes of the size $2^3 \times 2^3 \times 2^3$ when each filter is divided into 2 segments as explained in section 5.

Communication complexities of this protocol on a 3-dimensional data structure are as follows: $\mathscr{C}$ sends $2^3 + 2^3 + 2^3$ encryptions to the server. These encryptions are of the size of the modulus N^2, i.e., 4096 bits. This means $\mathscr{C}$ sends 12 KB of data to $\mathscr{S}$. The server $\mathscr{S}$ generates 4 results for each cube and, a total of 2×4 results, each of the size of the modulus N^2. Therefore, $\mathscr{S}$ sends 4 KB of data to $\mathscr{C}$.

The time consumption of our implementations can be found in Table 4.

7.3. Implementation on 4-dimensional approach

The implementation of our protocol on a 4-dimensional database is

Table 3
Summary of time complexities using different filters on a 2-dimensional database. The size of the database is 2^{17}. We performed our protocol on a $2^3 \times 2^3$ matrix.

Protocol on a 2-dimensional database	Preprocessing by $\mathscr{C}$	Query for $\mathscr{S}$	Query for $\mathscr{C}$
Our Protocol with Bloom Filter	1.8 s	0.9 s	≤5 s
Our Protocol with Cuckoo Filter	1.8 s	0.75 s	≤1 s

Table 4
Summary of time complexities using different filters on a 3-dimensional database. The size of the database is 2^{17}. We performed our protocol on a $2^3 \times 2^3 \times 2^3$ cube.

Protocol on a 3-dimensional database	Preprocessing by $\mathscr{C}$	Query for $\mathscr{S}$	Query for $\mathscr{C}$
Our Protocol with Bloom Filter	2.7 s	0.95 s	≤12 s
Our Protocol with Cuckoo Filter	2.7 s	0.8 s	≤2.5 s

Table 5
Summary of time complexities using different filters on a 4-dimensional approach. The size of the database is 2^{17}. We performed our protocol on a $2^2 \times 2^2 \times 2^2 \times 2^2$ 4-hypercube.

Protocol on a 4-dimensional database	Preprocessing by $\mathscr{C}$	Query for $\mathscr{S}$	Query for $\mathscr{C}$
Our Protocol with Bloom Filter	1.8 s	1.6 s	≤26 s
Our Protocol with Cuckoo Filter	1.8 s	1.4 s	≤5.3 s

similar to the previous subsections. For simplicity, we use the same parameters as explained in Subsection 7.1. Here again, the client reveals the first 4 bits of the hash value x and there are approximately 2^{17} samples in the subset that the client is interested in. The server arranges a 4-dimensional hypercube of the size $2^2 \times 2^2 \times 2^2 \times 2^2$, divides the subset that the client is interested in into 256 segments, and inserts each segment into a Bloom/Cuckoo filter and then into four 4-dimensional hypercubes as explained in Section 5.

Communication complexities of this protocol on the 4-dimensional database are as follows: $\mathscr{C}$ sends $2^2+2^2+2^2+2^2$ encryptions to the server. These encryptions are of the size of the modulus N^2 and therefore are of the size of 4096 bits. This means $\mathscr{C}$ sends 8 KB data to $\mathscr{S}$. The server $\mathscr{S}$ computes 8 results per data structure, therefore, $\mathscr{S}$ generates 4×8 results, each of the size of the modulus N^2. Therefore, $\mathscr{S}$ sends 16 KB data to $\mathscr{C}$.

Table 5 shows the efficiency of our implementations.

7.4. Performance evaluation

So far we assume that the size of the database is 2^{21}. In the malware case, this database size is realistic. Moreover, we assume that the client reveals the first four bits of his/her hash value to the server. This keeps the execution time of the protocol (on the server side) less than 1 s. Now, we repeat the same implementation for databases of bigger sizes. Hereafter, we assume that the client does not reveal any bits of his/her hash item.

In this part, we evaluate the performance of our protocol for 3 different database sizes on 3 different dimensions of the data structures. We report the summary of complexities in Table 6 when $a = 3$, and in Table 7 when $a = 4$. The size of modulus N is 2048 bits. The elements in the matrix, cube, etc., are of the size of 2048 bits, except when $a = 4$, the data structure is 4-dimensional and the database size is 2^{21}. In this case, the entries of the 4-hypercube are of the size of 512 bits, because there are 2^{16} entries in this data structure and a total of 2^{25} bits of data. Tables 6 and 7 are also illustrated in Fig. 3.

