

**Algorithm 1** Setup Algorithm for *Graph Enc*1



**Input:** A secret key *K* , a key pair *( pk, sk)*, an approximation ratio *α*, an amplification factor *φ*, and an original graph *G*.

**Output:** The encrypted graph index *6*.

˜

1: Generate the *2-hop labeling index 6* = {*6out, 6in* } from

*G*.

2: Initialize two dictionaries *Iout* and *Iin* .

3: Let be the maximum distance over all the sketches and set *N* 2 1.

= *B* +

*B*

4: **for** each *u G* **do**

∈

5: Set *Tout,u h(K, u* 1*)*, *Tin,u h(K, u* 2*)*.

= || = ||

6: **for** each *(v, du,v, cu,v ) 6out(u)* **do**

∈

7: Compute *V h(K,v* 0*)*.

= ||

8: Compute *Du,v* SWHE.Enc*( pk,* 2*N* −*du,v )*.

=

9: Compute *Cu,v* ORE.Enc*(K, φcu,v)*.

=

10: Insert (*V, Du,v , Cu,v* ) into the dictionary *Iout Tout,u* .

[ ]

11: **end for**

12: Repeat the above procedure for each sketch in *6in(u)*

integer and should be carefully selected to enlarge the plaintext space of *cu,v*. In practice, the product of *φ* and the maximum cost value over all the sketches should be sufficiently large (e.g., at least 280), which is used to provide a sufficient randomness to the inputs. Since *φ* is kept private by the *user*, the *cloud server* cannot learn the real values of *cu,v* .

* **Query:** To perform an *α*-CSD query with an origin *s*, a destination *t*, and a cost constraint *θ* , the *user* generates query tokens *τs h(K, s* 1*)* and *τt h(K, t* 2*)*, and sends them to the *cloud server*. The *cloud server* obtains *Iout τs* and *Iin τt* from the index. For each encrypted vertex identifier *v* that appears in both *Iout τs* and *Iin τt* , the *cloud server* performs a cost constraint filtering operation (which will be described in details in Section VI), and adds each pair *(Ds,v, Dv,t)* which satisfies the cost constraint *φθ* into a candidate set *Y* . Note that the cost constraint is multiplied by *φ* because

[ ]

[ ]

[ ] [ ]

= || = ||

Σ|*Y* |

and add entries into *Iin Tin,u* .

[ ]

13: **end for**



we encrypt the cost *φcu,v*, instead of *cu,v*. Then, the *cloud server* directly obtains *d* =

*i s,v*

*i*

*v,t*

*s*

*v,*

*i*=1 *di* ,

14: **return** *6*˜ = {*Iout, Iin* } as the encrypted graph index.

where *di* = SWHE.Eval*(*×*, Di,v, Di t)* for each pair

*(D*

*, D*

*)* in *Y* . The correctness of the above calcula-

* 1. CONSTRUCTION OF Connor

In this section, we introduce our graph encryption scheme

Connor for the privacy-preserving *α*-CSD querying.

1. *Construction Overview*

The construction process is based on two particular pseudo- random functions *h* and *g*, and a somewhat homomorphic encryption (SWHE) scheme. In this paper, we adopt the concrete instantiation of a SWHE scheme in the literature [16]. The parameters of *h* and *g* are illustrated in Equation (1),

*h* : {0*,* 1}*λ* × {0*,* 1}∗ → {0*,* 1}*λ* (1a)

*g* : {0*,* 1}*λ* × {0*,* 1}∗ → {0*,* 1}*λ*+*z*+*k* (1b)

where *λ* is the security parameter, and *k* and *z* are the output lengths of the ORE and SWHE encryptions, respectively.

We start with a straightforward construction *Graph Enc*1

=

*(Key Gen, Setup, Query)* as follows, including:

* + **KeyGen:** Given the security parameter *λ*, the *user* ran- domly generates a secret key *K* and a pair of public and secret keys *( pk, sk)* for SWHE.
  + **Setup:** Given an original graph *G*, an approximation ratio *α*, and an amplification factor *φ*, the *user* obtains the encrypted graph index by using Algorithm 1. The 2HCLI

*6 6out, 6in* of *G* can be generated by the method described in Section III-B.

