A Privacy-Preserving Graph Encryption Scheme Based on Oblivious RAM

Seyni Kane[2,3]* and Anis Bkakria[1][0000−0002−9758−4617]

[1] IRT SystemX, Palaiseau, France
[2] Applied Crypto Group, Orange Innovation, 14000 Caen, France.
[3] SAMOVAR, Télécom SudParis, Institut Polytechnique de Paris, France.

Abstract. Graph encryption schemes play a crucial role in facilitating secure queries on encrypted graphs hosted on untrusted servers. With applications spanning navigation systems, network topology, and social networks, the need to safeguard sensitive data becomes paramount. Existing graph encryption methods, however, exhibit vulnerabilities by inadvertently revealing aspects of the graph structure and query patterns, posing threats to security and privacy. In response, we propose a novel graph encryption scheme designed to mitigate access pattern and query pattern leakage through the integration of oblivious RAM and trusted execution environment techniques, exemplified by a Trusted Execution Environment (TEE). Our solution establishes two key security objectives: (1) ensuring that adversaries, when presented with an encrypted graph, remain oblivious to any information regarding the underlying graph, and (2) achieving query indistinguishability by concealing access patterns. Additionally, we conducted experimentation to evaluate the efficiency of the proposed schemes when dealing with real-world location navigation services.

Keywords: Privacy Enhancing Technology · Graph Encryption Scheme · Oblivious RAM · Trusted Execution Environment

1 Introduction

Cloud computing is a paradigm that offers on-demand storage and computing resources to individuals and enterprises. Large data sizes motivate storage outsourcing for reasons of cost, availability, and efficiency. Storing and processing data securely on the cloud can be challenging. To protect outsourced data, Structured Encryption (SE) [6] has been proposed. A structured encryption scheme encrypts structured data in such a way that it can be queried through the use of a query-specific token that can only be generated with knowledge of the secret key. In addition, the query process reveals no useful information about either the query or the data. Structured encryption schemes include graph encryption schemes [14, 18].

* The research presented in this paper was conducted while the author was affiliated with IRT-SystemX

Graph encryption schemes have many applications, such as private navigation systems [29], online social networks [16], modeling highly confidential infrastructure and more. The main challenge is to design graph encryption schemes that are secure, expressive, and efficient. Some cryptographic techniques, such as Fully Homomorphic Encryption (FHE) [12] and secure Multi-Party Computation (MPC), can achieve high security but low efficiency. Other solutions trade off some security for better efficiency by allowing controlled leakage [14, 18]. One example of leakage is the Access Pattern (AP), which is the set of nodes and edges accessed by a query on the encrypted graph. Another example is the Query Pattern (QP), which is the set of queries issued on the encrypted graph. An adversary who observes the AP and QP can infer information about the graph structure, connectivity, and content, as well as the query frequency, similarity, and user interests.

Ghosh, Kamara and Tamassia (GKT) [14] proposed a practical graph encryption scheme that supports shortest path queries with a good trade-off between efficiency and security. They encrypts the graph using a recursive algorithm that partitions the graph into subgraphs and encrypts them separately. It has optimal preprocessing time and space, and query time proportional to those for the unencrypted graph. However, it still leaks AP and QP. Falzon et al. [10] presented an attack on the GKT scheme that exploits the QP leakage to recover the queries with high probability.

Due to access pattern and query pattern leakage, existing graph encryption schemes are not suitable for highly sensitive graphs. A possible solution is to use Oblivious RAM (ORAM) [15] techniques to hide the AP and QP leakage from an untrusted server, and avoid such as the ones proposed in [10]. ORAM is a technique that allows a client to access data in a data store without revealing which item it is interested in. It does this by accessing multiple items each time and periodically reshuffling some or all of the data in the data store.

However, using ORAM for graph encryption schemes can be inefficient, because it requires the client to interact with the server for each recursive call to fetch the path between two nodes. This results in high communication complexity. A way to overcome this issue is to use Trusted Execution Environments (TEE), which is a technology that allows an application to create a protected area in its address space, called an enclave. The data and instructions inside the enclave are confidential and secure even if the attacker has full control of the operating system. TEE can be used to create more efficient oblivious data structures [11, 24, 25, 30], by placing a mini-client inside the untrusted server. This reduces the communication complexity between the client and the server.

