An Efficient Query Recovery Attack Against a Graph Encryption Scheme

Francesca Falzon
Brown University
University of Chicago
United States of America
francesca_falzon@brown.edu

Kenneth G. Paterson
ETH 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 on random graphs, and show that for the real-world datasets 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 [4], Neo4j [21], GraphDB [22]) and academia (e.g. Pregel [18], GraphLab [17], Trinity [26]).

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 only support one type of query, e.g. adjacency queries [7], neighbor queries [7], approximate shortest distance queries [19], and exact shortest path queries [12, 27].

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 and decrypting this sequence reveals the vertices in the shortest path.

The GKT scheme of [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.

Ghosh et al. [12] provide a security proof of the GKT scheme in a simulation-based security model that assumes an honest-but-curious (semi-honest) server. The approach identifies a leakage profile for the GKT scheme and formally proves that the scheme leaks nothing more than this. The leakage profile comes in two parts: setup leakage (available to the server upon receipt of the encrypted data structure) and query leakage (that becomes available to the server as it processes each query). Specifically, the query leakage leaks when two queries are equal i.e. the ***query pattern***, the length of the queried path, and how two paths with the same destination intersect.

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, 20, 25]. 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. In such a scenario, the map and traffic information are widely available, but the routing information of individual users is sensitive.

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. That leakage is uniquely determined by the search token that initiates the look-up. 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, then the adversary can 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. In practice, this is a realistic 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 runtime of this phase is $O(n' \cdot n^2)$ where $n' \leq n$ is the number of complete query trees computed in the online phase. The output is guaranteed to contain the correct query.

In general, the leakage from a query can be consistent with many candidates for that query, 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)***. For such graphs, the GKT scheme offers almost no protection to queries. 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 with varying graph sizes and edge probabilities. 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 query recovery tends to result in smaller sets of candidate queries when the graphs are less dense, and that dense graphs tend to have more symmetries and hence result in larger sets of candidate queries. Note also that our attack is best possible: it always outputs a minimal set of candidates consistent with the query leakage, and the correct query is always included in the set.

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. Our supporting graph theoretic theorems may be of independent interest.
(4) We evaluate our attack against real-world datasets and random graphs.
(5) We motivate the need for detailed cryptanalysis of GESs.

1.1 Prior and Related Work

We now describe prior and related work concerning graph encryption schemes and leakage abuse attacks on structured encryption.

Graph Encryption. Chase and Kamara present the first graph encryption scheme that supports both adjacency queries and focused subgraph queries [7]. Poh et al. give a scheme for encrypting conceptual graphs [23]. Meng et al. present three schemes that support approximate shortest path queries on encrypted graphs, each with a slightly different leakage profile [19]. 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 between any two vertices in the graph. Ghosh et al. [12] and Wang et al. [27] present schemes that supports exact shortest path queries on encrypted graphs.

Other solutions for privacy preserving graph structures use other techniques like secure multiparty computation and private information retrieval (e.g. [15, 28]) and differential privacy (e.g. [24]). These approaches have different security goals from encrypted database schemes that are built on symmetric encryption.

Attacks. The leakage of graph encryption schemes was first analyzed by Goetschmann [13]. The author considers schemes that support approximate shortest path queries that use sketch-based distance oracles (e.g. [19]), presents two methods for estimating distances between nodes, and gives a query recovery attack that aims to recover the vertices in an encrypted query; the experimental evaluation demonstrates that with auxiliary knowledge on some queries, the adversary can distinguish among candidate vertices which vertex was queried. Our attack is also a query recovery attack,

but uses knowledge of the graph G rather than partial knowledge of some queries.

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 1c.

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 and automorphisms.

Definition 2.1. 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 2.2. 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. In this paper, 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 by this algorithm. Pseudocode for ComputeNames is given in Appendix A.

3 THE GKT GRAPH ENCRYPTION SCHEME

In this section, we give an overview of the graph encryption scheme of Ghosh et al. [12] and describe its leakage.

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 $(u, w_1), (w_1, w_2), \ldots, (w_{t-1}, v) \in E$. We require that this path is of minimal length in G, i.e. there does not exist a sequence of edges $(u, w'_1), (w'_1, w'_2), \ldots, (w'_{t-1'}, v) \in E$ such that $t' < t$.

SPSP queries may be answered using a number of different data structures. The GKT scheme makes use of the ***SP-matrix*** [8]. 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 [5, 7]. To ensure that the GKT scheme is non-interactive, the underlying DES must be response-revealing. Since it is germane to our analysis, we provide the syntax of a DES next.

Definition 3.1. A ***dictionary encryption scheme (DES)*** is a tuple of four algorithms $\mathsf{DES} = (\mathsf{DES.Gen}, \mathsf{DES.Encrypt}, \mathsf{DES.Token}, \mathsf{DES.Get})$ with the following syntax:

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

Correctness for a DES DES states that for all dictionaries D, for all keys sk output by DES.Gen and for pairs (lab, val) in D, executing DES.Get on input tk = DES.Token(sk, lab) and dictionary ED = DES.Encrypt(sk, D) results in output val.

Note that while the GKT scheme itself is response-hiding (i.e. the shortest path is not returned in plaintext to the client), the underlying DES used in the scheme is response-revealing, that is, the values in its encrypted dictionary ED are revealed at query time. The response-revealing property of the DES is necessary to enable the GKT scheme to operate in a non-interactive manner.

Now we provide 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 (lab, val) in SPDX the following steps are carried out. A search token tk is computed from val using algorithm DES.Token and a ciphertext c is computed by encrypting val using SKE.Encrypt. Then SPDX′[lab] is set to (tk, c). The resulting dictionary SPDX′ is then encrypted using 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 resp = 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 val = (w_k, v) from which the shortest path from u to v can be constructed.

Complexity. The GKT scheme's setup takes time $O(n^3)$ and is dominated by the cost of computing the SP-matrix. Token generation takes time $O(1)$ (assuming use of an efficient DES) and querying EDB takes time $O(t)$ where t is the maximum length of a shortest path in G. The server storage is $O(n^2)$.

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 lab = (v_i, v_j) and values are of the form val = (tk, c), using a DES. Here tk is a search token obtained by running DES.Token on a pair (w, v_j) and c is obtained by running SKE.Encrypt also on (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 DES.Token on inputs lab = (v_i, v_j). Moreover, these tokens also appear in the values in EDB that are revealed to the server at query time, that is, in 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 (in fact no two distinct queries can produce the same first token tk_1, since each such first token must be used to derive a unique label in EDB identifying the beginning of a specific shortest path).

