An Efficient Query Recovery Attack Against a Graph Encryption Scheme

An Efficient Query Recovery Attack Against a Graph Encryption Scheme

Francesca Falzon[1,2] and Kenneth G. Paterson[3]

[1] Brown University, Providence, RI, USA
[2] University of Chicago, Chicago, IL, USA
ffalzon@uchicago.edu
[3] ETH Zürich, Zürich, Switzerland
kenny.paterson@inf.ethz.ch

Abstract. Ghosh, Kamara and Tamassia (ASIA CCS 2021) presented a Graph Encryption Scheme supporting shortest path queries. We show how to perform a query recovery attack against this GKT scheme when the adversary is given the original graph together with the leakage of certain subsets of queries. Our attack falls within the security model used by Ghosh et al., and is the first targeting schemes supporting shortest path queries. Our attack uses classical graph algorithms to compute the canonical names of the single-destination shortest path spanning trees of the underlying graph and uses these canonical names to pre-compute the set of candidate queries that match each response. Then, when all shortest path queries to a single node have been observed, the canonical names for the corresponding query tree are computed and the responses are matched to the candidate queries from the offline phase. The output is guaranteed to contain the correct query. For a graph on n vertices, our attack runs in time $O(n^3)$ and matches the time complexity of the GKT scheme's setup. We evaluate the attack's performance using the real world datasets used in the original paper and show that as many as 21.9% of the queries can be uniquely recovered and as many as 50% of the queries result in sets of only three candidates.

Keywords: Encrypted databases · Leakage-abuse attacks · Cryptanalysis

1 Introduction

Graphs are a powerful tool that can be used to model many problems related to social networks, biological networks, geographic relationships, etc. Plaintext graph database systems have already received much attention in both industry (e.g. Amazon Neptune [2], Facebook TAO [23], Neo4j [18]) and academia (e.g. GraphLab [15], Trinity [22]). With the rise of data storage outsourcing, there is an increased interest in Graph Encryption Schemes (GES). A GES enables a client to encrypt a graph, outsource the storage of the encrypted graph to an untrusted server, and finally to make certain types of graph queries to the server. Current GES typically support one type of query, e.g. adjacency queries [6], approximate shortest distance queries [16], and exact shortest path queries [12,24].

In this paper, we analyse the security of the GES of Ghosh, Kamara and Tamassia [12] from ASIA CCS 2021. We refer to this scheme henceforth as the ***GKT scheme***. The GKT scheme encrypts a graph G such that when a shortest path query (u, v) is issued for some vertices u and v of G, the server returns information allowing the client to quickly recover the shortest path between u and v in G. The scheme pre-computes a matrix called the ***SP-matrix*** from which shortest paths can be efficiently computed, and then creates an encrypted version of this matrix which we refer to as the encrypted database (EDB). EDB is sent to the server. At query time, the client computes a search token for the query (u, v); this token is sent to the server and is used to start a sequence of look-ups to EDB. Each look-up results in a new token and a ciphertext encrypting the next vertex on the shortest path from u to v. The concatenation of these ciphertexts is returned to the client.

The GKT scheme [12] is very elegant and efficient. For a graph on n vertices, computing the SP-matrix takes time $O(n^3)$ and dominates the setup time. Building a search token involves computing a pseudo-random function. Processing a query (u, v) at the server requires t look-ups in EDB, where t is the length of the shortest path from u to v. Importantly, thanks to the design of the scheme, query processing can be done without interaction with the client, except to receive the initial search token and to return the result. This results in EDB revealing at query time the sequence of labels (tokens) needed for the recursive lookup and the sequence of (encrypted) vertices that is eventually returned to the client.

We exploit the query leakage of the GKT scheme to mount a ***query recovery (QR)*** attack against the scheme. Our attack can be mounted by the honest-but-curious server and requires knowledge of the graph G. This may appear to be a strong requirement, but is in fact weaker than is permitted in the security model of [12], where the adversary can even *choose* G. Assuming that the graph G is public is a standard assumption for many schemes that support private graph queries [12,17,21]. This model is perfect for routing and navigation systems in which the road network may easily be obtained online via Google Maps or Waze, but the client may wish to keep its queries private.

Our attack has two phases. First, it has an offline, pre-processing phase that is carried out on the graph G. In this phase, we extract from G a plaintext description of all its shortest path trees. We then process these trees and compute candidate queries for each query using each tree's canonical labels. A canonical label is an encoding of a graph that can be used to decide when graphs are isomorphic; a canonical label of a rooted tree can be computed efficiently using the AHU algorithm [1]. This concludes the offline phase of the attack. Its time complexity is $O(n^3)$ where n is the number of vertices in G, and matches the run time of our overall attack and the run time of the GKT scheme's setup. Both our attack and the setup are lower bounded by the time to compute the all-pairs shortest paths, which takes $O(n^3)$ time for general graphs [11].

The second phase of the attack is online: As queries are issued, the adversary constructs a second set of trees that correspond to the sequence of labels computed by the server when processing each query i.e. the per-query leakage of the scheme.

This description uses the labels of EDB (which are search tokens) as vertices; two labels are connected if the first points to the second in EDB. When an entire tree has been constructed, the adversary can then run the AHU algorithm again to compute the canonical names associated with this ***query tree***. An entire query tree Q can be built when all queries to a particular destination have been issued. This is realistic in a routing scenario where many trips may share a common popular destination (e.g. an airport, school, or distribution center).

By correctness of the scheme, there exists a collection of isomorphisms mapping Q to at least one tree computed in the offline phase. Such isomorphisms also map shortest paths to shortest paths. We thus perform a matching between the paths in the trees from the online phase to the trees in the offline phase. This can be done efficiently using a modified AHU algorithm [1] that we develop and which decides when one path can be mapped to another by an isomorphism of trees. This yields two look-up tables which, when composed, map each path in the first set of trees to a set of candidate paths in the second set. We use the search token of the queries associated with Q to look up the possible candidate queries in the tables computed in the online phase, and output them. The output is guaranteed to contain the correct query.

In general, the leakage from a query can be consistent with many candidates, and the correct candidate cannot be uniquely determined. Graph theoretically, this is because there can be many possible isomorphisms between pairs of trees in our two sets. If we consider the chosen graph setting, it is easy to construct a graph G where, given any query tree Q of G, its isomorphism is uniquely determined and there is a unique candidate for each query of Q, i.e. we can achieve what we call ***full query recovery (FQR)***. In other cases, the query leakage may result in one or only a few possible query candidates, which may be damaging in practice. In order to explore the effectiveness of our attack, we support it with experiments on 8 real-world graphs (6 of which were used in [12]) and on random graphs. Our results show that for the given real-world graphs, as many as 21.9 % of all queries can be uniquely recovered and as many as half of all queries can be mapped to at most 3 candidate queries. Our experimental results show that QR tends to result in smaller sets of candidate queries when the graphs are less dense, and that dense graphs tend to have more symmetries.