= { }

Let be the maximum distance over all the sketches and *N* 2 1. Motivated by the literature [1], each distance *du,v* is encrypted as 2*N* −*du,v* by the SWHE to protect its

= *B* +

*B*

real value (line 8). Considering that 2*x* 2*y* is bounded by 2*max(x,y)*−1, the SWHE encryption of distance allows for obtaining the minimum sum over a certain number of

+

distance pairs.

Each cost *cu,v*, multiplied by the amplification factor *φ*, is encrypted by the ORE encryption (line 9). *φ* is a big

tion follows homomorphic properties of SWHE. We refer the readers to [1] for more details.

Finally, the *cloud server* returns *d* to the *user*, who, in turn, obtains the answer to the *α*-CSD query by decrypting *d* with its secret key *sk*.

Note that this straightforward approach does not only cor- rectly answer the *α*-CSD query over encrypted graphs, but also protects the vertex identifier, distance, and cost information.

However, the encrypted graph index obtained from Algorithm 1, without performing any queries, still results in information leakage. On one hand, it reveals the length of each encrypted sketch, i.e., *Iout u* and *Iin u* , as well as the order information of ORE-encrypted costs in all sketches. On the other hand, it also discloses the number of com- mon vertices between *Iout u* and *Iin v* , which indicates the number of vertices that connect *u* to *v*. In particular, if the *cloud server* knows that there is no common vertex between

[ ] [ ]

[ ] [ ]

*Iout* [*u*] and *Iin* [*v*], it learns that *u* cannot reach *v*.

1. *Privacy-preserving α-CSD Querying*

In order to enhance protection of sensitive information, we construct a privacy-preserving *α*-CSD querying scheme *Graph Enc*2 *(Key Gen, Setup, Query)*, where the key generation procedure is the same as in *Graph Enc*1, with improved index construction and CSD query procedures as exhibited in Algorithms 2 and 3, respectively.

=

The *Setup* for *Graph Enc*2 works as follows. The *user* first builds the 2HCLI *6* of graph *G*, and then encrypts sketches associated with *u G* (i.e., *6out(u)* and *6in(u)*), as described in lines 2-17.

∈

Note that in order to prevent the leakage of the sketch size in the previous straightforward approach, we split each encrypted sketch *Iout(u)* and *Iin (u)*, and ensure that they are stored in the dictionary separately, with a size of one. More precisely, we utilize a counter *ω* and generate the unique

**Algorithm 2** Setup Algorithm for *Graph Enc*2



**Input:** A secret key *K* , a key pair *( pk, sk)*, an approximation ratio *α*, an amplification factor *φ*, and an original graph *G*.

**Output:** The encrypted graph index *6*.

˜

1: Generate the 2HCLI *6 6out, 6in* of *G*.

= { }

2: Initialize two dictionary *Iout* and *Iin* .

3: Let be the maximum distance over the sketches and set

*B*

*N* 2 1.

= *B* +

4: **for** each *u G* **do**

∈

5: Set *Sout,u h(K, u* 1*)*, *Tout,u h(K, u* 2*)*, *Sin,u*

= || = || =

*h(K, u* 3*)*, and *Tin,u h(K, u* 4*)*.

|| = ||

6: Initialize a counter *ω* 0

=

7: **for** each *(v, du,v, cu,v ) 6out(u)* **do**

∈

8: Compute *V h(K,v* 0*)*.

= ||

9: Compute *Du,v* SWHE.Enc*( pk,* 2*N* −*du,v )*.

=

10: Compute *Cu,v* ORE.Enc*(K, φcu,v)*.

=

11: Set *Tout,u,v h(Tout,u, ω)* and *Sout,u,v*

= =

*g(Sout,u, ω)*.

12: Compute *Tu,v Sout,u,v (V Du,v Cu,v)*.

= ⊕ || ||

13: Set *Iout Tout,u,v Tu,v* .

[ ]=

14: Set *ω ω* 1.

= +

15: **end for**

16: Repeat the above procedure for each sketch in *6in(u)* and obtain *Iin Tin,u,v* , except that: (i) set *Tin,u,v h(Tin,u, ω)* and *Sin,u,v g(Sin,u, ω)*, and (ii) com- pute *Tu,v Sin,u,v (V Du,v Cu,v)*.

= ⊕ || ||

=

[ ] =

17: **end for**

18: **return** *6*˜ = {*Iout, Iin* } as the encrypted graph index.



**Algorithm 3** Query Algorithm for *Graph Enc*2



**Input:** The *user*’s input are the secret key *K* , secret key pair

*( pk, sk)*, an amplification factor *0*, and the query *q*

=

*(s, t,θ)*. The *cloud server*’s input is the encrypted index

˜

*6*.