Notice also that token sequences for different queries can be overlapping; indeed since the tokens are computed by running DES.Token on inputs 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. We note that 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 $\boldsymbol{Q}$ as $\mathcal{L}(G, \boldsymbol{Q})$. For a formal proof of security that establishes the leakage profile of the GKT scheme please refer to [12]. 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

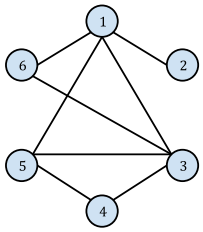
Suppose that all queries have been issued and that we have constructed all n query trees $\{Q_i\}_{i \in [n]}$, each tree having n vertices. We observe that there exists a one-to-one matching between the query trees $\{Q_i\}_{i \in [n]}$ and the SDSP trees $\{T_v\}_{v \in V}$ of G such that each matched 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

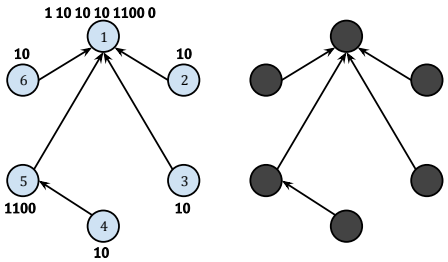
tree Q is computed (recall that each query tree must have n vertices since G is connected) the server finds all possible isomorphisms between Q and the SDSP trees. Then, for each token sequence in Q, it computes the set of paths in the SDSP trees to which that token sequence can be mapped under the possible 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 particular, without auxiliary information, the adversary can only recover the candidate queries up to symmetries arising from the isomorphisms between the query trees and the SDSP trees. 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.



(a) Original graph G.



(b) SDSP tree for vertex 1. (c) The inferred leakage.

Figure 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 the matching query tree that is leaked during setup (without any vertex labels).

4 QUERY RECOVERY

4.1 Threat Model and Assumptions

We consider a ***passive, persistent, honest-but-curious*** adversary that has compromised the server and can observe the initial search token issued, all subsequent search tokens revealed during the query processing, and the response. In particular, this adversary could be the server itself. In Appendix B we briefly outline a modified version of our attack in which we assume that the adversary has only compromised the communication channels between the client and server, and can thus only see the search tokens used to initiate the recursive look-up and the server responses.

We assume that the adversary knows the graph G that has been encrypted to create EDB. As noted previously, 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 routing/navigation scenarios. We further assume that the adversary sees enough queries to construct a subset of the n query trees. We emphasize that computing all n trees does *not* require observing all possible queries; in the real world datasets we tested, we were able to construct all query trees with as few as 68.1% of the possible queries. This is because constructing a query tree that corresponds to T_v only requires observing the queries that start at the leaf nodes of T_v and end at v. In SDSP trees with few leaves, only a small fraction of queries is needed.

We assume that the all-pairs shortest path algorithm used in constructing the SP-matrix from G during setup is deterministic. We assume that this algorithm is known to the adversary. Such an assumption is reasonable as the adversary knows G and many shortest path algorithms are deterministic, including Floyd-Warshall [11] and many of its adaptations.

4.2 Formalising Query Recovery Attacks

Query recovery (QR) in general is the goal of determining the plaintext value of queries that have been issued by the client. The notion of query recovery was introduced by Islam et al. [14] in the context of leakage abuse attacks on SSE schemes and has been extensively studied in the context of SSE and related schemes since.

We study the problem of query recovery in the context of GESs, specifically, the GKT scheme: given G, the setup leakage of the GKT scheme and the query leakage from a set of SPSP queries, our adversary's 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 4.1. (Consistency) Let $G = (V, E)$ be a graph, $\boldsymbol{Q} = \{q_1, \ldots, q_k\}$ be the set of SPSP queries that are issued, and $S = \{s_1, s_2, \ldots, s_k\}$ be the set of token 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*** $\mathcal{L}(G, \boldsymbol{Q})$ if it satisfies $\mathcal{L}(G, \boldsymbol{Q}) = \mathcal{L}(G, \pi(S))$.

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 $\mathcal{L}(G, \boldsymbol{Q})$.

Definition 4.2. (QR) Let $G = (V, E)$ be a graph, $\boldsymbol{Q} = \{q_1, \ldots, q_k\}$ be a set of SPSP queries, and S the corresponding set of token

sequences. Let Π be the set of all assignments consistent with $\mathcal{L}(G, \boldsymbol{Q})$. 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 $\boldsymbol{Q}$), 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.

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 give such an example below.

Example 4.3. *Suppose we observe the set of token sequences* $\{s_i : i \in [5]\}$ *such that* s_1, s_2, s_3, s_4 *correspond to paths of length 1 and* s_5 *corresponds to a path of length 2, with* s_4 *a subsequence of* s_5*, and which allows us to construct the query tree in Figure 1c. Further suppose that the resulting query tree is not isomorphic to any other query tree, so we know that all queries in S are rooted at 1. An adversary achieving QR must output the following mappings:*

$$\{s_1 : \{(6,1),(3,1),(2,1)\}, s_2 : \{(6,1),(3,1),(2,1)\},$$
$$s_3 : \{(6,1),(3,1),(2,1)\}, s_4 : \{(5,1)\}, s_5 : \{(4,1)\}\}.$$

However, if the adversary could fix the assignment s_1 *to* $(1,6)$ *(for example, by using auxiliary information) then* s_2 *could only be mapped to either* $(1,3)$ *or* $(1,2)$.

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

Definition 4.4. (FQR) Let $G = (V, E)$ be a graph, $\boldsymbol{Q} = \{q_1, \ldots, q_k\}$ be a set of SPSP queries, and S the corresponding set of token sequences. Let Π be the set of assignments consistent with $\mathcal{L}(G, \boldsymbol{Q})$. 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 $\boldsymbol{Q}$) 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 every path in each SDSP tree is fixed by all automorphisms of the tree. It is easy to construct graphs for which these conditions hold (see Section 4.10). For such graphs, our QR attack always achieves FQR.

4.3 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 4.5. *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* $i \in [t]$.