We summarize our core contributions as follows:

1. We present the first attack against a GES that supports shortest path queries, and the second known attack against GESs, to our knowledge.
2. We use the GKT scheme's leakage to mount an efficient query recovery attack against the scheme. We explain how, for our real world datasets, the set of all query trees can be recovered with as few as 68.1% of the queries.
3. We make use of the classical AHU algorithm for the graph isomorphism problem for rooted trees and develop a new algorithm for deciding when a path in one tree can be mapped onto a path in another under an isomorphism.
4. We evaluate our attack against real-world datasets and random graphs.
5. We motivate the need for detailed cryptanalysis of GESs.

All proofs can be found in the full version [10].

1.1 Prior and Related Work

Chase and Kamara present the first graph encryption scheme that supports both adjacency queries and focused subgraph queries [6]. Poh et al. give a scheme for encrypting conceptual graphs [19]. Meng et al. present three schemes that support approximate shortest path queries on encrypted graphs, each with a slightly different leakage profile [16]. To reduce storage overhead, their solution leverages sketch-based oracles that select seed vertices and store the exact shortest distance from all vertices to the seeds; these distances are then used to estimate shortest paths in the graph. Ghosh et al. [12] and Wang et al. [24] present schemes that support exact shortest path queries on encrypted graphs. Other solutions for privacy preserving graph structures use other techniques (e.g. [20,25]), however, these approaches have different security goals.

The leakage of GESs was first analyzed by Goetschmann [13]. The author considers schemes that support approximate shortest path queries that use sketch-based distance oracles (e.g. [16]), presents two methods for estimating distances between nodes, and gives a query recovery attack that aims to recover the vertices in an encrypted query.

2 Preliminaries

Notation. For some integer n, let $[n] = \{1, 2, \ldots, n\}$. We denote concatenation of two strings a and b as $a||b$.

Graphs. A ***graph*** is a pair $G = (V, E)$ consisting of a vertex set V of size n and an edge set E of size m. A graph is ***directed*** if the edges specify a direction from one vertex to another. Two vertices $u, v \in V$ are ***connected*** if there exists a path from u to v in G. In this paper, we assume that all graphs G are connected for simplicity of presentation. However our attack and its constituent algorithms directly apply to multi-component graphs too.

A ***tree*** is a connected, acyclic graph. A ***rooted tree*** $T = (V, E, r)$ is a tree in which one vertex r has been designated the root. For some rooted tree $T = (V, E, r)$ and vertex $v \in V$ we denote by $T[v]$ the subtree of T induced by v and all its descendants.

Given a graph $G = (V, E)$ and some vertex $v \in V$, we define a ***single-destination shortest path (SDSP) tree*** for v to be a directed spanning tree T such that T is a subgraph of G, v is the only sink in T, and each path from $u \in V \setminus \{v\}$ to v in T is a shortest path from u to v in G. An example of an SDSP tree can be found in Figure 1b.

We also define two binary options on graphs. Given two graphs $G = (V, E)$ and $H = (V', E')$, the union of G and H is defined as $G \cup H = (V \cup V', E \cup E')$. Given a graph $G = (V, E)$ and a subgraph $H = (V', E')$ such that $V' \subseteq V, E' \subseteq E$, the graph subtraction of H from G is defined as $G \setminus H = (V \setminus V', E \setminus E')$.

Dictionaries. A ***dictionary*** D is a map from some label space $\mathbb{L}$ to a value space $\mathbb{V}$. If $\mathsf{lab} \mapsto \mathsf{val}$, then we write $\mathsf{D}[\mathsf{lab}] = \mathsf{val}$.

Hash functions. A set H of functions $U \to [M]$ is a ***universal hash function family*** if, for every distinct $x, y \in U$ the hash function family H satisfies the following constraint: $\Pr_{h \leftarrow H}[h(x) = h(y)] \leq 1/M$.

2.1 Graph Isomorphisms

Our approach will make heavy use of graph isomorphisms.

Definition 1. *An **isomorphism of graphs** $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is a bijection between vertex sets $\varphi : V_1 \to V_2$ such that for all $u, v \in V_1, (u, v) \in E_1$ if and only if $(\varphi(u), \varphi(v)) \in E_2$. If such an isomorphism exists, we write $G_1 \cong G_2$.*

Definition 2. *An **isomorphism of rooted trees** $T_1 = (V_1, E_1, r_1)$ and $T_2 = (V_2, E_2, r_2)$ is an isomorphism φ from T_1 to T_2 (as graphs) such that $\varphi(r_1) = r_2$.*

2.2 Canonical Names

A ***canonical name*** $\mathsf{Name}(\cdot)$ is an encoding mapping graphs to bit-strings such that, for any two graphs H and G, $\mathsf{Name}(G) = \mathsf{Name}(H)$ if and only if $G \cong H$. For rooted trees Aho, Hopcraft, and Ullman (AHU) [1] describe an algorithm for computing a specific canonical name in $O(n)$ time. We refer to this as *the* canonical name and describe it next.

The AHU Algorithm. We use a modified AHU algorithm, which we denote as ComputeNames, to compute the canonical names of rooted trees (and their subtrees) and determine if they are isomorphic. ComputeNames takes as input a rooted tree $T = (V, E, r)$, a vertex $v \in V$, and an empty dictionary Names. It outputs the canonical name of the subtree $T[v]$ (which we also refer to as the canonical name of v) and a dictionary Names that maps each descendent u of v to the canonical name of $T[u]$. The algorithm proceeds from the leaves to the root. It assigns the name '10' to all leaves of the tree. It then recursively visits each descendent u of v and assigns u a name by sorting the names of its children in increasing lexicographic order, concatenating them into an intermediate name $children\_names$ and assigning the name '$1||children\_names||0$' to u (see Figure 1b for an example). The canonical name of T, $\mathsf{Name}(T)$, is the name assigned to the root r. Pseudocode for the AHU algorithm can be found in the full version [10].

2.3 Threat Model and Assumptions

We consider a ***passive, persistent, honest-but-curious*** adversary that has compromised the server and can observe the initial search tokens issued, all subsequent search tokens revealed during query processing, and the responses.

We assume that the adversary knows the graph G that has been encrypted to create EDB. This is a strong assumption, but fits within the security model used in [12] (where G can even be chosen) and is realistic in many navigation

scenarios. We further assume that the adversary sees enough queries to construct at least one query tree. We emphasize that computing a complete query tree does *not* require observing all possible queries to the root; observing just the queries starting at the leaf nodes of the tree is sufficient for constructing a query tree. In SDSP trees with few leaves, only a small fraction of queries is thus needed. We assume that the all-pairs shortest path algorithm used in constructing the SP-matrix from G during setup is deterministic and that it is known to the adversary. Such an assumption is reasonable since many shortest path algorithms are deterministic, including Floyd-Warshall [11].