1.1 Our Contributions

In this paper, we introduce an innovative graph encryption scheme designed to facilitate shortest path queries without compromising any information pertaining to the underlying data or the query itself. Our approach builds upon the foundation of the GKT scheme [14], widely recognized as the state-of-the-art solution for graph encryption, and incorporates advanced privacy-preserving techniques such as Oblivious RAM (ORAM) [15] and hardware isolation mechanisms like Trusted Execution Environments (TEE) [8].

By seamlessly integrating ORAM, our scheme effectively eliminates Access Pattern and Query Pattern leakage. Additionally, the incorporation of TEE optimizes communication complexity between the client and the server by establishing a secure enclave within the untrusted server. The key contributions proposed in this work can be resumed as follows:

– We introduce TOGES, the first graph encryption scheme that synergistically employs ORAM and TEE to conceal both AP and QP leakage from an untrusted server. We prove the adaptive semantic security of our scheme, along with the assurance of AP indistinguishability and QP indistinguishability.
– We implement our scheme and conduct extensive evaluations on a real location navigation dataset, showcasing its practicality and efficiency.

1.2 Organization of the paper

The rest of this paper is organized as follows: Section 2 reviews the related work on graph encryption, ORAM, and TEEs. It also discusses recent constructions that combine ORAM and TEE to create more efficient oblivious data structures. Section 3 introduces the notation, definitions, and background schemes used in this paper, such as GKT, Path ORAM, and TEEs. Section 4 defines the system and security model that we consider for our construction. It also states the assumptions, threat model, and security goals that we aim to achieve. Section 5 describes our construction in two versions: a basic version utilizing ORAM-based graph encryption, and an enhanced version leveraging TEE-ORAM based graph encryption. Section 6 analyzes the security of our construction and provides a formal security proof. Section 7 analyzes the performance of our construction. Section 8 concludes the paper and suggests some possible future directions.

2 Related Work

In this section, we delve into the advancements in the key theoretical foundations underpinning our research. We commence with an exploration of graph encryption, followed by an examination of ORAM. Lastly, we survey pertinent literature that leverages TEE to enhance the efficiency of oblivious data structures.

Several approaches have been proposed for private shortest path computation. Previous studies [21, 22, 31] have delved into PIR-based solutions aimed at concealing the client's location. In the approaches outlined in [22, 21], the client initiates a confidential retrieval of specific subregions within the graph that pertain to its query. Following this, the client locally calculates the shortest path within the retrieved subgraph. In the work presented in [31], the client discreetly solicits columns from the next-hop routing matrix to glean information about the subsequent hop in the shortest path. While these methodologies effectively safeguard the privacy of the client's location, it is essential to note that they do not address the concealment of the server's routing details.

Diverging from prior methodologies, Graph encryption schemes are a type of structured encryption that allows querying encrypted graph. It was introduced by Chase and Kamara [6] in their seminal work on structured encryption which is a generalization of searchable encryption to the setting of arbitrarily data structure. They also presented a model and a security definition for graph encryption. Meng et al. [18] proposed three graph encryption schemes that support approximate shortest distance queries on encrypted graphs, using different distance oracles, each with a slightly different leakage profile. Ghosh et al. [14] proposed an efficient graph encryption scheme that supports exact shortest path queries on encrypted graphs, using a recursive algorithm based on SP-matrix and non-response revealing Dictionary Encryption Scheme (DES) [4]. However, their scheme has some leakage that can be exploited by query recovery attacks, as shown by Falzon and Paterson [10].

Another family of solutions are those based on Oblivious RAM (ORAM), a technique employed to conceal the access pattern of structured encryption—an area of primary interest. The inception of ORAM can be attributed to Goldreich and Ostrovsky [15], who pioneered its development in the context of software protection and simulation. Subsequently, numerous practical constructions have emerged, with efforts aimed at enhancing the efficiency of Goldreich and Ostrovsky's initial model, as evidenced by works such as [5, 32]. However, a notable drawback of these advancements lies in their substantial client-side storage requirements, which scale proportionally with the size of the underlying data structure.

Addressing the need for more practical and space-efficient ORAMs, Shi et al. introduced recursive ORAMs [26]. This innovative approach is rooted in tree data structures, where each node functions as a small bucket ORAM. Accessing an element involves traversing a path in the tree, leading to partial reshuffling for enhanced security.