We assume that the server uses 32 threads to compute the PIR on hypercubes.

For simplicity, we assume in this paper that the database is inserted into a symmetrically shaped data structure. In the more general case, the data structure can also be a hyper-rectangle, i.e., the number of entries for each dimension can be different. In other words, the choice of a could be varied in each dimension, but we leave this for further study.

We analyse the performance of our protocol for databases of sizes 2^{21}, 2^{23} and 2^{25}, and data structures of three different dimensions where the value for parameter a is 3 or 4. Next, we explain how we choose parameters for the performance analysis. It can be seen from Fig. 2 that the optimal parameter combinations with respect to communication complexity are as follows: for database size 2^{25}, we have $c = 29$, and the optimum is reached when $a = 3, \mathscr{N} = 6$. For database size 2^{23}, we have $c = 27$, and there are two optimal combinations: $a = 3, \mathscr{N} = 5$; and $a = 4, \mathscr{N} = 4$. For database size 2^{21}, we have $c = 25$, and there are, again, two optimal combinations: $a = 3, \mathscr{N} = 4$; and $a = 4, \mathscr{N} = 3$. On the other hand, from the point of view of computational complexity, the optimal combinations are as follows:

- for database size 2^{25}, $a = 9, \mathscr{N} = 2$;
- for database size 2^{23}, $a = 8, \mathscr{N} = 2$;

Table 6
Sing different database sizes on 3 different dimensions when the value of $a = 3$.

DB Size	Dimensions	b	Preprocessing by $\mathscr{C}$	Query for $\mathscr{S}$	Query for $\mathscr{C}$	Comm. $\mathscr{C} \to \mathscr{S}$	Compl. $\mathscr{S} \to \mathscr{C}$
2^{21}	2-dimensional	2^8	1.8 s	14.4 s	0.5 s	8	256
	3-dimensional	2^5	2.7 s	20.6 s	1.2 s	12	64
	4-dimensional	2^2	3.6 s	22.2 s	2.6 s	16	16
2^{23}	2-dimensional	2^{10}	1.8 s	57.6 s	0.5 s	8	1024
	3-dimensional	2^7	2.7 s	82.4 s	1.2 s	12	256
	4-dimensional	2^4	3.6 s	88.8 s	2.6 s	16	64
2^{25}	2-dimensional	2^{12}	1.8 s	230.4 s	0.5 s	8	4096
	3-dimensional	2^9	2.7 s	329.6 s	1.2 s	12	1024
	4-dimensional	2^6	3.6 s	355.2 s	2.6 s	16	256

Table 7
Summary of complexities using different database sizes on 3 different dimensions when the value of $a = 4$.

DB Size	Dimensions	b	Preprocessing by $\mathscr{C}$	Query for $\mathscr{S}$	Query for $\mathscr{C}$	Comm. $\mathscr{C} \to \mathscr{S}$	Compl. $\mathscr{S} \to \mathscr{C}$
2^{21}	2-dimensional	2^6	3.6 s	13 s	0.5 s	16	64
	3-dimensional	2^2	5.4 s	15.8 s	1.2 s	24	8
	4-dimensional	1	7.2 s	20 s	2.6 s	32	4
2^{23}	2-dimensional	2^8	3.6 s	52 s	0.5 s	16	256
	3-dimensional	2^4	5.4 s	63.2 s	1.2 s	24	32
	4-dimensional	1	7.2 s	64.8 s	2.6 s	32	4
2^{25}	2-dimensional	2^{10}	3.6 s	208 s	0.5 s	16	1024
	3-dimensional	2^6	5.4 s	252.8 s	1.2 s	24	128
	4-dimensional	2^2	7.2 s	259.2 s	2.6 s	32	16

Table 8
Comparison of performances between our protocol and the previous arts, when the set size is 2^{21}, and the time complexity is approximated for an Intel processor. Here the communication complexity of the protocol by Pinkas et al. is obtained from Table 7 of [16], where $n_1 = 2^{21}$ and $n_2 = 1$. In this table, perfect privacy and zero privacy are respectively denoted by $++$ and $-$. Moreover, revealing a few bits of extra information to the other party is shown by $+$, and $+-$ shows that the client obtains even more information.