3 The GKT Graph Encryption Scheme

3.1 GKT Scheme Overview

The GKT scheme supports ***single pair shortest path (SPSP)*** queries. The graphs may be directed or undirected, and the edges may be weighted or unweighted. An SPSP query on a graph $G = (V, E)$ takes as input a pair of vertices $(u, v) \in V \times V$, and outputs a path $p_{u,v} = (u, w_1, \ldots, w_\ell, v)$ such that $(u, w_1), (w_1, w_2), \ldots, (w_{t-1}, v) \in E$ and $p_{u,v}$ is of minimal length.

SPSP queries may be answered using a number of different data structures. The GKT scheme makes use of the ***SP-matrix*** [7]. For a graph $G = (V, E)$, the SP-matrix M is a $|V| \times |V|$ matrix defined as follows. Entry $M[i, j]$ stores the second vertex along the shortest path from vertex v_i to v_j; if no such path exists, then it stores $\perp$. An SPSP query (v_i, v_j) is answered by computing $M[i, j] = v_k$ to obtain the next vertex along the path and then recursing on (v_k, v_j) until $\perp$ is returned. At a high level, the GKT scheme proceeds by computing an SP-matrix for the query graph and then using this matrix to compute a dictionary SPDX'. This dictionary is then encrypted using a dictionary encryption scheme (DES) such as [4,6]. To ensure that the GKT scheme is non-interactive, the underlying DES must be response-revealing. We provide the syntax of a DES next.

Definition 3. *A **dictionary encryption scheme (DES)** is a tuple of algorithms* $\mathsf{DES} = (\mathsf{Gen}, \mathsf{Encrypt}, \mathsf{Token}, \mathsf{Get})$ *with the following syntax:*

- $\mathsf{DES.Gen}$ *is probabilistic and takes as input security parameter* λ*, and outputs secret key* sk*.*
- $\mathsf{DES.Encrypt}$ *takes as input key* sk *and dictionary* D*, and outputs an encrypted dictionary* ED*.*
- $\mathsf{DES.Token}$ *takes as input key* sk *and label* lab*, and outputs a search token* tk*.*
- $\mathsf{DES.Get}$ *takes as input search token* tk *and encrypted dictionary* ED*, and returns plaintext value* val*.*

While the GKT scheme is response-hiding (i.e. the shortest path is not returned in plaintext to the client), the underlying DES must be response-revealing (i.e. the values in its encrypted dictionary ED are revealed at query time). We provide a detailed description of the GKT scheme in Appendix A.

3.2 Leakage of the GKT Scheme

Ghosh et al. [12] provide a formal specification of their scheme's leakage. Informally, the setup leakage of their scheme is the number of vertex pairs in G that are connected by a path, while the query leakage consists of the query pattern (which pairs of queries are equal), the path intersection pattern (the overlap between pairs of shortest paths seen in queries), and the lengths of the shortest paths arising in queries. See [12, Section 4.1] for more details.

Recall that in the GKT scheme, the server obtains EDB by encrypting the underlying dictionary SPDX', in which labels are of the form $\mathsf{lab} = (v_i, v_j)$ and values are of the form $\mathsf{val} = (\mathsf{tk}, c)$, using a DES. Here tk is a token obtained by running $\mathsf{DES.Token}$ on a pair (w, v_j) and c is obtained by symmetrically encrypting (w, v_j). Since EDB is obtained by running DES on SPDX', this means that the labels in EDB are derived from tokens obtained by running $\mathsf{DES.Token}$ on inputs $\mathsf{lab} = (v_i, v_j)$. Moreover, these tokens also appear in the values in EDB that are revealed to the server at query time, i.e. the entries (tk, c).

In turn, the query leakage reveals to the server the token used to initiate a search, as well as all the subsequent pairs (tk, c) that are obtained by recursively processing such a query. Let us denote the sequence of search tokens associated with the processing of some (unknown) query q for a shortest path of length t as $s = \mathsf{tk}_1 \| \mathsf{tk}_2 \| \ldots \| \mathsf{tk}_{t+1} \in \{0,1\}^*$. We refer to this string as the ***token sequence of*** q. Since the search tokens correspond to the sequence of vertices in the queried path, there are as many tokens in the sequence as there are vertices in the shortest path for the query. Note that, by correctness of DES used in the construction of EDB, no two distinct queries can result in the same token sequence.

Notice also that token sequences for different queries can be overlapping; indeed since the tokens are computed by running $\mathsf{DES.Token}$ on inputs $\mathsf{lab} = (v_i, v)$ where v is the final vertex of a shortest path, two token sequences are overlapping if and only if they correspond to queries (and shortest paths) having the same end vertex. Hence, given the query leakage of a set of queries, the adversary can compute all the token sequences and construct from them $n' \leq n$ directed trees, $\{Q_i\}_{i \in [n']}$, each tree having at most n vertices and a single root vertex. The vertices across all n' trees are labelled with the search tokens in EDB and there is a directed edge from tk to tk' if and only if tk and tk' are adjacent in some token sequence. (Each tree has at most n vertices because of our assumption about G being connected.) We call this set of trees the ***query trees***. Each query tree corresponds to the set of queries having the same end vertex. Each tree has a single sink (root) that corresponds to a unique vertex $v \in V$. The tree paths correspond to the shortest paths from vertices $w \in V \setminus \{v\}$ to v, such that w and v are connected in G. Ghosh et al. [12] also discuss these trees but they do not analyze the theoretical limits of what can be inferred from them.

We denote the leakage of the GKT scheme on a graph G after issuing a set of SPSP queries $\mathcal{Q}$ as $\mathcal{L}(G, \mathcal{Q})$. We stress that our attacks are based only on the leakage of the scheme, as established above, and not on breaking the underlying cryptographic primitives of the scheme.

3.3 Implications of Leakage

Suppose that all queries have been issued and that we have constructed all query trees $\{Q_i\}_{i\in[n]}$. We observe that there exists a 1-1 matching between the query trees $\{Q_i\}_{i\in[n]}$ and the SDSP trees $\{T_v\}_{v\in V}$ of G such that each matched pair of trees is isomorphic. The reason is that the query trees are just differently labelled versions of the SDSP trees; in turn, this stems from the fact that paths in the query trees are in 1-1 correspondence with the shortest paths in G.

This now reveals the core of our query recovery attack, developed in detail in Section 4 below. The server with access to G first computes all the SDSP trees offline. As queries are issued, it then constructs the query trees one path at a time. Once a complete query tree Q is computed (recall that each query tree must have n vertices since G is connected) the server finds all possible isomorphisms between Q and the SDSP trees. Then, for each token sequence in Q, it computes the set of paths in the SDSP trees to which that token sequence can be mapped under the possible isomorphisms. This set of paths yields the set of possible queries to which the token sequence can correspond. This information is stored in a pair of dictionaries, which can be used to look up the candidate queries.

To illustrate the core attack idea, Figure 1 depicts (1a) a graph G, (1b) its SDSP tree for vertex 1 (with vertex labels and canonical names), and (1c) the matching query tree (without vertex labels). It is then clear that the leakage from the unique shortest path of length 2 in Figure (1c) can only be mapped to the corresponding path with edges $(4, 5)$, $(5, 1)$ in Figure (1b) under isomorphisms, and similarly the shortest path of length 1 that is a subpath of that path of length 2 can only be mapped to path $(5, 1)$. On the other hand, the 3 remaining paths of length 1 can be mapped under isomorphisms to *any* of the length 1 paths $(2, 1)$, $(3, 1)$, or $(6, 1)$ and so cannot be uniquely recovered.