Subsequent contributions, such as those by [20, 27], further refined and expanded upon the concept of recursive ORAMs. Notably, Path ORAM [27] stands out as one of the most intriguing implementations, currently recognized as among the most efficient ORAM schemes. Path ORAM employs a binary tree structure, accessing data along the path from the root to the leaf. After each access, the data undergoes shuffling and re-encryption to mitigate potential information leakage. The challenge of large client-side storage, exemplified by the position map in Path ORAM, is effectively addressed by incorporating a second ORAM for achieving recursion.

There are recent advancements that integrate Oblivious RAM (ORAM) techniques with Trusted Execution Environments (TEE) to enhance the efficiency of oblivious data structures.Trusted Execution Environments, such as Intel's Software Guard eXtensions (SGX) [3, 8, 17], employ hardware-based approaches to optimize search operations on encrypted data within an untrusted server. Intel's SGX, in particular, has inspired various applications of TEEs.

One application involves constructing secure indexes for encrypted data. For instance, Fuhry et al. introduced HardIDX [11], an encrypted database index based on B$^+$ trees, implementing search functionality within an SGX enclave. Mishra et al. proposed Oblix [19], an oblivious search index that conceals memory accesses and result size while supporting insertions and deletions. Oblix utilizes novel oblivious-access techniques on hardware enclave platforms like Intel SGX.

Another application is private search over encrypted data. Cui et al. developed an SGX-assisted scheme [9] that protects access patterns against side channel attacks without compromising search efficiency. Amjad et al. introduced the first SGX-supported dynamic Searchable Symmetric Encryption (SSE) constructions [2], ensuring both backward and forward privacy.

Relevant to our work are approaches utilizing oblivious RAM to conceal access patterns, leveraging secure enclaves like Intel SGX for improved efficiency. For instance, Sasy et al. proposed Zerotrace [25], an enclave-based ORAM scheme offering security against powerful adversaries. Rachid et al. devised efficient techniques for constructing ORAMs using Intel SGX [24], implementing and comparing multiple ORAM schemes. Wu et al. introduced OBI [30], a multi-path ORAM demonstrating superiority over traditional single-path ORAM in terms of local stash size and insertion efficiency, while ensuring strong security guarantees.

As previously mentioned, prevalent graph encryption methods, exemplified in [14, 18], exhibit vulnerabilities, particularly in Access Pattern (AP) and Query Pattern (QP) leakage, which could be exploited for information extraction. In response, our privacy-preserving graph encryption scheme has been meticulously designed to eliminate both AP and QP leaks. In comparison to existing recursive ORAM techniques [20, 27], we enhance our ORAM execution's efficiency by introducing a non-interactive approach through server-side TEE integration, thereby fortifying security while achieving superior performance.

3 Preliminaries

In this section, we will introduce some fundamental definitions and cryptographic notions essential for the rest if this paper.

3.1 Graph

A graph $G = (V, E)$ consists of a set of vertices V and a set of edges E that connect vertices in pairs. A graph is directed if the edges have a direction from one vertex to another. Two vertices $u, v \in V$ are connected if there is a path from u to v in G. We only consider static graphs that support **Single Pair Shortest Path (SPSP)** queries. A SPSP query $SPath(G, (u, v))$ [14], takes a graph $G = (V, E)$ and two vertices $u, v \in V$ as input and returns a simple path $p_{u,v}$, i.e, a list of nodes $(w_1, ..., w_t)$ such that $(u, w_1), (w_1, w_2), \ldots, (w_t, v) \in E$. If there is no path from u to v in G, SPath returns $\perp$.

A **tree** is a graph that is connected and has no cycle. A rooted tree $T = (V, E, r)$ is a tree with a root vertex r. For any rooted tree $T = (V, E, r)$ and vertex $v \in V$, we write $T[v]$ for the subtree of T that has v and all its descendants.

3.2 SP-matrix [14]

We uses SP-matrix as a data structure to for graphs, in order to be able to perform recursive queries on the graphs.

Definition 1 (SP-matrix[7]). *An SP-matrix structure is a $|V| \times |V|$ matrix M_G that is built from a graph $G = (V, E)$ by running an All-Pairs Shortest Path (APSP) algorithm between all pairs of vertices. We associate the rows and columns of M_G with vertices in V and use (v_i, v_j) to refer to the item at row v_i and column v_j. For each pair of vertices (v_i, v_j), we store the first vertex on a shortest path from v_i to v_j at $M_G[v_i, v_j]$, or $\perp$ if there is no path. All the diagonal entries in M_G are set to $\perp$.*

Given M_G, we can find the shortest path between two vertices (v_i, v_j) as follows. Look at (v_i, v_j) in M_G to get an item w. If $w \neq \perp$, then w is the first vertex on the shortest path from v_i to v_j and we repeat the process with (w, v_j). The recursion ends when we see a $\perp$. The query time is optimal, i.e., it is proportional to the length of the shortest path.