Protocol	Privacy for Client	Privacy for Server	Communication Complexity	Time Complexity Pre-process	Time Complexity On-line
Pinkas et al. [16]	$++$	$++$	80 MB	0	sec
Meskanen et al. [19]	$++$	$+$	4 MB	hours	ms
Trivial Solution	$++$	$-$	4 MB	0	minimal
Our Protocol on 2-d database	$+$	$+-$	24 KB	sec	sec
Our Protocol on 3-d database	$+$	$+-$	16 KB	sec	sec
Our Protocol on 4-d database	$+$	$+-$	24 KB	sec	sec

- for database size 2^{21}, $a = 7$, $\mathscr{N} = 2$.

We have chosen values $a = 3$ and $a = 4$ because of the low communication complexity, whereas low dimension values $\mathscr{N} = 2$, $\mathscr{N} = 3$ and $\mathscr{N} = 4$ are chosen because of the low computational complexity2^{25}.

Fig. 3 shows that 2-dimensional data structures are the best choices from the point of view of time complexity, and 4-dimensional data structures are the worst. From the point of view of communication complexity, the performances are in the opposite order for databases of sizes 2^{23} and 2^{25}.

One observation for databases of sizes 2^{23} and 2^{25} is that $a = 4$ has better performance than $a = 3$. The 2-dimensional data structure when $a = 3$ always has the worst communication complexity, the 2-dimensional structure with $a = 4$ and the 3-dimensional structure with $a = 3$ are the next. The choice of a 4-dimensional structure with $a = 4$ gives a significantly smaller communication complexity than the 2-dimensional data structures. For the database size 2^{21}, the order of different parameter combinations is not the same as that for databases of bigger sizes.

8. Security and privacy analysis

In this section, we present security and privacy analyse of the protocol. We analyse the main protocol against the semi-honest client, the malicious client, the semi-honest server and the malicious server.

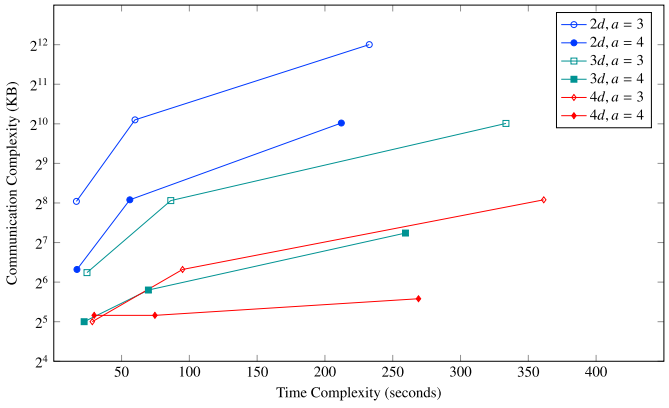

Fig. 3. In each line, for each of the six cases, there are three data points and from left to right. They represent databases of the sizes of 2^{21}, 2^{23} and 2^{25}. The x-axis shows the time complexity of the protocol in seconds, and the y-axis shows the communication complexity of the protocol in KB. The blue lines present the 2-dimensional hypercubes, the green lines present the 3-dimensional hypercubes and the red ones present the 4-dimensional hypercubes. The open dots are used when the value of parameter a is equal to and the closed dots correspond to the value 4 for parameter a.

Theorem 1. *Any server, semi-honest or malicious, does not learn anything about the client's hash value nor the result of the membership test.*

Proof. The client encrypts the position of x in the $\mathscr{N}$-dimensional hypercube M utilizing Paillier cryptosystem. Only information that $\mathscr{S}$ gets from $\mathscr{C}$ is in the first message of the protocol. The message contains encrypted zeros and ones that provide information about x. But to get even partial information, $\mathscr{S}$ would need to break the semantic security of the Paillier cryptosystem. □

In our practical case, the client reveals which 1/16th of the database his/her element belongs to before the protocol starts. Therefore, the server learns this information about x. We remark that if there is a significant correlation between the queries of two different clients, the server can guess that these two clients are in possession of common files.

It is not possible to reach a similar result for the secrecy of database. But the following theorem gives an upper bound for the amount of information that is leaked during the protocol.

Theorem 2. *The client, semi-honest or malicious, learns at most $2^{\mathscr{N}-1}b \cdot \log_2(N^2)$ bits of information about X, where b is the number of matrices that M has been divided into, N^2 is the modulus in Paillier, and $\mathscr{N}$ is the number of dimensions in the data structure.*

Proof. During the protocol, the server sends $2^{\mathscr{N}-1}$ Paillier cryptotexts of length $\log_2(N^2)$ bits for each small hypercube. There are b small $\mathscr{N}$-dimensional hypercubes, so $\mathscr{S}$ sends $2^{\mathscr{N}-1}b \cdot \log_2(N^2)$ bits to the client. Therefore, the maximum amount of information that any client can learn from one round of the protocol is $2^{\mathscr{N}-1}b \cdot \log_2(N^2)$ bits □.