Proof. By assumption $\varphi(u) = v$ and by definition of isomorphism of rooted trees we also have that $\varphi(r) = r'$. Since T is a tree, then we know that there exists a unique path between u and r, and between v and r'. Isomorphisms of graphs must be edge preserving, and so φ must map the subgraph $p_{u,r}$ to $p_{v,r'}$. These two paths can only be isomorphic if they are the same length and thus $t = \ell$. Putting together these two facts we have that

$$(\varphi(u), \varphi(w_1)) = (v, w'_1), (\varphi(w_1), \varphi(w_2)) = (w'_1, w'_2),$$
$$\ldots, (\varphi(w_t), \varphi(r)) = (w_t, r')$$

which concludes the proof. □

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:

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

Computing ***path names*** will form 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 4.5 we explain how to apply a universal hash function to the path names to compress their length from $O(n^2)$ to $O(\log n)$ bits, thereby reducing storage and run time complexity.

Proposition 4.6. *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* $\varphi_i : T[c_i] \to T[c'_i]$.

Proof. To see the forwards direction, let φ denote an isomorphism from T to T' and note that if $\varphi(c) = c'$ for $c \in C$, then by the edge-preservation property of isomorphisms, φ must map the vertices of $T[c]$ to the vertices of $T[c']$, and thus $T[c] \cong T[c']$. For the backwards direction, we construct an isomorphism φ from T to T'. Let φ_r be the trivial isomorphism that takes r to r' and let

$$\varphi = \varphi_1 \cup \varphi_2 \cup \cdots \cup \varphi_k \cup \varphi_r.$$

Let $(a, b) \in E$. If (a, b) is an edge in $T[c_i]$ for some $c_i \in C$ then it is easy to see that by restricting φ to the vertices in $T[c_i]$, we have that $(\varphi(a), \varphi(b))$ is an edge in $T'[c'_i] \subseteq T'$. If $(a, b) = (c_i, r)$ for some $c_i \in C$, then $(\varphi(c_i), \varphi(r)) = (c'_i, r')$. Since c'_i is a child of r' then $(\varphi(a), \varphi(b)) \in E'$. A similar argument holds for showing that if $(a, b) \in E'$, then $(\varphi^{-1}(a), \varphi^{-1}(b)) \in E$. □

Lemma 4.7. *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* $\varphi(u) = v$.

Proof. Let C and C' denote the set of children of r and r', respectively. Since φ is an isomorphism and is edge preserving, then it must map C to C' and we necessarily have that $k = |C| = |C'|$.

We now use Proposition 4.6 to prove the lemma. Let $\hat{\varphi}$ be any isomorphism from T to T'. If $\hat{\varphi}$ maps u to v then we are done. Otherwise, $\hat{\varphi}(u) = c' \neq v$ and $\hat{\varphi}^{-1}(v) = c \neq u$ for some $c' \in C'$ and $c \in C$. By Proposition 4.6 we have that $T[u] \cong T'[c']$ and

$T[c] \cong T'[v]$, and by assumption we also know that $T[u] \cong T'[v]$. Thus, by transitivity, we conclude that $T[c] \cong T'[c']$. Let W be the vertices in $T \setminus (T[u] \cup T[c])$ and let π be an isomorphism from $T[c]$ to $T'[c']$. Then $\varphi = \hat{\varphi}|_W \cup \sigma \cup \pi$ is a collection of isomorphisms on all the trees rooted at the children of the roots. Thus φ is an isomorphism from T to T' that maps u to v. □

We now come to our main technical result:

Theorem 4.8. *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* $\mathsf{PathName}_T(u) = \mathsf{PathName}_{T'}(v)$.

Proof. The forwards direction follows from Lemma 4.5.

For the backwards direction, suppose that $\mathsf{PathName}_T(u) = \mathsf{PathName}_{T'}(v)$. Since a path name includes the canonical name of the entire tree, we deduce that $\mathsf{Name}(T[r]) = \mathsf{Name}(T'[r'])$; it follows that $T \cong T'$. Similarly we deduce that $T[u] \cong T'[v]$. More generally, let $p_{u,r} = (u, w_1, \ldots, w_{t-1}, r)$ and $p_{v,r'} = (v, w'_1, \ldots, w'_{t-1}, r')$ be paths in T and T', respectively. Then for all $i \in [t-1]$ we have that $\mathsf{Name}(T[w_i]) = \mathsf{Name}(T'[w'_i])$.

We now prove the result inductively on the vertices along the path from u to r. For the base case, take any isomorphism φ_0 from $T[u]$ to $T'[v]$ and note that this must necessarily map u to v.

We can now extend this reasoning level-by-level upwards, at each stage using the equalities of components of the two path names to extend the isomorphism. Suppose that for $k \leq t-1$ there exists an isomorphism φ_k from $T[w_k]$ to $T'[w'_k]$ such that $\varphi_k(u) = v$. By equality of path names we have that $T[w_{k+1}] \cong T'[w'_{k+1}]$. Note also that w_k and w'_k are children of w_{k+1} and w'_{k+1}, respectively. Applying Lemma 4.7, we see that there exists an isomorphism φ_{k+1} from $T[w_{k+1}]$ to $T'[w'_{k+1}]$ such that $\varphi_{k+1}|_{T[w_k]} = \varphi_k$. Since u is a vertex in $T[w_k]$, it follows that $\varphi_{k+1}(u) = \varphi_k(u) = v$. This completes the induction and with it the proof. □

Theorem 4.8 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 4.9. *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'.*

4.4 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.

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.

In practice, the attack can output a single large table T matching token sequences s to sets of queries. However, storing this large table will be more expensive than storing D and M when G has high symmetry. Moreover, D can be indexed by the first token tk in each token sequence s (since tk uniquely determines the sequence).

In the following subsections, we expand on each of the steps in the above overview.

4.5 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 Algorithm 4 (ComputeNames) on tree T, its root r, and an empty dictionary Names, 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 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, PathNames is returned. The pseudocode for ComputePathNames can be found in Algorithm 1.

Theorem 4.10. *Let $T = (V, E, r)$ be a rooted tree and* PathNames *be the output of running Algorithm 1 on T. 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 $O(1/n^3)$.*

Proof. Let $u, v \in V$ be distinct and let $(u, w_1, \ldots, w_k, r = w_{k+1})$ and $(v, w'_1, \ldots, w'_{k'}, r = w_{k'+1})$ be paths in T. We thus have

$$\mathsf{PathNames}[u] = h(\mathsf{Names}[u] \| h(\mathsf{Names}[w_1] \| h(\mathsf{Names}[w_2] \| \ldots)))$$

and similarly for $\mathsf{PathNames}[v]$. For there to be a collision between their path names then either: (1) $\mathsf{Names}[u] \| \mathsf{PathNames}[w_1]$ and $\mathsf{Names}[v] \| \mathsf{PathNames}[w'_1]$ collide or (2) for some $i \in [\min\{k, k'\}]$ and $k, k' \leq n-2$, we have that $\mathsf{Names}[w_i] \| \mathsf{PathNames}[w_{i+1}]$ and $\mathsf{Names}[w'_i] \| \mathsf{PathNames}[w'_{i+1}]$ collide. Recall that a canonical name is unique up to isomorphism of the rooted tree.