3.3 GKT

The GKT scheme [14] supports single pair shortest path (SPSP) queries. 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 $p_{u,v} = (u, w_1, \ldots, w_t, v)$ 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. For a graph $G = (V, E)$, the SP-matrix M_G is a $|V| \times |V|$ matrix defined as follows. Entry $M_G[i, j]$ 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 $M_G[i, j] = v_k$ 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 [6, 4]. To ensure that the GKT scheme is non-interactive, the underlying DES must be response-revealing [4].

Leakage of the GKT Scheme Ghosh et al. [14] 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 (i.e., if and when a query is repeated), the length of the shortest path, and what they refer to as the path intersection pattern (PIP).

Given a of sequence of SPSP queries $(q_1, \ldots, q_t)$, where $q_i = (u_i, v_i)$, PIP of q_t reveals the intersections of $p_t = SPSP(G, (u_t, v_t))$ with previous shortest paths that have the same destination v_t, (see [14] for more details).

Attacks on the GKT Scheme Falzon et al. [10] proposed an efficient query recovery attack against GKT, exploiting query leakage to mount the attack. In their approach, the adversary receives the original graph along with leaked information about specific subsets of queries. They leverage the query leakage within the GKT scheme to execute a Query Recovery (QR) attack against it. This attack, feasible for an honest-but-curious server, necessitates knowledge of the graph G. While this might seem like a stringent requirement, it is, in fact, less demanding than the conditions allowed in the security model of [14], where the adversary even has the liberty to choose G.

The attack comprises two phases. Initially, there is an offline pre-processing phase conducted on the graph G. In this stage, they extract a plaintext description of all its shortest path trees from G'. Subsequently, they process these trees and compute candidate queries for each query, utilizing the canonical labels of each tree. A canonical label serves as an encoding of a graph, facilitating the determination of graph isomorphism. The canonical label of a rooted tree can be efficiently computed using the Aho-Hopcroft-Ullman (AHU) algorithm [1].

4 System and Security Models

In this section, we introduce the system model, system definition, threats model and security requirements of our construction.

4.1 System Model

In the presented scenario, we examine a situation where a client `C`, constrained by limited resources, seeks to delegate both data storage and computation tasks to an untrusted cloud server `SP` equipped with a TEE. This two-party system comprises a client, the data owner, and a cloud server, responsible for hosting the data. Crucially, the client places trust in the TEE embedded within the server, treating it as an extension of its own infrastructure.

4.2 Assumptions and Attacker Model

As mentioned earlier, our scheme involves two parties: `C` responsible for encrypting a graph-based dataset and uploading it to the `CS`. `C` is the party that submits queries, and assume its trustworthiness.

`CS` hosts a TEE and stores the encrypted tree ORAM. Additionally, `CS` loads data into the enclave for query processing. We consider the possibility of an attacker taking control of the operating system on `CS`.

TEE provides a secure execution environment by cryptographically safeguarding code and data on an untrusted server. However, it is susceptible to side-channel attacks. These attacks enable adversaries to extract sensitive information by observing the effects of processing without direct access to the information source. Although the operating system (OS) is untrusted, it still manages the enclave's resources, allowing it to monitor the enclave's behavior. Specifically, the OS can generate a precise trace of the enclave's code and data accesses at the page granularity. This trace may be exploited later to deduce information about the outsourced data and/or executed SPSP queries. It's important to note that we do not address these side-channel attacks in our work. Thus we assume an adversary who cannot extract information from the TEE.

Conversely, the adversary has visibility into all communications between the TEE and the external entities, including the untrusted domain and `C`.

Particularly, the TEE is supposed to manage the position map and handles queries from `C` without disclosing sensitive information to **CS**. It also establishes a secure channel with `C` to protect their communication. We presume the TEE's trustworthiness and consider the data and data access within the TEE as secure. We Assume **CS** to be an honest but curious (semi-honest) entity, meaning it follows the protocol while attempting to acquire information about the outsourced data.

4.3 Security Goals

We are interested in building a privacy preserving graph encryption scheme that meets the following security requirements:

1. Given an encrypted graph, an adversary cannot learn any information about the underlying graph.
2. Given the view of a polynomial number of query executions for an adaptively generated sequence of queries $q = (q_1, \dots, q_n)$, an adversary cannot learn any information about q and its access pattern, except the size of the respective path.