**Output:** *user*’s output is *distq* and *cloud server*’s output is

.

⊥

1: *user* generates *Sout,s h(K, s* 1*)*, *Tout,s h(K, s* 2*)*,

= || = ||

*Sin,t h(K, t* 3*)* and *Tin,t h(K, t* 4*)*.

= || = ||

2: *user* constructs a cost constraint tree *Tθ* based on *φ θ*

∗

using secret *K* as described in Section VI.

3: *user* sends *τs,t (Sout,s, Tout,s, Sin,t , Tin,t , Tθ )* to *cloud server*.

=

4: *cloud server* parses *τs,t* as *(Sout,s, Tout,s, Sin,t , Tin,t , Tθ )*.

5: *cloud server* initializes a set *Ls* and a counter *ω* 0.

=

6: *cloud server* computes *Tout,s,v h(Tout,s, ω)*.

=

7: **while** *Iout Tout,s,v* **do**

[ ] /=⊥

8: *cloud server* computes *Sout,s,v g(Sout,s, ω)*.

=

9: *cloud server* performs *(V Ds,v Cs,v) Ts,v Sout,s,v*.

|| || = ⊕

10: *cloud server* add *(V, Ds,v, Cs,v)* into *Ls*.

11: Set *ω ω* 1.

= +

12: *cloud server* computes *Tout,s,v h(Tout,s, ω)*.

=

13: **end while**

14: *cloud server* initializes a set *Lt* and a counter *ω* 0.

=

15: *cloud server* computes *Tin,v,t h(Tin,t , ω)*.

=

16: **while** *Iin Tin,v,t* **do**

[ ] /=⊥

17: *cloud server* computes *Sin,v,t g(Sin,t , ω)*.

=

18: *cloud server* performs *(V Dv,t Cv,t) Tv,t Sin,v,t* .

|| || = ⊕

19: *cloud server* add *(V, Dv,t , Cv,t)* into *Lt* .

20: Set *ω* = *ω* + 1.

=

*Tout,u,v* and *Sout,u,v* for each entity in *6out(u)* (line 11). Similarly, the unique *Tin,u,v* and *Sin,u,v* for each entity in *6in (u)* can be generated (line 16). The *Tout,u,v* (or *Tin,u,v* ) indicates the position that this entity will be stored in *Iout* (or *Iin* ), which ensures each position in the dictionary *Iout* (or *Iin* ) having only one entity.

*Sout,u,v* (or *Sin,u,v* ) is used to make an XOR operation with *(V* ||*Du,v*||*Cu,v)*. Since *Sout,u,v* (or *Sin,u,v* ) is different

21: *cloud server* computes *Tin,v,t h(Tin,t , ω)*.

22: **end while**

23: For each encrypted vertex identifier *v* that appears in both in *Ls* and *Lt* , the *cloud server* performs the cost constraint filtering operation through Algorithm 4, and add the pair *(Ds,v, Dv,t)* which satisfies the cost constraint *φθ* into a set *Y* . The pair that Algorithm 4 cannot verify is also added into *Y* .

24: For each pair in *Y* , the *cloud server* first computes *di* =

*, D*

for each sketch, the XOR operation makes the resulting *Tu,v*

SWHE.Eval*(*×*, Di*

*i*

*v,t*

*s,v*

Σ

*)*, and then computes *d* =

indistinguishable, which guarantees that the *static* encrypted graph index *6*˜ reveals neither the number of common vertices

25:

|*Y* |

*i*=1

*di* .

returns *d* to the *user*.

between *Iout(u)* and *Iin (v)*, nor the order information of costs.

The *Query* in Algorithm 3 works as follows. Assume that the *user* asks for the shortest distance between *s* and *t*, whose total cost does not exceed *θ* . She first generates the query token *τs,t* and sends it to the *cloud server* (lines 1-3). Upon receiving the token *τs,t*, the *cloud server* searches in the index and obtains *Ls* and *Lt* (lines 5-22). That is, the *cloud server* iteratively judges whether the dictionary *Iout* (*Iin* ) contains the key *Tout,s,v* (*Tin,v,t* ) or not. If it exists, then it adds the corresponding entity into the set *Ls* (*Lt* ).