Let C_{uv} denote the event that the path names of u and v collide, and let C^j_{uv} denote the event that the j-th nested hash of u's and v's path names collide. A collision on the path names occurs when

any of the at most n pairs of nested hash values (used to compute the path names of u and v) collide. By definition of universal hash function we have $\mathbb{E}[C_{uv}^j] < 1/n^6$. Thus, by linearity of expectation,

$$\mathbb{E}[C_{uv}] = \sum_{j=1}^{\min\{k+1,k'+1\}} \mathbb{E}[C_{uv}^j] < \frac{n}{n^6} = \frac{1}{n^5}.$$

Let C denote the event of any collision of path names in T. Then by linearity of expectation the expected number of collisions is

$$\mathbb{E}[C] = \sum_u \sum_v \mathbb{E}[C_{uv}] < \frac{n^2}{n^5} = \frac{1}{n^3}.$$

□

Corollary 4.11. *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 4.12. *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 $O(n^2)$.*

Proof. Correctness follows easily from Theorem 4.10 and by a recursive argument.

Calling ComputeNames (Algorithm 4) takes $O(n^2)$ time. Reading the name of the root r and assigning the hash of its name takes at most time $O(n)$. Every node is pushed onto the stack once, and thus the **while** loop on line 12 iterates n times. Assigning a new path name on line 16 takes time $O(n)$ since Names$[v]$ is $O(n)$ bits, PathNames$[u]$ is $O(\log n)$ bits, and computing the hash takes constant time. Pushing the children of a given vertex onto the stack takes time $O(n)$ for a total run time of $O(n^2)$. PathNames maps the vertices to the hash of their path names. Each vertex and its hashed path name can be encoded with $O(\log n)$ bits yielding a dictionary of size $O(n \log n)$. □

4.6 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 has 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 4.8 to construct this map and describe the steps in detail below.

Algorithm 1: ComputePathNames

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

1: // Compute the canonical name of $T[v]$ for all $v \in V$.
2: Initialize empty dictionaries Names and PathNames
3: Initialize empty stack S
4: Names $\leftarrow$ ComputeNames$(T, r,$ Names$)$
5: $h \leftarrow H$
6:
7: // Concatenate the canonical names into path names.
8: S.push(r)
9: Mark r as explored
10: PathNames$[r] \leftarrow h($Names$[r])$
11:
12: **while** $S \neq \emptyset$ **do**
13: $v \leftarrow S$.pop( )
14: **if** v is not explored **then**
15: Let u be the parent of v
16: PathNames$[v] = h($Names$[v] \| ";" \|$PathNames$[u])$
17: Mark v as explored
18: **for** children w of v **do**
19: S.push(w)
20: **return** PathNames

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 $path\_name \leftarrow$ PathNames$[v]$, and check whether $path\_name$ is a label in M. If yes, M$[path_name]$ $\leftarrow$ M$[path_name] \cup \{(v, r)\}$. Otherwise M$[path_name] \leftarrow \{(v, r)\}$. The pseudocode for computing M can be found in Algorithm 2.

Lemma 4.13. *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 $O(n^3)$.*

Proof. For each vertex $r \in V$ we compute a dictionary mapping each vertex in T_r to its respective path names. The correctness of path names follows from Lemma 4.12.

We now analyze the run time. Computing all-pairs shortest path takes time $O(n^3)$. The **for** loop on line 5 iterates through n vertices. For each vertex in V, we run Algorithm 1 (ComputePathNames) which takes $O(n^2)$ time and the inner **for** loop on line 9 takes $O(n)$ time. Thus, the **for** loop on line 5 takes a total time of $O(n^3)$.

The multimap maps hashes of the path names to a list of candidate queries. The hashed path names have size $O(\log n)$ and there are at most n^2 distinct path names; each query corresponds to only one path name and is $O(\log n)$ bits long. The multimap thus has total size $O(n^2 \log n)$. □

4.7 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

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
  // Compute the path names of each vertex.
  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
```

Algorithm 3: QueryMapping

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