4.4 Syntax

Our construction is defined by three algorithm `Setup`, `Query`, and `Reveal`, and is described in three versions (a trivial, enhanced, and more enhanced version). In all version of `OBGE` the `Setup` and the `Reveal` algorithms are executed by `C`. And the `Query` algorithm is executed by `C` in collaboration with **CS**.

– `Setup`$(G = (V, E), \lambda, P,$ `SKE`$)$: is PPT algorithm that takes as input a gaph G, a security parameter λ, a PRF P, and symmetric-key encryption `SKE`. It outputs an ecrypted graph in a tree ORAM T data structure, a position map PM, and keys $key := (K_1, K_2, K')$. The keys are generated during the key generation sub-protocol of the setup algorithm. K_1 and K_2 are for `SKE`, and K' is for P.
– `Query`$(q = (u, v), T, PM, K_2, K')$: is PPT algorithm that takes as input shortest path query $q = (u, v)$, an encrypted tree ORAM T, a position PM, and keys (K_2, K'). It outputs an encrypted path *resp*.

– `Reveal`($resp, K_1$): is PPT algorithm that takes as input an encrypted shortest path $resp$, and the key K_1. It outputs the decrypted shortest path $p_{u,v}$.

4.5 Security Definition

We adopt the standard security definition for graph encryption schemes [6, 18, 14], that follows the real/ideal simulation paradigm and is parameterized by a leakage function $\mathcal{L}$ that formalizes the leakage of the scheme.

Definition 2. *Let $\Pi = (Setup, Query, Reveal)$ denote a graph encryption scheme with respect to the above syntax, and λ a security parameter. Let $\mathcal{A}$ a PPT stateful adversary, $\mathcal{CH}$ his challenger, and $\mathcal{S}$ a stateful simulator that gets the leakage functions $\mathcal{L}$. We consider the probabilistic experiments $\boldsymbol{Real}_{\Pi,\mathcal{A}}(\lambda)$ and $\boldsymbol{Ideal}_{\Pi,\mathcal{A},\mathcal{S}}(\lambda)$ described as follows:*

$\boldsymbol{Real}_{\Pi,\mathcal{A}}(\lambda)$ *:*

– *The challenger $\mathcal{CH}$ runs the key generation procedure from the `Setup` algorithm, and generate key $:= (K_1, K_2, K')$ for symmetric-key encryption and the pseudo-ramdom function.*
– *$\mathcal{A}$ chooses a graph $G = (V, E)$ and sends it to $\mathcal{CH}$ who encrypt it using K_1 and output an encrypted tree ORAM T.*
– *$\mathcal{A}$ makes a polynomial number of adaptive queries $q = (q_1, \ldots, q_n)$, $n \in \mathbf{N}, n \leq Poly(\lambda)$. For each query q_i, $\mathcal{A}$ receives p_i and a token $(tk_i \leftarrow P'_K(q_i))$ from the challenger $\mathcal{CH}$ as the transcript of the execution of the `Query` and `Reveal` algorithm. Finally, $\mathcal{A}$ returns a bit b as the output by the experiment.*

$\boldsymbol{Ideal}_{\Pi,\mathcal{A},\mathbf{S}}(\lambda)$ *:*

– *$\mathcal{A}$ chose graph $G = (V, E)$.*
– *Given $\mathcal{L}(G)$, $\mathcal{S}$ output an encrypted tree ORAM T, and send it to $\mathcal{A}$.*
– *$\mathcal{A}$ makes a polynomial number of adaptive queries $q = (q_1, \ldots, q_n), n \in \mathbf{N}, n \leq Poly(\lambda)$. For each query q_i, $\mathcal{S}$ given $\mathcal{L}(G, q_i)$ returns token tk_i and p_i to $\mathcal{A}$. By simulating the execution of the `Query` and `Reveal` algorithms with $\mathcal{S}$ playing the role of `C`, and $\mathcal{A}$ playing the role of the server. Finally, $\mathcal{A}$ returns a bit b as the output by the experiment.*

where $b = 1$ indicates that $\mathcal{A}$ believes it is interacting with the real experiment, and $b = 0$ indicates otherwise. We say that Π is $\mathcal{L}$-secure against adaptive chosen-query attacks if for all PPT adversaries $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that

$$|PR[\boldsymbol{Real}_{\Pi,\mathcal{A}}(\lambda)] - PR[\boldsymbol{Ideal}_{\Pi,\mathcal{A},\mathcal{S}}(\lambda)]| \leq negl(\lambda) \tag{1}$$

5 ORAM Based Graph Encryption (`OBGE`)

In this section, we will present our construction: a trivial version without recursion, which serves as a model for the scheme, an enhanced version that uses TEEs, and the enhanced version which is a recursive version of the enhanced version.