Once *Ls* and *Lt* are obtained, the *cloud server* performs the cost constraint filtering (line 23) and computes *d* (line 24), which are the same as described in the straightforward approach. Finally, the *user* gets the final answer by decrypt- ing *d*, which is returned by the *cloud server*, using its *sk*.

*cloud server*

26: *user* decrypts *d* with *sk*.

27: **return** Decrypted value of *d* as *distq*.



* 1. TREE-BASED CIPHERTEXTS COMPARISON APPROACH

This section introduces a tree-based ciphertexts comparison approach, which is used for cost constraint filtering in the graph encryption scheme described in Section V.

1. *Scenarios*

Assume that there is a *user* (i.e., *U* ) and a *server* (i.e., *R*). *U* has many integers which are encrypted by a kind of cryptography algorithm and then outsourced to *R*. Now, *U* wants to ask for *R* to obtain integer pairs, e.g., (*x* , *y*),

whose sum does not exceed *θ* . Note that the plaintexts of *x* , *y* and *θ* could not be disclosed to , except for the greater- than, equality, or less-than relationship. A naive approach is to download all the integers, calculate the summation locally, and choose the integer pairs satisfying the constraint. This method, however, is meaningless if one wants to offload the computation to the cloud. Hence, it is desirable to have a practical solution to this problem.

*R*

Note that this scenario is different from the well-known SMC scheme. In the setting of SMC [41], [42], a set of (two or more) parties with private inputs wish to compute a function of their inputs while revealing nothing but the result of the function, which is used for many practical applications, such as exchange markets. SMC is a collaborative computing problem that solves the privacy preserving problem among a group of mutually untrusted participants. The ciphertexts of all pairs of (*x* , *y*) and the cost constraint *θ* are outsourced to the cloud server, which is responsible for the inequality tests. Furthermore, we could reveal the relationship between the sum of two ciphertexts and another ciphertext to the *server*, which is referred to as *controlled disclosure* in the literature [17].

It seems that we might leverage the homomorphic encryp- tion technique, since it supports a sum operation of calculating *x y*. Nevertheless, as the homomorphic encryption is prob- abilistic, we are unable to determine the relationship between

+

*x* + *y* and *θ* over their ciphertexts.

1. *Main Idea*

The main idea of the tree-based ciphertexts comparison protocol is to encode an integer with the ORE primitive. To the best of our knowledge, none of the existing approaches can support ORE and homomorphism properties simultaneously. Hence, we design a novel method to address this problem, which is motivated by the following facts.

If we want to compare *x y* with *θ* , we can compare *x* with *θ/*2 and *y* with *θ/*2, respectively. Now, we result in 4 possible cases corresponding to combinations of the two relationships. If *x > θ/*2 (*x θ/*2) and *y > θ/*2 (*y θ/*2), we can know that *x y > θ* (*x y θ* ). In the rest two cases, i.e., *x > θ/*2 and *y < θ/*2, or *x θ/*2 and *y θ/*2, we cannot achieve a deterministic result. At this point, we can further divide *θ/*2 into *θ/*4. And then we can compare *x* and *y* with *θ/*4 and 3*θ/*4, respectively.

+

≤ ≥

+ + ≤

≤ ≤

By iteratively performing such an operation, we can deter- mine the relationship between *x y* and *θ* with an increasing probability. Due to the ORE property, it is easy to perform the above operations over ciphertexts. Next, we will show how to implement this idea efficiently by utilizing a tree structure.

+

1. *Details of Protocol*

To implement the comparison of *x y* and *θ* over their ciphertexts, we construct a *cost constraint tree*, whose nodes represent specific values that are related to *θ* . For clarity, we define *E(m)* as the ORE ciphertext of *m*.

+

An example of the tree structure is depicted in Fig. 6. For each node, we assign 0 to its left child path, while 1 to the right child path. If an integer is not greater than the value of