The client who follows the protocol will retrieve one element of M, that is $(n/2^{2a}) \cdot K$ bits, where value K is the number of bits required per element in the filter and can be computed using Table 1. In our case, the most amount of information that the malicious client may learn is 2^{17} bits, which is four times the amount of information that the honest client may learn, that is 2^{15} bits of the Bloom/Cuckoo filter. Thus the amount of information that malicious $\mathscr{C}$ may learn is not much more than what the honest $\mathscr{C}$ *learns*.

In our application scenario, we assume that the client who gets a positive result for the PMT reveals x to the server to handle the potential malware. In the case of a true positive, this is fine. But in the case of a false positive, the server learns the hash value of the client's file. We assume that this is acceptable when the probability of false positive is small. There is always the possibility that a malicious server changes the filters in such a way that the probability of false positives is greater at least for certain elements. This kind of cheating could only be detected in the long run if the frequency of false positives is detected to be too large.

In Table 8, we compare the privacy and complexity aspects of our protocol with those of the protocols of [16,19], and with the trivial solution where the server just sends the whole Bloom/Cuckoo filter to the client, who then makes the query by himself/herself. In this table, $++$ means the best performance from the point of view of privacy, and $-$ means the worst one.

Table 9
Summary of notations.

Notation	Description	Notation	Description
a	2^a entries per dimension in a hypercube	b	Number of hypercubes
b	Num. of entries per bucket in Cuckoo filter	c	2^c is the size of Bloom/Cuckoo filter
c	Plaintext in Paillier cryptosystem	C	Average bits per item in Cuckoo filter
$\boldsymbol{C}$	Ciphertext space	$\mathscr{C}$	Client
D_{s_k}	Decryption function	E_k	Encryption function
e	Integer e is such that $N^e < \max(\mathbf{C}) \leq N^{(e+1)}$	$\mathbf{K}$	Key space
k	Public key	$\mathscr{K}$	Hypercube
l	Number of hash functions of Bloom filter	l_C	Length of Ciphertext
l_P	Length of Plaintext	m	Number of bits in Bloom filter
$\boldsymbol{M}$	Matrix	m'	Number of buckets in Cuckoo filter
$\boldsymbol{N}$	Public key in Paillier cryptosystem	n	Number of items in the database
$\mathscr{N}$	Number of dimensions	OT	Oblivious Transfer
$\boldsymbol{P}$	Plaintext space	PIR	Private information retrieval
PMT	Private membership test	PSI	Private Set Intersection
$\mathscr{S}$	Server	s_k	Secret key
U	A set of encrypted values generated by $\mathscr{S}$	x	Client's search item
X	Server's database	α	Encrypted vector generated by client
β	Encrypted vector generated by client	η	Encrypted vector generated by client

9. Conclusion

In this paper, we propose a practical protocol for privacy-preserving database queries. We utilize Bloom filters or Cuckoo filters to find out whether a certain item is in the database. This implies we allow a small number of false-positive outcomes but, on the other hand, rule out false negatives completely.

As another building block in our protocol, homomorphic encryption is used. We utilize the scheme of Chang [6] to do a search in the Bloom/Cuckoo filter in such a way that the server which who holds the database does not learn anything else about the query than what happened.

Our protocol has a much lower communication complexity than prior schemes, and its computation complexity is also low enough for practical use cases.

We measure the performance of our protocol in a realistic scenario: A server $\mathscr{S}$ has a database of 2^{21} malware samples, and an anti-malware client wants to check a file against this database in a privacy-preserving manner. Note that in this scenario, false positives are much less serious errors than false negatives. Our implementation shows that the proposed protocol can be used in real-world applications, for example, in Android apps or website reputation services.

Utilizing the Cuckoo filter rather than the Bloom filter makes the protocol slightly faster and more space-efficient. Moreover, the Cuckoo filter has richer functionality because it also supports the deletion of items from the database.

We use the Paillier cryptosystem for homomorphic encryption. Future work could try to find better performance with some other cryptosystems. Another direction for future work is to apply our protocol for a wider selection of use cases.