5.1 A First Construction

Here we give the description of our trivial construction followed by the algorithms with their detailed explanation. Our trivial construction uses a binary tree storage on the server as in Path ORAM, to eliminate the access and query pattern in the GKT schemes.

Description of the Protocol We propose a novel protocol `OBGE = (Setup, Query, Reveal)` for privacy-preserving graph encryption based on GKT [14], and Path ORAM [27]. Our protocol allows a client to outsource a graph to a server and query it efficiently without revealing any information about the graph structure or the query results. We use a symmetric-key encryption scheme `SKE = (Gen, Encrypt, Decrypt)` similar to GKT and a pseudorandom function P in our protocol.

The setup phase includes The key generation procedure, that produces K_1 and K_2 for `SKE`, and K' for the PRF P. In the setup phase, the client encrypts the graph and sends it to the server. In the query phase, we use the `Access` protocol in Path ORAM [27] as sub-procedure in order to be able to perform the oblivious access to the ORAM structure. To do so the client sends a query token the ORAM controller which send back the corresponding encrypted path. The reveal phase is just a decryption of the encrypted path returned by the query phase.

Setup The setup process is depicted in Algorithm 1. Suppose we have a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. `C` creates an SP-matrix M_G, which stores the shortest paths between any pair of vertices in G. Then, the client performs the same steps as in GKT to obtain a dictionary $SPDX$, which maps each pair of vertices (u, v) to the next vertex w on the shortest path linking u to v. The client then transforms $SPDX$ into a new dictionary $SPDX'$ as follows. For each entry $(u, v) \mapsto w$ in $SPDX$, the client generates two tokens: one for the key, $tk \leftarrow P_{K'}(u, v)$, and one for the value, $tk' \leftarrow P_{K'}(w, v)$, using a pseudorandom function P. The client also encrypts the value using a symmetric key encryption scheme, $ct' \leftarrow \texttt{SKE.Encrypt}(K_1, (w, v))$, where K_1 is a secret key. The client then sets $SPDX'[tk] := (tk', ct')$ to construct the new dictionary.

The client creates a binary tree ORAM T of size N, where $ZN > |V|^2$, and each node acts as a bucket capable of holding up to Z blocks. A block is represented by the tuple $(tk, (tk', ct'), x)$, where tk is the token identifying the block (linked to the query (u, v)), tk' is the token identifying the next hop on the shortest path between u and v, ct' is the encryption of (w, v) by SKE, and x denotes the leaf node identifier in the ORAM tree T. The client initializes an empty stash ST, an array capable of holding blocks, and a position map PM, consisting of two columns. The first column stores tokens identifying blocks, and the second column stores the leaf identifier currently associated with each block. For each token $tk \in SPDX'$, the client randomly assigns a leaf identifier $x \leftarrow_R 0, 1, \ldots 2^L - 1$ and records it in the position map PM.

```
Algorithm 1: Setup
  input : Graph $G = (V, E)$, security parameter $\lambda$, pseudo random
          function $P$, symmetric-key encryption SKE
  output: Encrypted tree ORAM $T$, Position map $PM$, keys
          $key := (K_1, K_2, K')$
  Initialize empty dictionary $SPDX, SPDX'$;
  Initialise empty position map $PM$, and empty stash $ST$;
  $sp \leftarrow$ SKE.Gen($\lambda$);                              ▷ Generate keys
  $K_1 \leftarrow$ SKE.Gen($sp$); $K_2 \leftarrow SKE.Gen(sp)$; $K' \leftarrow_R \{0,1\}^\lambda$;
  $key := (K_1, K_2, K')$;
  $SPDX :=$ ComputeSPDX($G$);
  for $(lab, val) \in SPDX$ do
      $tk_{lab} \leftarrow P_{K'}(lab)$; $tk_{val} \leftarrow P_{K'}(val)$;     ▷ Encrypt labels and
       values
      $SPDX'[tk_{lab}] := (tk_{val}, ct)$, where $ct =$ SKE.Encrypt$_{K_1}(val)$;
  end
  for $tk \in SPDX'$ do
      $PM[tk] := x$, where $x \leftarrow_R \{0, 1, \ldots, 2^L - 1\}$;        ▷ Assign random
       positions
  end
  for $tk \in PM$ do
      $(tk', ct') \leftarrow SPDX'[tk]$;                  ▷ Retrieve encrypted values
      $c \leftarrow SKE.Encrypt_{K_2}(block)$, where $block = (tk, (tk', ct'), x)$;
      Upload $c$ on $\mathcal{P}(x)$;          ▷ Encrypt and upload encrypted block
  end
  return $T, PM, ST, key := (K_1, K_2, K')$;
```