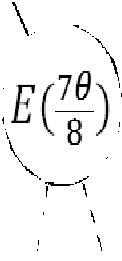
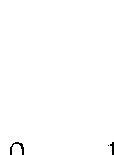
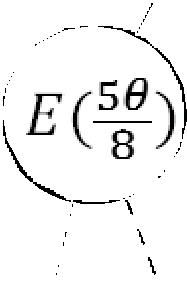
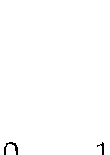
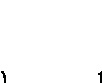
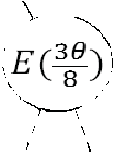
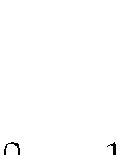
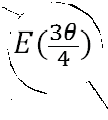
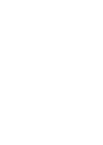
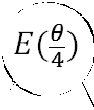
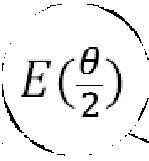
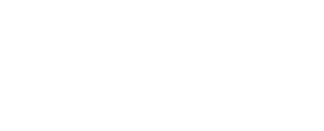
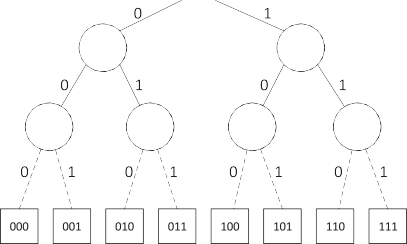


Fig. 6. An example of the cost constraint tree with a depth of 3, where circles represent nodes. The boxes in the dashed rectangle indicate path codes for all possible comparison results. Note that these boxes are not a part of the tree.

this node, we take the left child path for further comparison; otherwise, we take the right child path. Thus, for any path from the root node to a leaf node, we can obtain a path code, which is an effective representation of the comparison procedure. For instance, an incoming integer 5*θ/*16 would traverse Nodes *E(θ/*2*)*, *E(θ/*4*)*, and *E(*3*θ/*8*)*, and thereby end with a path code of 010. We define the length (i.e., the number of bits) of a path code as *β*. Note that *β* is actually equal to the depth of the tree which is denoted by *dθ* .

Now the relationship between *x y* and *θ* can be determined as follows. We first get the ORE ciphertexts of *x* and *y*, as well as their path codes *cx* and *cy* by traversing the tree separately. When computing *cx cy*, if an overflow occurs (i.e., *cx cy* 2*β* ), we know that *x y >θ* with confidence. If *cx cy* 2*β* 2, we also know that *x y θ* with confidence. Otherwise, we are unable to determine the relationship and end up with an *uncertainty*. We summarize this procedure in Algorithm 4.

+ ≤ − + ≤

+ ≥ +

+

+



**Algorithm 4** Tree-Based Ciphertexts Comparison Algorithm



**Input:** Two ORE ciphertexts *E(x)*, *E(y)* and a cost constraint tree whose depth is *dθ* .

**Output:** The relationship between *x y* and *θ* .

+

1: Initialize a counter *ω* 1 and two empty strings *cx* and

=

*cy*.

2: **while** *ω dθ* **do**

≤

3: Visit the *ω*-th level of the tree with *E(x)* and concatenate

*cx* with corresponding 0 or 1.

4: Visit the *ω*-th level of the tree with *E(y)* and concatenate

*cy* with corresponding 0 or 1.

5: Set *ω ω* 1*.*

= +

6: **end while**

7: **if** *cx cy* 2*ω* **then**

+ ≥

8: **return** *>*.

9: **end if**

10: **if** *cx cy* 2*ω* 2 **then**

+ ≤ −

11: **return** .

≤

12: **end if**

13: **return** *uncertainty*.



*Discussion:* Observe that when we go through a cost con- straint tree, one more step can further reduce the uncertainty of the relationship between *x* + *y* and *θ* by half. We denote

the probability of *uncertainty* as

1 *β*

*Pr* [¬*certainty*]= *(* 2 *) .*

* 1. *Setup Leakage:* The leakage function *Setup* of our construction reveals the information that can be deduced from the secure 2HCLI *6* of graph *G*, including the total number of vertices in the graph *n*, the maximum distance over all the

where *β* is the length of the path code. We can easily know

*L*

the probability of certainty is

sketches

*B* = *maxu*∈*V max*{*(v,du,v ,cu,v )*∈*6out,(v,du,v ,cu,v )*∈*6in* }*du,v,*

1 *β*

˜

*Pr* [*certainty*]= 1 − *Pr* [¬*certainty*]= 1 − *(* 2 *) .*

and the size of *6*˜. More precisely, the size of *6*˜ consists

When the tree depth is 6 (e.g., *β* 6), the probability of certainty could reach about 0.9844.