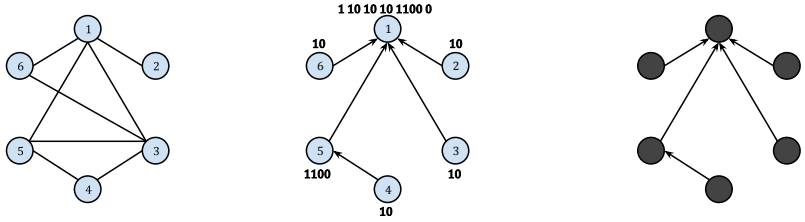
Since the adversary only learns the query trees and token sequences from the leakage, the degree of query recovery that can be achieved based on that leakage is limited. In Section 5, we show that in practice this is often not an issue since many queries result in only a very small number of candidate queries.

4 Query Recovery

4.1 Formalising Query Recovery Attacks

Query recovery in general is the goal of determining the plaintext value of queries that have been issued by the client. We study the problem of query recovery in the context of GESs, specifically, the GKT scheme: given G, the setup leakage and the query leakage from a set of SPSP queries, our goal is to match the leakage for each SPSP query with the corresponding start and end vertices (u, v) of a path in G. As noted above, there may be a number of candidate queries that can be assigned to the leakage from each query. We now formally describe the adversary's goals.

Definition 4. *(Consistency) Let $G = (V, E)$ be a graph, $\mathcal{Q} = \{q_1, \ldots, q_k\}$ be the set of SPSP queries that are issued, and $S = \{s_1, s_2, \ldots, s_k\}$ be the set of token*



(a) Graph G. (b) SDSP tree T_1. (c) Leakage.

Fig. 1: (a) Original graph G, (b) its corresponding SDSP tree for vertex 1 in G with the canonical names labeling all the vertices of the tree, and (c) the matching query tree that is leaked during setup (without any vertex labels).

sequences of the queries issued. An assignment $\pi : S \to V \times V$ is a mapping from token sequences to SPSP queries. An assignment π is said to be ***consistent with the leakage*** *$\mathcal{L}(G, \mathcal{Q})$ if it satisfies $\mathcal{L}(G, \mathcal{Q}) = \mathcal{L}(G, \pi(S))$.*

Informally, consistency requires that, for each $s_i \in S$, the query $\pi(s_i)$ specified by assignment π *could* feasibly result in the observed leakage $\mathcal{L}(G, \mathcal{Q})$.

Definition 5. *(QR) Let $G = (V, E)$ be a graph, $\mathcal{Q} = \{q_1, \ldots, q_k\}$ be a set of SPSP queries, and S the corresponding set of token sequences. Let Π be the set of all assignments consistent with $\mathcal{L}(G, \mathcal{Q})$. The adversary achieves* ***query recovery (QR)*** *when it computes and outputs a mapping: $s \mapsto \{\pi(s) : \pi \in \Pi\}$ for all $s \in S$.*

Informally, the adversary achieves query recovery if, for each $s \in S$ (a set of token sequences resulting from queries in $\mathcal{Q}$), it outputs a set of query candidates $\{\pi(s) : \pi \in \Pi\}$ containing every query that is consistent with the leakage. Note that this implies that the output always contains the correct query (and possibly more). This is the best the adversary can do, given the available leakage. Note, however, that there is some information not conveyed in this mapping. In particular, by fixing an assignment for a given token sequence, we may fix or reduce the possible assignments for other query responses.

We now define a special type of QR when there exists only one assignment consistent with the query leakage, i.e. all queries can be uniquely recovered.

Definition 6. *(FQR) Let $G = (V, E)$ be a graph, $\mathcal{Q} = \{q_1, \ldots, q_k\}$ be a set of SPSP queries, and S the corresponding set of token sequences. Let Π be the set of assignments consistent with $\mathcal{L}(G, \mathcal{Q})$. We say that the adversary achieves* ***full query recovery (FQR)*** *when it (a) achieves QR, and (b) $|\Pi| = 1$.*

That is, there is a unique assignment of token sequences to queries consistent with the leakage. Whether FQR is *always possible* (i.e. for every possible set of queries $\mathcal{Q}$) depends on the graph G. Specifically, we will see that FQR is always possible if and only if each SDSP tree arising in G is non-isomorphic and there exists one unique isomorphism from each SDSP tree to its corresponding query tree. It is easy to construct graphs for which these conditions hold (see Section 4.9). For such graphs, our QR attack always achieves FQR.

4.2 Technical Results

We develop some technical results concerning isomorphisms of trees and the behaviour of paths under those isomorphisms that we will need in the remainder of the paper. For any rooted tree $T = (V, E, r)$ and any $u \in V$, let $T[u] \subseteq T$ denote the subtree induced by u and all its descendants in T.

Lemma 1. *Let $T = (V, E, r)$ and $T' = (V', E', r')$ be rooted trees. Let $p_{u,r} = (u, w_1, \ldots, w_t, r)$ and $p_{v,r'} = (v, w'_1, \ldots, w'_\ell, r')$ be paths in T and T', respectively. If there exists an isomorphism $\varphi : T \to T'$ such that $\varphi(u) = v$, then $t = \ell$ and $\varphi(w_i) = w'_i$ for all $i \in [t]$.*

Given a rooted tree $T = (V, E, r)$ and any $u \in V$, let $\mathsf{PathName}_T(u)$ denote the concatenation of the canonical names of vertices along the path from u to r in T, separated by semicolons:

$$\mathsf{PathName}_T(u) \;=\; \mathsf{Name}(T[u]) \| \text{“;”} \| \mathsf{Name}(T[w_1]) \| \text{“;”} \| \ldots \| \mathsf{Name}(T[r]). \quad (1)$$

Computing ***path names*** forms the core of our QR attack. Before we explain how we use them, we prove a sequence of results about the relationship between path names and isomorphisms. In Section 4.4 we explain how to apply a universal hash function to the path names to compress their length to $O(\log n)$ bits.