C begins by initializing the tree T. For each $block = (tk, (tk', ct'), x)$, employ C to encrypt the block using K_2: $c \leftarrow$ SKE.Encrypt$(K_2, block)$. Next, retrieve the corresponding leaf identifier $x := PM[tk]$ from the position map. Subsequently, upload the encrypted block to the appropriate node along the path from the root to the leaf x in T, denoted as $\mathcal{P}(x)$. This process follows the same mechanism as described in PathORAM [27]. Consequently, by the end of the setup phase, buckets containing fewer than Z data blocks are populated with dummy blocks.

The client finishes the setup phase by uploading the ORAM tree T to the server. The client stores the position map PM, the stash ST, and the keys. The Setup protocol is described in detail in Algorithm 1.

Query To query the shortest path between nodes u and v, the client computes the token tk of the query using $P_{K'}(u, v)$. And execute `Access`(tk) with tk, which return the decrypted $block = (tk, (tk', ct')) \leftarrow$ `SKE.Decrypt`$_{K_2}(c)$. If $block \neq \bot$ `C` parse $block$ store the ciphertext ct' in a variable $resp$, and recall `Access`(tk'), this time with token of the next hope tk'. Repeats this process from step 4 until the `Access` procedure return $\bot$. The `Query` protocol is described in detail in Algorithm 2.

Algorithm 2: Query

```
input  : Query $q = (u, v)$, Encrypted tree ORAM $T$, $K_2, K'$
output: Encrypted path $resp$

C computes $tk \leftarrow P_{K'}(q)$;        ▷ Extract the corresponding leaf identifier
$x := PM[tk]$, $status := \epsilon$, $resp := \epsilon$;        ▷ Initialization
Set variable $curr := tk$;        ▷ Set current node to the leaf
while $status \neq$ "$SearchEnd$" do
    C executes $block \leftarrow$ Access$(curr)$;        ▷ Access current block
    if $block = \bot$ then
        set $status :=$ "$SearchEnd$";
    else
        Parse $block$ as $(tk, (tk', ct'), x)$;
        set  $resp := resp \cup ct'$ and $curr := tk'$;
    end
end
return $resp$;        ▷ Return encrypted path
```

Reveal Once the client receive the cipher-text he decrypts the ciphertext with his key K_1 and obtains the plaintext, which is the requested path. The `Reveal` protocol is described in detail in Algorithm 3.

Algorithm 3: Reveal

```
input  : $resp$, $key := K_1$
output: Decrypted path $p_{u,v}$

C Set variable $p_{u,v} := \epsilon$ ;        ▷ Initialization
Parse $resp$ as $ct_1 \cup ct_2 \cup \ldots \cup ct_k$ ;        ▷ Parse encrypted path
for $i = 1, \ldots k$ do
    $m_i \leftarrow$ SKE.Decrypt$(K_1, ct_i)$ ;        ▷ Decrypt each node
    Set $m := m \cup m_i$ ;        ▷ Aggregate decrypted nodes
end
return $M$;
```

Theorem 1 (Correctness of `OBGE`). *If P is a secure `PRF`, `SKE` is correct and the `ORAM` is correct, then `OBGE` is correct.*

Proof. The correctness follows from the fact that the tokens generated by P have a negligible probability of colliding in a random function, and consequently, this negligible probability extends to the PRF P as well.

Complexity Analysis

Space Complexity The space complexity analysis encompasses both server and client storage requirements.