```
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
```

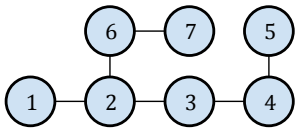


Figure 2: An example graph for which FQR is always possible, no matter what set of SPSP queries is issued.

the query trees using the search tokens revealed at query time. First, it initializes an empty graph F.

As label-value pairs (lab, 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 F. At any given time, F will be a forest comprised of $n' \le n$ 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 complete query trees.* A complete query tree corresponds to the set of all queries to some fixed destination vertex. For ease of explanation, we assume Algorithm 3 (QueryMapping) takes as input the set of all complete query trees that have been constructed from the leakage.

4.8 Map the Token Sequences to SPSP Queries

In the last step, we take as input the set of complete 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 resspective query tree .

We first initialize an empty dictionary D. For each complete query tree Q_i, we compute PathNames ← ComputePathNames(Q_i) and take the union of PathNames and D. The pseudocode for computing D can be found in Algorithm 3.

Theorem 4.14. *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 $\varphi(r) = r'$.*

Proof. The correctness of $\{Q_i\}_{i\in[n]}$ follows from the correctness of the GKT scheme. Dictionary D contains a map of each vertex in $\cup_{i\in[n]} Q_i$ to its path name. The correctness of M and D follows from Lemmas 4.13 and 4.12, respectively.

Let (v, r) be a pair comprised of a non-root vertex v and a root vertex r in a complete query tree Q, and let s be the token sequence corresponding to (v, r). Let $(v', r') \in \mathsf{M}[\mathsf{D}[s]]$. By composition of D and M we must have that $\mathsf{PathName}_Q(v) = \mathsf{PathName}_{T_{r'}}(v')$. Applying Theorem 4.8, there is thus an isomorphism from Q to $T_{r'}$ that maps v to v' and r to r'.

We now analyze the run time. Preprocessing G (Algorithm 2) takes time $O(n^3)$. Computing the path names (Algorithm 1) of $n' \le n$ trees takes $O(n^3)$ time, which gives us an upper bound on the run time of the whole attack. □

4.9 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 $\mathsf{M}[\mathsf{D}[s]]$.

4.10 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 M ← PreprocessGraph(G), D ← 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 4.9, 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 4.15. *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.*

By correctness, our attack achieves FQR whenever it is possible. In Figure 4.10, we show an example graph G for which FQR is always possible. Indeed, each tree $\{T_v\}_{v\in[7]}$ has a unique canonical name, and for all $v \in [7]$, each vertex u in T_v has a unique path name. More generally, 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. It is easy to see that our attack achieves FQR for all graphs $G \in \mathcal{G}$.

5 EXPERIMENTS

We support our theoretical results with experiments on both real world datasets and random graphs.

5.1 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 [10]; 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 [5] 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 [9].

To implement our attacks, we used used the same shortest path algorithm from NetworkX as in our scheme implementation. We also used our own implementation of the AHU algorithm (Algorithm 4) to compute canonical names. As mentioned previously, the attack is highly parallelizable, and we exploited this property when implementing our attack.

5.2 Graph Datasets

We evaluate our attacks on 6 of the same data sets as [12]; in addition we use the InternetRouting dataset from the University of Oregon Route Views Project collected on January 2, 2000 and the facebook-combined dataset. All 8 of these datasets were obtained from [16]. The InternetRouting and CA-GrQc datasets were extracted from the original datasets using the dense subset extraction algorithm by Charikar [6] as implemented by Ambavi et al. [3]. Details about these datasets can be found in Table 1.

In addition to real world datasets, we deployed our attacks on random graphs for $n = 100, 250, 500, 1000$ and edge probabilities $p = 0.2, 0.4, 0.6, 0.8$. The graphs were generated using the `fast_gnp_random_graph` function from NetworkX [9].

5.3 Query Reconstruction Results

Real world datasets. 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 can be found in Table 2. 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 out 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. We ran simulations for the larger graphs since storing all possible responses is very memory intensive; 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 the bottom row of Table 2.

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 Table 2, we plot the cumulative distribution functions (CDFs) of our experiments. The four Gnutella data sets exhibit a high recovery rate that can be explained by asymmetry and low density. 50% percent of all queries for the p2p-Gnutella08, p2p-Gnutella04, p2p-Gnutella25, p2p-Gnutella30 data sets 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. Histograms of the results can be found in Appendix C.