We evaluate the performance of our protocol for databases with 2^{21}, 2^{23}, and 2^{25} items, and data structures with 2, 3 and 4 dimensions. Fig. 3 shows the results of this evaluation. One possible direction for future work is to evaluate the performance of our protocol, utilizing hyper-rectangles, i.e., matching the number of entries different for each dimension.

In order to evaluate the communication and computation costs of the protocol, one can utilize Equations (16) and (17). We numerically find the optimal values for parameters a and $\mathscr{N}$ that result in the lowest possible computation and communication costs.

A table of notations is shown in Table 9.

Acknowledgements

We thank the anonymous reviewers of the Digital Communications and Networks journal, for their insightful comments and suggestions on this paper. We also thank Dr. Roope Vehkalahti for his helpful comments on this paper. This work was supported in part by Tekes project "Cloud-assisted Security Services" grant number 3887/31/2016 and by the Academy of Finland project "Cloud Security Services" (283135).

References

[1] M. Kosinski, D. Stillwell, T. Graepel, Private traits and attributes are predictable from digital records of human behavior, Proc. Natl. Acad. Sci. Unit. States Am. 110 (15) (2013) 5802–5805.
[2] P. Zhang, J. Ma, Channel characteristic aware privacy protection mechanism in WBAN, Sensors 18 (8) (2018) 2403.
[3] S. Seneviratne, A. Seneviratne, P. Mohapatra, A. Mahanti, Predicting user traits from a snapshot of apps installed on a smartphone, ACM SIGMOB - Mob. Comput. Commun. Rev. 18 (2) (2014) 1–8.
[4] D. Wu, S. Si, S. Wu, R. Wang, Dynamic trust relationships aware data privacy protection in mobile crowd-sensing, IEEE Internet Things J. 5 (4) (2018) 2958–2970.
[5] J. Xiong, J. Ren, L. Chen, Z. Yao, M. Lin, D. Wu, B. Niu, Enhancing privacy and availability for data clustering in intelligent electrical service of IoT, IEEE Internet Things J. 6 (2) (April 2019) 1530–1540.
[6] Y.-C. Chang, Single database private information retrieval with logarithmic communication, in: Australasian Conference on Information Security and Privacy, Springer, 2004, pp. 50–61.
[7] B.H. Bloom, Space/time trade-offs in hash coding with allowable errors, Commun. ACM 13 (7) (1970) 422–426.
[8] B. Fan, D.G. Andersen, M. Kaminsky, M.D. Mitzenmacher, Cuckoo filter: practically better than bloom, in: Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies, ACM, 2014, pp. 75–88.
[9] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, G. Varghese, An improved construction for counting bloom filters, in: European Symposium on Algorithms, Springer, 2006, pp. 684–695.
[10] R.L. Rivest, L. Adleman, M.L. Dertouzos, On data banks and privacy homomorphisms, Found. Secure Comput. 4 (11) (1978) 169–180.
[11] P. Paillier, D. Pointcheval, Efficient public-key cryptosystems provably secure against active adversaries, in: International Conference on the Theory and Application of Cryptology and Information Security, Springer, 1999, pp. 165–179.
[12] B. Chor, N. Gilboa, M. Naor, Private Information Retrieval by Keywords, CiteSeer, 1997.
[13] E. Kushilevitz, R. Ostrovsky, Replication is not needed: single database, computationally-private information retrieval, in: Proceedings., 38th Annual Symposium on Foundations of Computer Science, IEEE, 1997, pp. 364–373. Foundations of Computer Science, 1997.
[14] W. Gasarch, A survey on private information retrieval, Bull. EATCS 82 (2004) 72–107.

[15] C. Gentry, Z. Ramzan, Single-database private information retrieval with constant communication rate, in: International Colloquium on Automata, Languages, and Programming, Springer, 2005, pp. 803–815.
[16] B. Pinkas, T. Schneider, M. Zohner, Scalable Private Set Intersection Based on OT Extension, in submission), http://eprint.iacr.org/2016/930.
[17] M.O. Rabin, How to exchange secrets with oblivious transfer, IACR Cryptol. ePrint Archive 2005 (2005) 187.
[18] S. Tamrakar, J. Liu, A. Paverd, J.-E. Ekberg, B. Pinkas, N. Asokan, The Circle Game: Scalable Private Membership Test Using Trusted Hardware, arXiv preprint arXiv: 1606.01655.
[19] T. Meskanen, J. Liu, S. Ramezanian, V. Niemi, Private membership test for bloom filters, in: Trustcom/BigDataSE/ISPA, vol. 1, IEEE, 2015, pp. 515–522. IEEE, 2015.