=

*L* = *B*

Another observation is the comparison procedure reveals the order information between *x* (or *y*) and *θ* . Thus, the *server*

can infer the interval that *x* belongs to with precision of 2−*β* . To prevent the *server* from inferring the real value of *x* , in Connor, the *user* randomly picks a big integer

number *φ* that is applied to *x* , *y*, and *θ* simultaneously, which significantly enlarges the plaintext and ciphertext spaces (e.g., 2128). The value of *β* is generally a small integer (e.g., 6 in our implementation) that is determined by the *user*, and both *φ* and *θ* are kept secret by the *user*. Therefore, the *server* cannot infer the real value of *x* (or *y*) from the order relationship among ciphertexts. We will formally analyze the leakage functions and security issues in the next section.

* 1. COMPLEXITY AND SECURITY ANALYSES

This section presents the complexity and security analyses on the proposed graph encryption scheme Connor.

1. *Complexity Analysis*

Connor mainly consists of the *Setup* and *Query* algorithms, as described in Algorithms 2 and 3.

The dominant component in determining the complexity of the *Setup* algorithm is the encryption of the plain 2HCLI generated from a graph *G*. Let *μ* be the total sketch for all vertices in *G*, then the time complexity and space complexity are both *(nμ)*, where *n* is the number of vertices in *G*.

*O*

The *Query* algorithm consists of a query token generation process on the *user* side and a CSD query process on the cloud *server* side. Let *η* be the maximum size of the sketch associ- ated with each vertex in *G*. The complexity of the query token generation process is mainly determined by the construction of a cost constraint tree, whose time complexity and space complexity are both *(*2*dθ )*. For the CSD querying process, the time complexity of getting *Ls* and *Lt* , performing cost constraint filtering, and performing distance computation are *(η)*, *(ηdθ )*, and *(η)*, respectively. The space complexity of the above three components are *(η)*, *(η* 2*dθ )*, and *(η)*, respectively. Therefore, the total time complexity and space complexity of the CSD querying process are *O(ηdθ )* and

*O*

*O O* + *O*

*O O O*

*O(η* + 2*dθ )*, respectively.

1. *Security Analysis*

We now present the security analysis on Connor. For clarity, we first discuss the leakage functions, and then prove that Connor is secure under the CQA2-security model.

of the total number of sketch entities in *Iout* and *Iin* , which are denoted by *Kout* and *Kin*, respectively. Thus, the leakage function *Setup (n, , Kout, Kin )*.

Note that the order relationship of pairwise costs and the order relationship between the cost and cost constraint are not included in *Setup*, because for each entity in sketches, we make an XOR operation using a unique integer value after we encrypt it, and this makes each entity in sketches are indistinguishable.

*L*

* 1. *Query Leakage:* The leakage function *Quer y* of our con- struction consists of the query pattern leakage, the sketch pat- tern leakage, and the cost pattern leakage. Intuitively, the query pattern leakage reveals whether a query has appeared before. The sketch pattern leakage reveals the sketch associated to a queried vertex, the common vertices between two different sketches, and the size of the sketches of queried vertices. The cost pattern leakage reveals 1) the order relationship among costs, and 2) the order relationship between costs and the cost constraint during the query procedure. We formalize these leakage functions as follows.

*L*

*Definition 4 (QUERY PATTERN LEAKAGE): Let* **q** *(q*1*, q*2*, ... , qm) be a non-empty sequence of queries. Each query qi specifies a tuple (ui, vi , θi). For any two queries qi and q j , define Sim(qi, q j ) (ui u j , vi v j , θi θ j ), i.e., whether each element of qi (ui, vi , θi) matches each element of q j (u j ,v j ,θj ), respectively. Then, the query pattern leakage function QP (***q***) returns an m m (symmetric) matrix, in which each entry (i , j ) equals Sim(qi, q j ). Note that*

=

*L* ×

=

=

= = = =

*QP(***q***) does not leak the identities of the query vertices.*

*L*

*Definition 5 (SKETCH PATTERN LEAKAGE): Given a secure 2HCLI 6 of a graph G and a query q (u,v,θ), the sketch pattern leakage function SP (6, q) is defined as (Σ, ϒ). Σ is a list, each element of which is the sketches asso- ciated to the queried vertices, and ϒ is a pair (X, Z), where X h(v) (v, d, c) Iout and Z h(v) (v, d, c) Iin are*