Random graphs. We also deployed our attack on random graphs, varying the number of nodes ($n = 100, 250, 500, 1000$) and the edge probability ($p = 0.2, 0.4, 0.6, 0.8$). Query recovery was carried out after all possible queries had been issued. For each (n, p) pair we generated 50 random graphs, encrypted the graphs using the scheme, generated the leakage for all possible SPSP queries, and deployed our query recovery attack on the responses. Then for each recovered multimap, mapping the response to the set of candidate queries, we computed the average number of queries that had each possible number of candidate queries across all 50 graphs. The CDFs of these results can be found in Table 3.

In general, we see that an increase in p and n both result in an increase in the size of the maximum query candidate sets. For example, for $n = 100, p = 0.2$ we have that 6.3% of all queries are

Dataset	n	m	d	# Comp	$\frac{\#\text{ Unique}}{\text{Total}}$	% Unique	$\frac{\#\text{ Leaves in SDSP trees}}{\#\text{ Nodes in SDSP trees}}$	% Min	Percentile 50	90	99
InternetRouting	35	323	0.543	1	$\frac{28}{1190}$	2.353 %	$\frac{1120}{1190}$	94.1 %	40	84	90
CA-GrQc	46	1030	0.995	1	$\frac{3}{2070}$	0.145 %	$\frac{2065}{2070}$	99.8 %	1845	1845	1845
email-Eu-core	1005	16,706	0.0331	20	$\frac{65,659}{1,009,020}$	6.507 %	$\frac{787,486}{1,009,020}$	78.0 %	16	69	190
facebook-combined	4039	88,234	0.011	1	$\frac{33,634}{16,309,482}$	0.206 %	$\frac{16,194,084}{16,309,482}$	99.3 %	1826	11,424	20,480
p2p-Gnutella08	6301	20,777	0.001	2	$\frac{8,519,868}{39,696,300}$	21.463 %	$\frac{27,663,800}{39,696,300}$	69.7 %	4	12	64
p2p-Gnutella04	10,876	39,994	0.0006	1	$\frac{25,915,785}{118,276,500}$	21.911 %	$\frac{80,580,827}{118,276,500}$	68.1 %	3	9	32
p2p-Gnutella25	22687	54705	0.0002	13	$\frac{82,736,533}{514,677,282}$	16.075 %	$\frac{379,383,168}{514,677,282}$	73.7 %	5	18	54
p2p-Gnutella30	36682	88328	0.0001	12	$\frac{197,413,906}{1,345,532,442}$	14.671 %	$\frac{1,003,317,663}{1,345,532,442}$	74.6 %	5	24	60

Table 1: A list of all real-world datasets used in our experiments; n denotes the number of vertices; m denotes the number of edges of the graph dataset; $d = 2m/(n \cdot (n-1))$ denotes the density of the graph. The last three columns show the 50th, 90th, and 99th percentiles obtained for Query Recovery on the eight real-world datasets.

uniquely recoverable and 50% of all queries are recoverable to at most 10 candidate queries (representing 0.101% of all queries). For $n = 1000, p = 0.2$, we have that 1.5% of all queries are uniquely recoverable and 50% of all queries are recoverable to at most 107 candidate queries (representing 0.0107% of all queries).

As p increases from 0.2 to 0.8, the graphs become more dense and we see a similar trend as seen in the real-world data sets. Denser graphs are closer to complete, and result in more symmetries and larger candidate query sets. As p increases, we also see more "waves" in the CDFs; in the graphs showing the probability density functions (PDFs) (see Appendix C), these correspond to large clusters of candidate queries all of which have the same path name and hence which cannot be distinguished in a QR attack.

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]. The attack begins with an offline preprocessing phase of the graph. In the online phase, our attack waits until it has observed all queries to at least one destination vertex and then outputs a list of candidates for each of these queries. Our attack has the property that the output contains everything that is consistent with the leakage (and nothing more), and always contains the correct query. We have given a precise characterization of when full query recovery is possible. We evaluated our attack against real-world and random graphs.

A further variant of our attack – and which is an interesting open question – applies when arbitrary subsets of queries have been issued: then the adversary can construct *partial* query trees and attempt to identify isomorphic embeddings of them into the SDSP trees. Yet another variant of our attack, for a network adversary, is described in Appendix B.

This paper highlights the need for detailed cryptanalysis of graph encryption schemes. The value of such analysis was recognised in [12] but omitted on the grounds that the impact of the leakage is application-specific and can only be assessed in the context of particular use cases at the time of deployment. 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.

Our attack leaves open the question of whether other graph encryption schemes can be similarly attacked. On the constructive side, the question of whether we can build more secure schemes that utilize chaining in a non-interactive manner and which support shortest path queries remains open. Another interesting line of research includes constructing practical interactive graph database schemes that minimize the communication overhead and number of rounds between the client and the server. Moreover, there is still much work to be done regarding the design of encrypted graph database schemes that can support a variety of queries – an important property for schemes in practical settings.