Proposition 1. *Let $T = (V, E, r)$ and $T' = (V', E', r')$ be isomorphic rooted trees and let C and C' denote the set of children of r and r', respectively. There is an isomorphism from T to T' if and only if there is a perfect matching from C to C' such that for each matched pair $c_i \in C, c'_i \in C'$, there exists an isomorphism $\varphi_i : T[c_i] \to T[c'_i]$.*

Lemma 2. *Let $T = (V, E, r)$ and $T' = (V', E', r')$ be isomorphic rooted trees. Let u and v be children of r and r', respectively. Suppose that σ is an isomorphism from $T[u]$ to $T'[v]$. Then there exists an isomorphism φ from T to T' such that $\varphi|_{T[u]} = \sigma$ and $\varphi(u) = v$.*

We now come to our main technical result:

Theorem 1. *Let $T = (V, E, r)$ and $T' = (V', E', r')$ be rooted trees and let $u \in V$ and $v \in V'$. There exists an isomorphism $\varphi : T \to T'$ mapping u to v if and only if $\mathsf{PathName}_T(u) = \mathsf{PathName}_{T'}(v)$.*

Theorem 1 also gives us a method for identifying when there exists only a single isomorphism between two rooted trees. Suppose that $T = (V, E, r)$ and $T' = (V', E', r')$ are isomorphic rooted trees and that every vertex $v \in V$ has a distinct path name; then there exists exactly one isomorphism from T to T'. Intuitively, a vertex in T can only be mapped to a vertex in T' with the same path name. So if path names are unique, then each vertex in T can only be mapped to a single vertex in T', meaning there is only a single isomorphism

available. The converse also holds: if there exists exactly one isomorphism from T to T', then every vertex $v \in V$ necessarily has a distinct path name. This observation will be useful in characterizing when query reconstruction results in full query recovery. We summarise with:

Corollary 1. *Let $T = (V, E, r)$ and $T = (V', E', r')$ be isomorphic rooted trees. Every vertex $v \in V$ has a unique path name in T if and only if there exists a single isomorphism from T to T'.*

4.3 Overview of the Query Recovery Attack

Our QR attack takes as input the graph G, a set of token sequences corresponding to the set of issued queries, and comprises of the following steps:

0. **Preprocess the graph offline** (Algorithm 2). Compute the SDSP trees $\{T_v\}_{v \in V}$ of graph G. Then compute a multimap M that maps each path name arising in the T_v to the set of SPSP queries whose start vertices have the same path name.
1. **Compute the query trees online**. The trees are constructed from the token sequences as the queries are issued.
2. **Process the query trees** (Algorithm 3). Compute a dictionary D that maps each token sequence to the path name of the start vertex of the path.

Both the preprocessing step and the online steps take time $O(n^3)$ where n is the number of vertices in the graph. Note that steps 0 and 2 are trivially parallelizable. In the case that the APSP algorithm is randomized, the adversary can simply run the attack multiple times to account for different shortest path trees.

4.4 Computing the Path Names

Before diving into our attack, we describe our algorithm for computing path names which we use as a subroutine of our attack. Algorithm 1 (ComputePathNames) takes as input a rooted tree $T = (V, E, r)$ and outputs a dictionary mapping each vertex $v \in V$ to its path name. First, we call ComputeNames and obtain a dictionary Names that maps each vertex $v \in V$ to the canonical name of subtree $T[v]$. We will use a function h drawn from a universal hash function family H to compress the path names from $O(n^2)$ to $O(\log n)$. We initialize an empty dictionary $\mathsf{PathNames}$ and set $\mathsf{PathNames}[r] = h(\mathsf{Names}[r])$. We then traverse T in a depth first search manner; when a new vertex v is discovered during the traversal, we set $\mathsf{PathNames}[v]$ to the hash of the concatenation of the name of v and the path name of its parent u i.e.

$$\mathsf{PathNames}[v] = h(\mathsf{Names}[v] || \mathsf{PathNames}[u]). \tag{2}$$

When all vertices have been explored, $\mathsf{PathNames}$ is returned. The pseudocode can be found in Algorithm 1.

Lemma 3. *Let $G = (V, E)$ be a graph and let $\{T_r\}_{r \in V}$ be the set of SDSP trees of G. Let* PathNames *be the union of the outputs of running Algorithm 1 on each tree in $\{T_r\}_{r \in V}$. Let H be a universal hash function family mapping $\{0,1\}^* \to \{0,1\}^{6 \log n}$. Then for randomly sampled $h \leftarrow H$ the expected number of collisions in* PathNames *is at most $O(1/n)$.*

We note that to achieve a smaller probability of collision, one can choose a hash function family H whose output length is $c \log n$ where $c > 6$. For simplicity we invoke the universal hash function using SHA-256 truncated to 128 bits.

Lemma 4. *Let $T = (V, E, r)$ be a rooted tree on n vertices and H be a universal hash function family mapping $\{0,1\}^* \to \{0,1\}^{6 \log n}$. Upon input of T, Algorithm 1 returns a dictionary of size $O(n \log n)$ mapping each $v \in V$ to a hash of its path name in time $O(n^2)$.*

4.5 Preprocess the Graph

We first preprocess the original graph $G = (V, E)$ into the n SDSP trees. Since the adversary is assumed to have knowledge of G, this step can be done offline. We use the same all-pairs shortest paths algorithm used at setup on G to compute the n SDSP trees $\{T_v\}_{v \in V}$, where tree T_v is rooted at vertex v. For unweighted, undirected graphs, we can use breadth first search for a total run time of $O(n^2 + nm)$ where $m = |E|$; for general weighted graphs this step has a run time of $O(n^3)$ [11].

Next, we compute the path names of each vertex in $\{T_r\}_{r \in V}$, and then construct a multimap M that maps the (hashed) path name of each vertex in $\{T_r\}_{r \in V}$ to the set of SPSP queries whose start vertices have the same path name. We leverage Theorem 1 to construct this map.

We initialize an empty multimap M. For each $r \in V$ we compute PathNames by running Algorithm 1 (ComputePathNames) on T_r. For each vertex v in T_r we compute $path\_name \leftarrow \mathsf{PathNames}[v]$, and check whether $path\_name$ is a label in M. If yes, $\mathsf{M}[path\_name] \leftarrow \mathsf{M}[path\_name] \cup \{(v, r)\}$. Otherwise $\mathsf{M}[path\_name] \leftarrow \{(v, r)\}$. The pseudocode can be found in Algorithm 2.

Lemma 5. *Let $G = (V, E)$ be a graph on n vertices. Upon input of G, Algorithm 2 returns a multimap of size $O(n^2 \log n)$ mapping each $v \in V$ to its corresponding path name in time $O(n^3)$.*

4.6 Process the Search Tokens

We must now process the tokens revealed at query time. Recall that the tokens are revealed such that, the response to any shortest path query can be computed non-interactively. When a search token tk is sent to the server, the server recursively looks up each of the encrypted vertices along the path. The adversary can thus compute the query trees using the search tokens revealed at query time. First, it

initializes an empty graph G'. As label-value pairs $(\mathsf{lab}, \mathsf{val})$ are revealed in EDB, the adversary parses $\mathsf{tk}_{\mathsf{curr}} \leftarrow \mathsf{lab}$ and $(\mathsf{tk}_{\mathsf{next}}, c) \leftarrow \mathsf{val}$, and adds $(\mathsf{tk}_{\mathsf{curr}}, \mathsf{tk}_{\mathsf{next}})$ as a directed edge to G'. At any given time, G' will be a forest comprised of $n' \leq n$ trees, $\{Q_i\}_{i\in[n']}$, such that each Q_i has at most n nodes. Identifying the individual trees in the forest can be done in time $O(n^2)$. The adversary can compute the query trees online and the final step of the attack can be run on any set of query trees. A complete query tree corresponds to the set of all queries to some fixed destination. For ease of explanation, we assume Algorithm 3 takes as input the set of complete query trees constructed from the leakage.