˜

*L* ˜

=

= : ∈ = : ∈

*multi-sets and h* 0*,* 1 *λ* 0*,* 1 ∗ 0*,* 1 *λ is a particular*

: { } × { } → { }

*pseudo-random function.*

*Definition 6 (COST PATTERN LEAKAGE): The cost con- straint θ in a query q can essentially be represented by a certain number of uniform intervals. Let dθ be the depth of the cost constraint tree Tθ (c.f. Section VI). The intervals asso- ciated with θ are (i* 1*)θ/*2*dθ , iθ/*2*dθ , where* 1 *i* 2*dθ . Assign each interval with a list μ, i.e., the i-th interval is associated with μi , which stores all the cost values belong to this interval. The leaked interval information forms an array Arr, of which the i -th element is μi (i.e., Arr i μi ). In addition, assume that z is the total number of entries in the sketches of the queried vertices. For each pair of costs*

[ ] =

[ − ] ≤ ≤

*ci and c j , its order relationship of the greater-than, equality, and less-than can be represented by* 1*,* 0*, and* −1*, respectively.*

*The leaked order information of costs is a z* × *z (symmetric)*

previously, *S* sets the *Tθ*∗ to the value that was previously

*matrix* ∇ *with each entry (i , j ) being* 1*,* 0*, or* −1*. Therefore, the cost pattern leakage function LCP (6, q)* = *( Arr,* ∇*).*

˜

*L* = *L L L*˜ ˜

*S*

Thus, *Quer y ( QP (****q****), SP (6, q), CP (6, q))*.

The leakage functions are defined over the 2HCLI rather than the original graph. In fact, the information leakage of the original graph is limited to the minimum number of paths for the queried source-destination vertices. It can be defined as an *n n* (symmetric) matrix *K*, where *n* is the number of vertices in the graph. Each element in *K* is NULL, 0, or a positive integer, which indicates an uncertain status (i.e., topology is well protected), disconnection, or the minimum number of paths of the two queried vertices, respectively.

×

For the cost values in the 2HCLI, we introduce a *user*-held amplification factor *φ* to enlarge the plaintext and ciphertext spaces. Thus, the *server* cannot infer the real cost values just from their order information revealed by the leakage function *CP (6, q))*. For the distance values in the 2HCLI, we use the SWHE encryption to protect their real values from the *server*. *Theorem 1: If the cryptography primitives g, h, ORE, and the SWHE are secure, then the proposed graph encryption scheme M (Key Gen, Setup, Query) is ( Setup, Quer y)-*

*L* ˜

= *L L*

*secure against the adaptive chosen-query attack.*

*Proof:* The key idea is constructing a simulator *S*. Given the leakage functions *LSetup* and *LQuer y*, *S* constructs a fake

used. Otherwise, constructs a full binary tree based *θ* and encrypts each tree node by using the ORE scheme with a randomly generated key. returns this encrypted tree as *Tθ*∗.

simulates the query procedure as follows. Given the query token *(So*∗*ut,s, To*∗*ut,s, Si*∗*n,t , Ti*∗*n,t , Tθ*∗*)*, first checks if the query has been queried before. If yes, returns the value that

*S*

*S*

*S*

*S*

was previously used as the query result. Otherwise, checks whether the queried vertex *s* (or *t*) has been queried before. If the query vertex *s* has appeared in a previous query,

*S*

*S*

sets *L*∗*s* to the values that were previously used from *Σ* of

*L* ˜ *S*

*SP( f , q)*. Otherwise, for a newly appeared vertex *s*, takes the following steps: To generate the sketches associated with *s*, *S* first initializes a set *L*∗*s* and a counter *ω*∗ = 0, Then,

it iteratively computes *To*∗*ut,s,v* = *h(To*∗*ut,s, w*∗*)* and *So*∗*ut,s,v* = *g(So*∗*ut,s, w*∗*)*, and adds the tuple *(V* ∗*, Ds*∗*,v , Cs*∗*,v )* into *L*∗*s* , until *Io*∗*ut To*∗*ut,s,v* does not exist, where *(V* ∗*, Ds*∗*,v , Cs*∗*,v ) Io*∗*ut To*∗*ut,s,v So*∗*ut,s,v* . Similarly, obtains the set *L*∗*t* for vertex *t*. Upon obtaining *L*∗*s* and *L*∗*t* , performs cost constraint filtering operation based on *Tθ*∗ to get the candidate set *Y* ∗. The theorem then follows from the CPA-security of SWHE. That is, performs the SWHE computation over *Y* ∗ and returns

*S*

*S*

[ ]⊕ *S*

[ ] =

the query result.