ACKNOWLEDGEMENTS

The first author was funded in part by the ThinkSwiss Research Scholarship, the NetApp University Research Fund, and ETH Zürich. Parts of this work were written while the first author was visiting ETH Zürich.

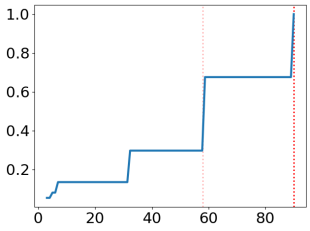
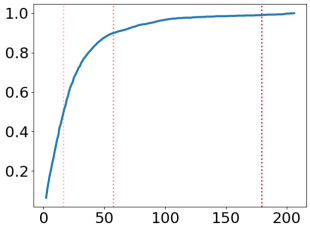
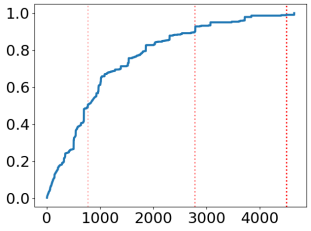
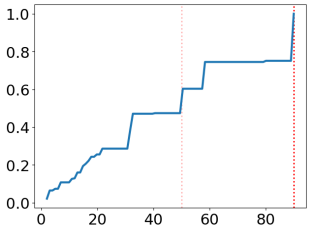
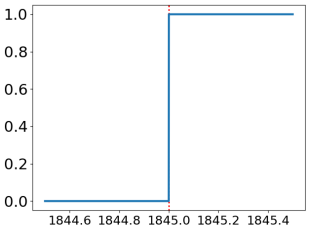
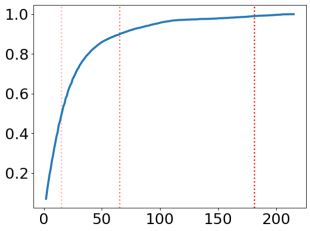
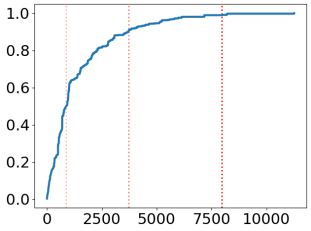
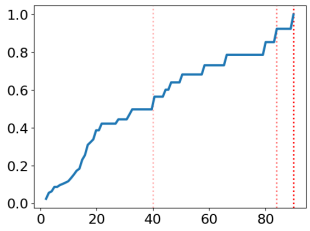
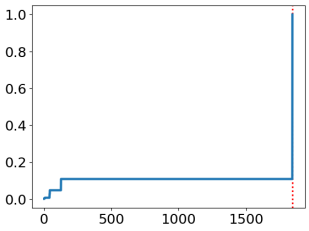
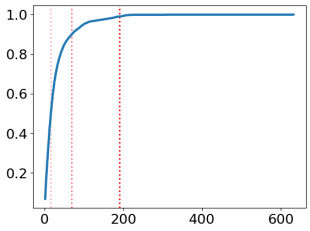
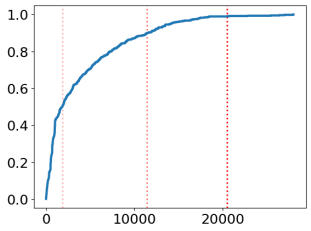
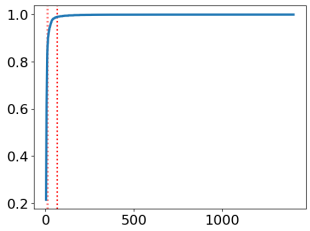
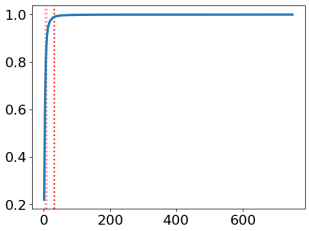
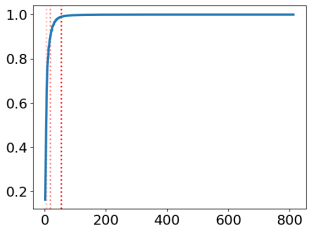
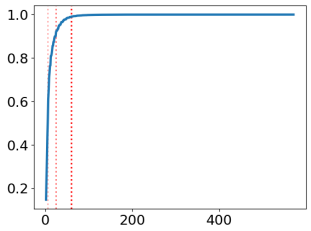
InternetRouting	Ca-GrQc	email-EU-core	facebook-combined
75% of the Queries (averaged over 10 runs)			
			
90% of the Queries (averaged over 10 runs)			
			
100% of the Queries			
			
p2p-Gnutella08	**p2p-Gnutella04**	**p2p-Gnutella25**	**p2p-Gnutella30**
100% of the Queries			
			

Table 2: CDFs for QR of the real-world data sets after observing (row 1) 75%, (row 2) 90%, and (rows 3 and 4) 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.

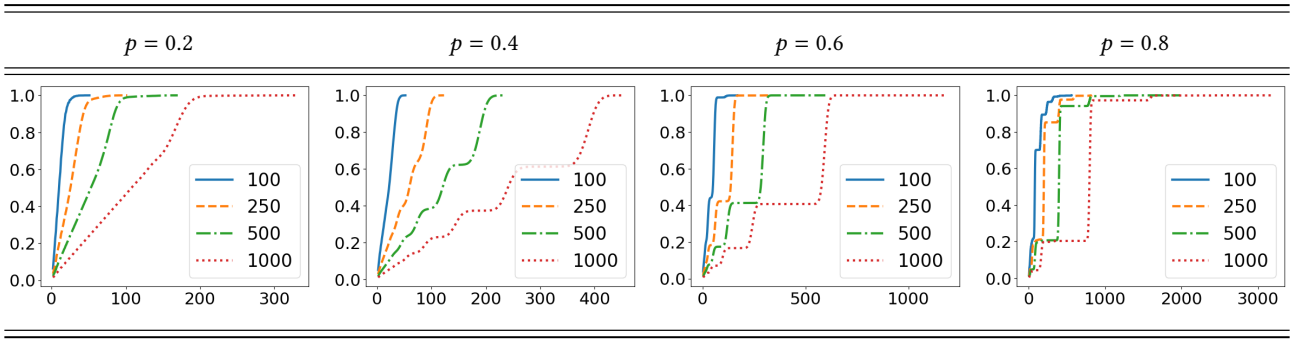


Table 3: CDFs for QR of random graphs for $n = 100, 250, 500, 1000$ and $p = 0.2, 0.4, 0.6, 0.8$ after observing 100% of the queries. On the x axis we plot the number of candidate queries output by our QR attack and on the y axis we plot the percent of total queries. For each (n, p) we generated 50 graphs and took an average of the number of vertices with each given set size of candidate queries. We observe that as the edge probability increases, the number of symmetries, and hence the number of candidate queries output tends to increase.

REFERENCES

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey Ullman. 1983. *Data Structures and Algorithms* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., USA.

[2] Amazon. 2021. Amazon Neptune. https://aws.amazon.com/neptune/ Accessed on October 27, 2021.

[3] Heer Ambavi, Mridul Sharma, and Varun Gohil. 2020. Densest-Subgraph-Discovery. https://github.com/varungohil/Densest-Subgraph-Discovery.

[4] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX Association, San Jose, CA, 49–60.

[5] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. 2014. Dynamic searchable encryption in very-large databases: data structures and implementation. In *21st Annual Network and Distributed System Security Symposium 2014 (NDSS 2014)*. The Internet Society.

[6] Moses Charikar. 2000. Greedy Approximation Algorithms for Finding Dense Components in a Graph. In *Approximation Algorithms for Combinatorial Optimization*, Klaus Jansen and Samir Khuller (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 84–95.

[7] Melissa Chase and Seny Kamara. 2010. 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)*. Springer, Singapore, 577–594.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[9] NetworkX Developers. 2021. NetworkX. https://networkx.org/ version 2.6.2.

[10] PyCryptodome Developers. 2021. PyCryptodome. https://www.pycryptodome.org/ version 3.10.1.

[11] Robert W. Floyd. 1962. Algorithm 97: Shortest path. *Commun. ACM* 5, 6 (1962), 345. https://doi.org/10.1145/367766.368168

[12] Esha Ghosh, Seny Kamara, and Roberto Tamassia. 2021. Efficient Graph Encryption Scheme for Shortest Path Queries. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security* (Virtual Event, Hong Kong) *(ASIA CCS '21)*. Association for Computing Machinery, New York, NY, USA, 516–525.

[13] Anselme Goetschmann. 2020. *Design and Analysis of Graph Encryption Schemes*. Master's Thesis. ETH Zürich.

[14] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012*. The Internet Society, San Diego, California, USA.

[15] Shangqi Lai, Xingliang Yuan, Shi-Feng Sun, Joseph K. Liu, Yuhong Liu, and Dongxi Liu. 2019. GraphSE^2: An Encrypted Graph Database for Privacy-Preserving Social Search. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security* (Auckland, New Zealand) *(Asia CCS '19)*. Association for Computing Machinery, New York, NY, USA, 41–54.

[16] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[17] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.* 5, 8 (April 2012), 716–727.

[18] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. 135–146.

[19] Xianrui Meng, Seny Kamara, Kobbi Nissim, and George Kollios. 2015. GRECS: Graph Encryption for Approximate Shortest Distance Queries. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 504–517.

[20] Kyriakos Mouratidis and Man Lung Yiu. 2012. Shortest Path Computation with No Information Leakage. *Proc. VLDB Endow.* 5, 8 (2012), 692–703. https://doi.org/10.14778/2212351.2212352

[21] Inc. Neo4j. 2021. Neo4j. https://neo4j.com/ Accessed on October 27, 2021.

[22] Ontotext. 2021. GraphDB. https://graphdb.ontotext.com/ Accessed on October 27, 2021.