4.7 Map the Token Sequences to SPSP Queries

In the last step, we take as input the set of query trees $\{Q_i\}_{i\in[n']}$ constructed from the leakage. We use the path names of each vertex in the $\{Q_i\}_{i\in[n']}$, to construct a dictionary D that maps each token sequence s to the path name of the starting vertex of the corresponding path in its respective query tree.

We first initialize an empty dictionary D. For each complete query tree Q_i, we compute $\mathsf{PathNames} \leftarrow \textsc{ComputePathNames}(Q_i)$ and take the union of $\mathsf{PathNames}$ and D. The pseudocode can be found in Algorithm 3.

Theorem 2. *Let $G = (V, E)$ be a graph and EDB be an encryption of G using the GKT scheme. Let $\{Q_i\}_{i\in[n']}$ be the query trees constructed from the leakage of queries issued to EDB. Upon input of G, Algorithm 2 returns a dictionary M mapping each path name to a set of SPSP queries in time $O(n^3)$. Upon input of G and $\{Q_i\}_{i\in[n']}$, Algorithm 3 returns a dictionary D mapping token sequences to path names in time $O(n^3)$. Moreover, the outputs D and M have the property that, for any token sequence s corresponding to a path (v, r) in a query tree and for every query $(v', r') \in \mathsf{M}[\mathsf{D}[s]]$, there exists an isomorphism φ from Q to $T_{r'}$ such that $\varphi(v) = v'$ and $\varphi(r) = r'$.*

4.8 Recover the Queries

Once the map between each node (token) in a query tree and its corresponding path name has been computed, the attacker can use M and D to compute the candidate queries of all queries in the complete query trees. Given M and D (outputs of Algorithms 2 and 3, respectively) and an observed token s matching a query in the query trees for some unknown query, the adversary can find the set of queries consistent with s by simply computing $\mathsf{M}[\mathsf{D}[s]]$.

4.9 Full Query Recovery

We conclude this section with a discussion of when FQR is possible. By the correctness of our attack, this is the case for a graph G, a set of complete query trees $\{Q_i\}_{i\in[n']}$, and associated token sequences S when for $\mathsf{M} \leftarrow \textsc{PreprocessGraph}(G)$, $\mathsf{D} \leftarrow \textsc{QueryMapping}(G, \{Q_i\}_{i\in n'})$ and all $s \in S$ we

have $|\mathsf{M}[\mathsf{D}[s]]| = 1$. We can also phrase a condition for FQR feasibility in graph-theoretic terms. Recall Corollary 1, which states that given two isomorphic rooted trees T and T', if each vertex in T has a unique path name, then there exists only one isomorphism from T to T'. We deduce that FQR is always achievable for any set of complete query trees, when all n^2 vertices in the SDSP trees have unique path names. Formally, we have the following:

Corollary 2. *Let $G = (V, E)$ be a graph and let $\{T_v\}_{v \in V}$ be the set of SDSP trees of G. Suppose every vertex in $\bigcup_{v \in V} T_v$ has a unique path name (and in particular, each $T \in \{T_v\}_{v \in V}$ has a unique canonical name). Then FQR can always be achieved on any complete query tree(s). The converse is also true.*

Our attack achieves FQR whenever possible. For example, let $\mathcal{G}$ be the family of graphs having one central vertex c and any number of paths all of distinct lengths appended to c. Our attack achieves FQR for all graphs $G \in \mathcal{G}$.

5 Experiments

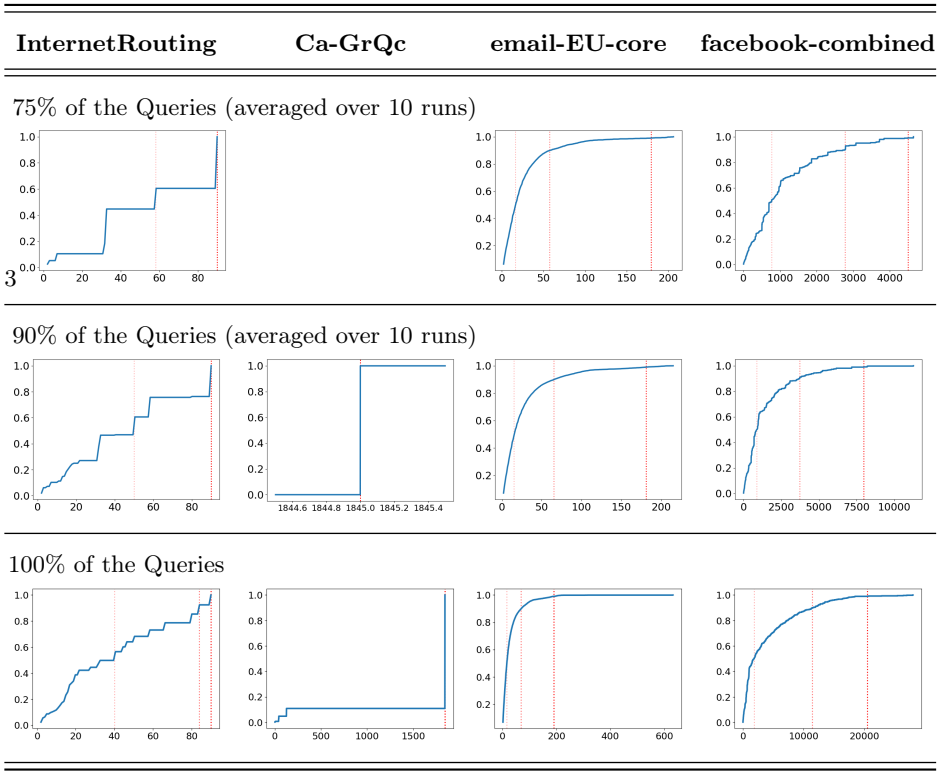
Implementation Details. We implemented our attacks in Python 3.7.6 and ran our experiments on a computing cluster with a 2 x 28 Core Intel Xeon Gold 6258R 2.7GHz Processor (Turbo up to 4GHz / AVX512 Support), and 384GB DDR4 2933MHz ECC Memory. To generate the leakage, we implemented the GES from [12] and we used the same machine for the client and the server. The cryptographic primitives were implemented using the PyCryptodome library version 3.10.1 [9]; for symmetric encryption we used AES-CBC with a 16B key and for collision resistant hash functions we used SHA-256. For the DES, we implemeted Π_{bas} from [4] and generated the tokens using HMAC with SHA-256 truncated to 128 bits. The shortest paths of the graphs were computed using the `single_source_shortest_path` algorithm from the NetworkX library version 2.6.2 [8]. We used our own implementation of the AHU algorithm. Our attack is highly parallelizable, and we exploited this when implementing our attack.

We evaluate our attacks on 8 real world datasets, the details of the datasets can be found in Appendix C. We also deployed our attacks on random graphs, the results of which can be found in the full version[10].