Since the cryptography primitives *g*, *h*, ORE, and SWHE are secure, the fake 2HCLI structure *6*˜∗ and the query

encrypted 2HCLI structure *6*˜∗ = {*Io*∗*ut , Ii*∗*n* } and a list of

query *q*∗. If for all PPT adversaries , they cannot distinguish between the two games **Real** and **Ideal**, we can say that our

*A*

sequence *q*∗ are indistinguishable from the real ones. There-

fore, for all PPT adversaries , they cannot distinguish between the two games **Real** and **Ideal**. Thus, we have

*A*

*A*

graph encryption scheme is *( Setup, Quer y)*-secure against the adaptive chosen-query attack.

*L L*

|**Pr**[ *Real*

*M,A*

*(λ)* = 1]− **Pr**[*Ideal*

*M,A,S*

*(λ)* = 1]| ≤ *negl(λ).*

* 1. *Simulating 6*∗*: S* handles each vertex *ui* (1 ≤ *i* ≤ *n*) to generate a fake *Iout* in 2HCLI based on the leakage function

˜∗

*LSetup*. *S* randomly chooses *wi* for *ui* with Σ *wi* = *Kout*,

*n*

1

and samples *li* 0*,* 1 *λ* and *ηi* 0*,* 1 *λ* uniformly without

← { } ← { }

repetition. For all 0 *i < wi* , takes the following steps to simulate each sketch: computes *lwi h(li , wi )* and *ηwi h(ηi , wi)*, where *h* is a particular pseudo-random function. Then, it encrypts each vertex *v* in the sketch of

=

*S* =

≤ *S*

*ui* by computing *V* ∗ *h(K* ∗*,v* 0*)*, where *K* ∗ is a fake

= ||

secret key. It randomly generates two integers *d* and *c* and obtains ciphertexts *D*∗ and *C*∗ by encrypting 2*N* −*d* (*N*

*B* +

=

2 1) and *c* using the SWHE and ORE schemes. Let *Ti ηwi (V* ∗ *D*∗ *C*∗*)*. stores *T*∗ in the index *Io*∗*ut* . That is, *Io*∗*ut lwi Ti*∗. Similarly, generates a fake *Ii*∗*n* and finally obtains the fake 2HCLI *6*∗ *Io*∗*ut , Ii*∗*n* .

˜

∗

*L* =

= { }

[ ]= *S*

= ⊕ || || *S*

Simulating *q*∗. Given the leakage function *Quer y*

˜ ˜

*( QP (q), SP (6, q), CP (6, q))*, simulates the query token as follows. first checks if either of the queried vertices

*S*

*L L L S*

*s* and *t* has appeared in any previous query. If *s* appeared previously, sets *So*∗*ut,s* and *To*∗*ut,s* to the values that were previously used. Otherwise, it sets *To*∗*ut,s li* and *So*∗*ut,s ηi* for some previously unused *li* and *ηi* . It then remembers the

= =

*S*

association among *ηi* , *li* , and *s*. takes the same steps for the queried vertex *t*: setting *Si*∗*n,t* and *Ti*∗*n,t* analogously and associating *t* with the selected *ηi* and *li* .

*S*

To simulate a fake cost constraint tree *Tθ*∗, first checks if the queried *θ* appeared in any previous query. If *θ* appeared

*S*

where *negl(λ)* is a negligible function. Q

* 1. PERFORMANCE EVALUATION

This section presents the evaluation of our graph encryption scheme through experiments on real-world datasets.

1. *Setup*
   1. *Testbed:* We implement the method introduced in [2] for building the 2HCLI. The ORE and SWHE in our imple- mentation follow the methods described in [15] and [16], respectively. The GMP library is used for big integer arith- metic. We set the security parameter *λ* 128 and use the OpenSSL library for all the basic cryptographic primitives. All the algorithms in our experiment are implemented in C . The experiments are conducted on a desktop PC equipped with Intel Xeon processor at 2.6 GHz and 8 GB RAM.

=

++

* 1. *Graph Sets:* The datesets used in our experiments are listed in Table II. All these datasets are publicly available from the Standford SNAP website3 and modeled as directed graphs. For the datasets soc-Epinions1 and Email-EuAll, we randomly select their subsets to make the index construction feasible with the limited computational resources. Since these graphs are unweighted, we generate a distance and a cost for each edge, the value of which follows a uniform distribution between 1 and 100. The cost criterion is used as the constraint.

3<http://snap.stanford.edu/data/>