[23] Geong Sen Poh, Moesfa Soeheila Mohamad, and Muhammad Reza Z'aba. 2012. Structured Encryption for Conceptual Graphs. In *Advances in Information and Computer Security*, Goichiro Hanaoka and Toshihiro Yamauchi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 105–122.

[24] Alessandra Sala, Xiaohan Zhao, Christo Wilson, Haitao Zheng, and Ben Y. Zhao. 2011. Sharing Graphs Using Differentially Private Graph Models. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference* (Berlin, Germany) *(IMC '11)*. Association for Computing Machinery, New York, NY, USA, 81–98.

[25] Adam Sealfon. 2016. Shortest Paths and Distances with Differential Privacy. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (San Francisco, California, USA) *(PODS '16)*. Association for Computing Machinery, New York, NY, USA, 29–41.

[26] Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: A Distributed Graph Engine on a Memory Cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 505–516.

[27] Qian Wang, Kui Ren, Minxin Du, Qi Li, and Aziz Mohaisen. 2017. SecGDB: Graph Encryption for Exact Shortest Distance Queries with Efficient Updates. In *Financial Cryptography and Data Security - 21st International Conference, FC 2017, Sliema, Malta, April 3-7, 2017, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 10322)*, Aggelos Kiayias (Ed.). Springer, 79–97.

[28] David J. Wu, Joe Zimmerman, Jérémy Planul, and John C. Mitchell. 2016. 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. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/privacy-preserving-shortest-path-computation.pdf

A AHU ALGORITHM PSEUDOCODE

In this section, we present a slightly modified version the original AHU algorithm [1] for computing canonical names of trees. ComputeNames takes as input a rooted tree $T = (V, E, r)$, a vertex $v \in V$, and dictionary Names. The algorithm returns a dictionary Names mapping each descendent u of v to the short canonical name of the subtree $T[u]$. Maintaining the dictionary Names enables us to compute the canonical names of the subtrees rooted at each $v \in V$ in a single traversal of T. ComputeNames takes time and space $O(n^2)$ where $|V| = n$. Note that the original AHU algorithm can be modified to run in $O(n)$ time, by only considering one level at a time and reassigning integers to the vertices at that level [1]. In contrast, we need to assign names to each vertex in the tree in order to later compute the path names.

The pseudocode of ComputeNames can be found in Algorithm 4.

Algorithm 4: [1] ComputeNames

Input: Rooted tree $T = (V, E, r)$, vertex $v \in V$, and dictionary Names
Output: Dictionary Names

if v is a leaf **then**
  Names$[v]$ = "10"
  return Names
else
  Initialize empty list $temp$ = [ ]
  for child u of v **do**
    Names ← ComputeNames$(T, v,$ Names$)$
    $temp$.append(Names$[u]$)
  Sort $temp$ in ascending order
  Concatenate names in $temp$ as $children\_name$
  Names$[v]$ = "1"$||children\_name||$"0"
  return Names

B ATTACK FOR A NETWORK ADVERSARY

Another variant of our attack is possible for a network adversary rather than a server-based adversary. Such an adversary is assumed to know G but is able to see only the communications between the client and server, that is, search tokens and responses (which, recall, are concatenations of ciphertexts). Note that, for an SPSP query (u, v), the ciphertexts correspond to the vertices that follow u along the path $p_{u,v}$. If the adversary were able to observe the query leakage from all possible queries then it could initialize a graph H and then for each response parse $c_1 \| c_2 \| \ldots \| c_t \leftarrow$ resp and add (c_i, c_{i+1}) to H for $i \in [t-1]$. The components of H then become the trees $\{Q_i\}_{i \in [n]}$. Note that these trees are the same as the original query trees, but are missing the leaf nodes. To complete the attack, the adversary would compute the SDSP trees $T_{v v \in V}$ and for every $T \in T_{v v \in V}$, delete the leaf nodes. Now the adversary can continue with the attack as described in Section 4.10, i.e. compute path names and assign the candidate SPSP queries to each resp. This attack would not be able to recover the candidate query values for the leaf nodes of the SDSP trees, so is slightly weaker than our main attack.

C ADDITIONAL EXPERIMENTAL RESULTS

In this section we include supplementary statistics about our experimental results. In Table 4 we plot the histograms of the results for QR on the real-world datasets assuming 100% of the queries have been observed. In Table 5 we plot the PDFs of the results for QR on the random graphs.

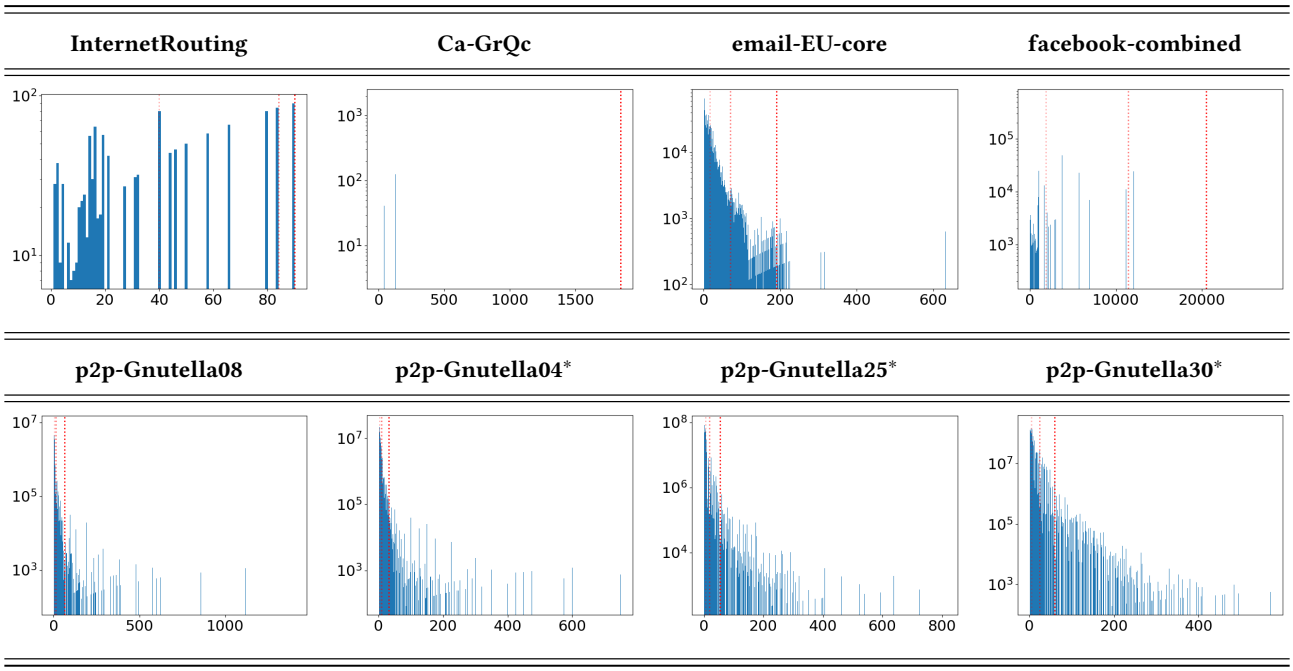


Table 4: Histograms for QR of the real world data sets after observing 100% of the queries. On the x axis we plot the number of candidate queries output by our QR attack and on the y axis we plot the number of queries. The red dotted lines indicate the 50th, 90th, and 99th percentiles. An asterisk next to the data set indicates that results were obtained via simulation, see discussion for details.

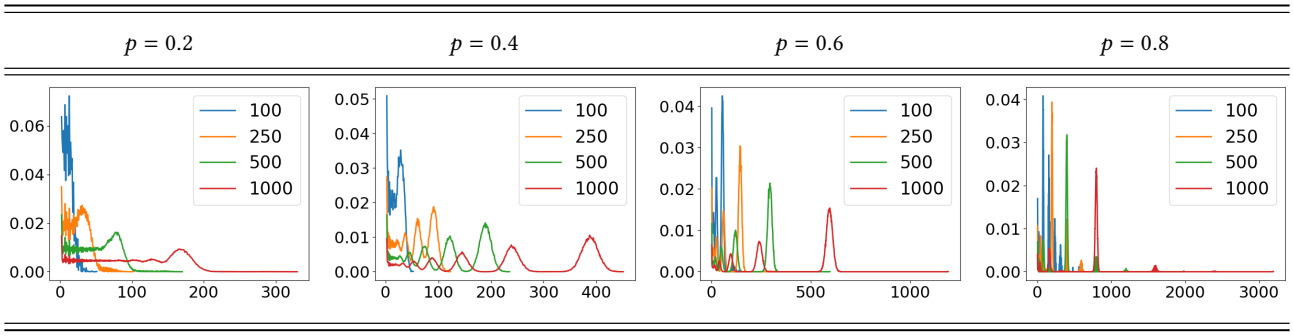


Table 5: PDFs for QR of random graphs after observing 100% of the queries. On the x axis we plot the number of candidate queries output by our QR attack and on the y axis we plot the percent of total queries.