5.1 Query Reconstruction Results

We carried out our attack on the Internet Routing, CA-GrQc, email-EU-Core, facebook-combined, and p2p-Gnutella08 datasets; The online portion of the attack (Algorithm 3) given all queries ran in 0.087s, 0.093s, 5.807s, 102.670s, and 339.957s for each dataset, respectively. For the first four datasets, we also ran attacks given 75% and 90% of the queries averaged over 10 runs and sampled as follows: The start vertex was chosen uniformly at random and the end vertex was chosen with probability linearly proportional to its out degree in the original graph. This simulates a more realistic setting in which certain "highly connected" destinations are chosen with higher frequency. The results of these experiments

Fig. 2: CDFs for QR of the data sets after observing (top) 75%, (middle) 90%, and (bottom) 100% of the queries. On the x axis we plot the number of candidate queries output by our attack and on the y axis we plot the percent of total queries. The red dotted lines indicate the 50th, 90th, and 99th percentiles. Because of Ca-GrQC's high symmetry, complete query trees could only be constructed after at least 80% of the queries were observed and hence its first graph is omitted.



can be found in Figs. 2 and 3. Queries can be reconstructed with just 75% of the queries. In fact, with high probability, we start seeing complete query trees with as few as 20% of the queries for the facebook-combined dataset.

For the remaining datasets we ran simulations to demonstrate the success that an adversary could achieve given 100% of the queries. Our simulations were carried as follows. Given G, the SDSP trees and the path names for each vertex in these trees were computed, and then a dictionary mapping each query in G to the set of candidate queries was constructed by identifying queries whose starting vertices have the same path name. The simulations only used the plaintext graph and the results show the success that an adversary would achieve in an end-to-end attack. In practice, our attack can be run on larger datasets by writing the map out to a back-end key-value store. These results can be found in Fig. 3.

In Table 1 we report the percent of uniquely recoverable queries when the attack is run on the set of all query trees. ***Uniquely recoverable queries*** are queries whose responses result in only one candidate. CA-GrQc had the smallest percentage of uniquely recoverable queries (0.145%) and the p2p-Gnutella04 had the largest percentage (21.911%). The small percentage for CA-GrQc can be attributed to its high density ($d = 0.995$), where density is defined as $d = 2m/(n \cdot (n-1))$. The CA-GrQc graph is nearly complete, and its SDSP trees display a high degree of symmetry. In fact, many of the query trees are isomorphic to the majority of SDSP trees, and the majority of SDSP trees have a star shape. Each non-root vertex in a star tree has the same path name, resulting in a large number of possible candidates per token sequence.

In Figs. 2 and 3, we plot the CDFs of our experiments. The Gnutella data sets exhibit a high recovery rate as a result of asymmetry and low density. 50% percent of all queries for p2p-Gnutella08, p2p-Gnutella04, p2p-Gnutella25, p2p-Gnutella30 result in at most 4, 3, 5, 5 candidate query values, respectively. Details of the 50th, 90th, and 99th percentiles can be found in Table 1.

6 Discussion

We have given a query recovery attack against the GKT graph encryption scheme from [12]. The attack model we consider is strong, but it fits within the model used in [12]. A variant of our attack for a network adversary is described in the full version [10]. Ghosh et al. [12] recommend combining the scheme with other mitigation techniques, like refreshing the key periodically; while this reduces the chance that an adversary sees a complete query tree, we note that this is not very efficient. The question of whether more practical techniques can be applied remains open.

This paper highlights the need for detailed cryptanalysis of GESs. The value of such analysis was recognised in [12] but omitted on the grounds that the impact of the leakage is application-specific. Our view is that such analysis should be done in tandem with security proofs (establishing leakage profiles) at the same time as schemes are developed. Of course, attacks should be assessed with respect to real-world datasets whenever possible, as we do here. We note that our attack works when a complete tree has been observed; one interesting open question is to extend our attack to arbitrary subsets of queries: then the adversary can construct query *subtrees* and attempt to identify isomorphic embeddings of them into the SDSP trees. Our attack leaves open the question of whether other GESs can be similarly attacked.

Acknowledgements This research was supported in part by the ThinkSwiss Research Scholarship, ETH Zürich, and the U.S. National Science Foundation. Work by F.F. performed in part while visiting ETH Zürich.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.: Data Structures and Algorithms. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edn. (1983)

2. Amazon: Amazon neptune (2021), https://aws.amazon.com/neptune/, accessed on October 27, 2021
3. Ambavi, H., Sharma, M., Gohil, V.: Densest-subgraph-discovery. https://github.com/varungohil/Densest-Subgraph-Discovery (2020)
4. Cash, D., Jaeger, J., Jarecki, S., Jutla, C.S., Krawczyk, H., Rosu, M., Steiner, M.: Dynamic searchable encryption in very-large databases: Data structures and implementation. In: 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014. The Internet Society (2014). https://doi.org/10.14722/ndss.2014.23264
5. Charikar, M.: Greedy approximation algorithms for finding dense components in a graph. In: Jansen, K., Khuller, S. (eds.) Approximation Algorithms for Combinatorial Optimization. pp. 84–95. Springer, Berlin, Heidelberg (2000). https://doi.org/10.1007/3-540-44436-x_10
6. Chase, M., Kamara, S.: Structured encryption and controlled disclosure. In: Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security. Lecture Notes in Computer Science, vol. 6477, pp. 577–594. Springer, Singapore (2010). https://doi.org/10.1007/978-3-642-17373-8_33
7. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Third Edition. The MIT Press, 3rd edn. (2009)
8. Developers, N.: Networkx (2021), https://networkx.org/, version 2.6.2
9. Developers, P.: Pycryptodome (2021), https://www.pycryptodome.org/, version 3.10.1
10. Falzon, F., Paterson, K.G.: An efficient query recovery attack against a graph encryption scheme. Cryptology ePrint Archive, Paper 2022/838 (2022), https://eprint.iacr.org/2022/838, https://eprint.iacr.org/2022/838
11. Floyd, R.W.: Algorithm 97: Shortest path. Commun. ACM **5**(6), 345 (1962). https://doi.org/10.1145/367766.368168
12. Ghosh, E., Kamara, S., Tamassia, R.: Efficient graph encryption scheme for shortest path queries. In: Proceedings of the 2021 ACM Asia CCS. p. 516–525. ASIA CCS '21, ACM, New York, NY, USA (2021). https://doi.org/10.1145/3433210.3453099
13. Goetschmann, A.: Design and Analysis of Graph Encryption Schemes. Master's thesis, ETH Zürich (2020)
14. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data (Jun 2014)
15. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed graphlab: A framework for machine learning and data mining in the cloud. Proc. VLDB Endow. **5**(8), 716–727 (Apr 2012). https://doi.org/10.14778/2212351.2212354
16. Meng, X., Kamara, S., Nissim, K., Kollios, G.: Grecs: Graph encryption for approximate shortest distance queries. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. p. 504–517. CCS '15, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2810103.2813672
17. Mouratidis, K., Yiu, M.L.: Shortest path computation with no information leakage. Proc. VLDB Endow. **5**(8), 692–703 (2012). https://doi.org/10.14778/2212351.2212352
18. Neo4j, I.: Neo4j (2021), https://neo4j.com/, accessed on October 27, 2021
19. Poh, G.S., Mohamad, M.S., Z'aba, M.R.: Structured encryption for conceptual graphs. In: Hanaoka, G., Yamauchi, T. (eds.) Advances in Information and Computer Security. pp. 105–122. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34117-5_7

20. Sala, A., Zhao, X., Wilson, C., Zheng, H., Zhao, B.Y.: Sharing graphs using differentially private graph models. In: Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference. p. 81–98. IMC '11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/2068816.2068825
21. Sealfon, A.: Shortest paths and distances with differential privacy. In: Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems. p. 29–41. PODS '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2902251.2902291
22. Shao, B., Wang, H., Li, Y.: Trinity: A distributed graph engine on a memory cloud. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. p. 505–516. SIGMOD '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2463676.2467799
23. Venkataramani, V., Amsden, Z., Bronson, N., Cabrera III, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Hoon, J., Kulkarni, S., Lawrence, N., Marchukov, M., Petrov, D., Puzar, L.: Tao: How facebook serves the social graph. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. p. 791–792. SIGMOD '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2213836.2213957
24. Wang, Q., Ren, K., Du, M., Li, Q., Mohaisen, A.: SecGDB: Graph encryption for exact shortest distance queries with efficient updates. In: Kiayias, A. (ed.) Financial Cryptography and Data Security - 21st International Conference, FC 2017, Sliema, Malta, April 3-7, 2017. Lecture Notes in Computer Science, vol. 10322, pp. 79–97. Springer (2017). https://doi.org/10.1007/978-3-319-70972-7_5
25. Wu, D.J., Zimmerman, J., Planul, J., Mitchell, J.C.: Privacy-preserving shortest path computation. In: 23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016. The Internet Society (2016). https://doi.org/10.14722/ndss.2016.23052

A A Detailed Description of the GKT Scheme

At setup, the client generates two secret keys: one for a symmetric encryption scheme SKE, and one for a dictionary encryption scheme DES. It takes the input graph G and computes the SP-matrix $M[i,j]$. It then computes a dictionary SPDX such that for each pair of vertices $(v_i, v_j) \in V \times V$, we set $\mathsf{SPDX}[(v_i, v_j)] = (w, v_j)$ if $i \neq j$ and in the SP-matrix we have $M[i,j] = w$ for some vertex w.

The client then computes a second dictionary SPDX' as follows. For each label-value pair $(\mathsf{lab}, \mathsf{val})$ in SPDX the following steps are carried out. A search token tk is computed from val using algorithm $\mathsf{DES.Token}$ and a ciphertext c is computed by encrypting val using $\mathsf{SKE.Encrypt}$. Then $\mathsf{SPDX}'[\mathsf{lab}]$ is set to (tk, c). The resulting dictionary SPDX' is then encrypted using $\mathsf{DES.Encrypt}$ to produce an output EDB, which is given to the server. Now the client can issue an SPSP query for a vertex pair (u, v) by generating a search token tk for (u, v) and sending it to the server. The server initializes an empty string resp and uses tk to search EDB and obtain a response a. If $a = \perp$, then it returns resp. Otherwise, it parses a as (tk', c), updates $\mathsf{resp} = \mathsf{resp}||c$ and recurses on tk' until $\perp$ is reached on look-up. The server returns resp, a concatenation of ciphertexts (or $\perp$) to the client. The client then uses its secret key to decrypt resp, obtaining a sequence of pairs $\mathsf{val} = (w_k, v)$ from which the shortest path from u to v can be constructed.

B Pseudocode for Attack

Algorithm 1: ComputePathNames

Input: Rooted tree $T = (V, E, r)$.
Output: Dictionary PathNames.

```
// Compute the canonical name of T[v] for all v ∈ V.
Initialize empty dictionaries Names and PathNames and empty stack S
Names ← ComputeNames(T, r, Names)
h ← H
// Concatenate the canonical names into path names.
S.push(r); Mark r as explored
PathNames[r] ← h(Names[r])
while S ≠ ∅ do
  v ← S.pop( )
  if v is not explored then
    Let u be the parent of v
    PathNames[v] = h(Names[v]PathNames[u])
    Mark v as explored
  for children w of v do S.push(w)
return PathNames
```

Algorithm 2: PreprocessGraph

Input: A graph G.
Output: A multimap M mapping path names to sets of SPSP queries.

```
// Compute the set of SDSP trees from G.
Initialize an empty multimap M
Compute {T_v}_{v∈V} by running all-pairs shortest path on G
for r ∈ V do
  PathNames ← ComputePathNames(T_r)
  // Map path names to candidate queries.
  for (v, path_name) in PathNames do
    if path_name is a label in M then
      M[path_name] ← M[path_name] ∪ {(v, r)}
    else M[path_name] ← {(v, r)}
return M
```

C Datasets

We evaluate our attacks on 6 of the same data sets as [12], the InternetRouting dataset from the University of Oregon Route Views Project, and the facebook-combined dataset [14]. InternetRouting and CA-GrQc were extracted using the dense subset extraction algorithm by Charikar [5] as implemented by Ambavi et al. [3]. Details about these datasets can be found in Table 1.

Algorithm 3: QueryMapping

Input: A graph G and a set of query trees $\{Q_i\}_{i\in[n']}$ with $n' \leq n$.
Output: A dictionary D mapping search tokens to path names.

```
Initialize empty dictionary D
for i ∈ [n'] do
    // Compute the path names of each vertex in the query trees.
    PathNames ← ComputePathNames(Q_i)
    D ← D ∪ PathNames
return D
```

Table 1: The datasets used in our experiments; n denotes the number of vertices and m the number of edges of the graph dataset. "% Unique" denotes the percent of queries that have one candidate. "% Min" denotes the minimum percent of queries needed to construct the set of all query trees (see Section 2.3).

Dataset	n	m	% Unique	% Min	Percentile		
					50	90	99
InternetRouting	35	323	2.353 %	94.1 %	40	84	90
CA-GrQc	46	1030	0.145 %	99.8 %	1845	1845	1845
email-Eu-core	1005	16,706	6.507 %	78.0 %	16	69	190
facebook-combined	4039	88,234	0.206 %	99.3 %	1826	11,424	20,480
p2p-Gnutella08	6301	20,777	21.463 %	69.7 %	4	12	64
p2p-Gnutella04	10,876	39,994	21.911 %	68.1 %	3	9	32
p2p-Gnutella25	22687	54705	16.075 %	73.7 %	5	18	54
p2p-Gnutella30	36682	88328	14.671 %	74.6 %	5	24	60

Fig. 3: CDFs for QR after observing 100% of queries. An asterisk indicates that the results were obtained by simulating the attack (see Section 5 for details).