- Server Storage. The server's storage entails maintaining a binary tree structure with a depth of $L = \lceil \log_2 N \rceil$ and 2^L leaves, where N denotes the number of nodes in the tree. Each node, known as a bucket, can accommodate up to Z real blocks. Consequently, the storage demand on the server side is $\mathcal{O}(ZN)$, which simplifies to $\mathcal{O}(N)$ due to the constant nature of Z.
- Client Storage. On the client side, storage involves retaining secret keys K_1, K_2, and K' for symmetric-key encryption (SKE) and the Pseudo-Random Function (PRF) P. Additionally, the client manages the stash ST and the position map PM. As per [27], the stash typically occupies $\mathcal{O}(\log N) \cdot w(1)$ blocks with high probability. The position map, comprising $NL = N \log N$ bits, translates to $\mathcal{O}(N)$ blocks.

Communication complexity The client communicates with the server for each query $q = (u, v)$ by reading and writing a path of $Z \log N$ blocks for each level of recursion. The number of recursion is equal to the length of the path $p_{u,v}$. Therefore, the total bandwidth used per query is $2|p_{u,v}|Z \log N$ blocks. Since Z is a constant, and $|p_{u,v}|$ is at most N, the bandwidth usage is $\mathcal{O}(N \log N)$ blocks.

Computation The server acts as a storage device, so it only retrieves and stores $\mathcal{O}(\log N)$ blocks per level of recursion. The computation is done by the client. The client's computation is $\mathcal{O}(N \log N) \cdot w(1)$ per query. In practice, most of this time is spent on encrypting and decrypting $\mathcal{O}(\log N)$ blocks per level of recursion.

Our basic construction is a straightforward adaptation of Path ORAM [27] with the GKT graph encryption scheme [14]. The client stores the stash ST and the position map PM. However, this can consume a lot of storage space on the client side, especially for large graphs. This goes against our goal, which was to reduce the client's storage and processing burden by delegating it to the cloud. Moreover, the protocol is interactive and requires communication between the client and the server for every level of recursion, which increases the communication overhead as we can see above.

5.2 Enhanced Construction

Recognizing the inadequacies of the basic construction when dealing with large graphs (e.g., location navigation graphs), we introduce a substantial enhancement aimed at employing a dual-pronged strategy to effectively tackle these challenges.

Initially, we implement a Trusted Execution Environment (TEE) on the server side, establishing a secure enclave that furnishes a confidential and tamper-resistant execution environment. Within this TEE enclave, a compact client module is deployed to oversee the management of the client's stash (ST) and position map (PM). This enclave functions as the Path ORAM controller, seamlessly handling client queries and furnishing corresponding results devoid of sensitive data exposure. By adopting this framework, we obviate the necessity for client-side storage and streamline the client-server interaction, thus mitigating security vulnerabilities associated with client-side operations.

Despite the discernible enhancements, the storage capacity of the TEE enclave remains a pivotal concern. Traditional TEE implementations, exemplified by Intel SGX, offer a finite internal storage capacity, typically set at 128 MB [28]. Although the position map and stash are usually smaller than the graph, they may still surpass the storage threshold for larger graphs. Notably, the combined sizes of the position map and stash may exceed the available storage capacity, presenting scalability hurdles.

To fortify our construction and alleviate the inherent storage limitations of TEEs, we propose a secondary approach. Leveraging a recursive ORAM data structure [23], meticulously engineered to curtail the volume of data stored within the TEE's internal memory, we aim to optimize TEE storage utilization while upholding the security assurances of the ORAM protocol. This stratagem not only enhances the scalability of our construction but also ensures the judicious employment of TEE resources, rendering it conducive for applications entailing expansive graphs and datasets.

6 Security Analysis

In the following, we examine the security of `OBGE`. We demonstrate that our construction is adaptively-secure with respect to a well-defined leakage profile. We define our leakage profile below.

Setup Leakage Our scheme has no setup leakage. Indeed, in the `Setup` phase of our scheme, we set the size of the ORAM tree $ZN > |v|^2$, so we do not reveal the size of the graph.

Query Leakage In the query phase, the server only learns the leaf identifiers associated with blocks containing the path of the current query. And we replace these leaf identifiers with random ones on the tree T as we access the blocks. Moreover, we re-encrypt the accessed blocks using a semantically secure SKE and relocate them either in the stash or in a new branch on the tree associated with the new leaf identifier. The server also does not learn the access pattern, which is guaranteed by the underlying Path ORAM protocol. So the only thing the server learns during a query is the length of the requested path $|p_{u,v}|$, by counting the number of recursions.

We can then formalize the leakage function of our scheme as $\mathcal{L} = \mathcal{L}_Q(q = (u, v)) = |p_{u,v}|$. We summarize the security of our scheme in the next theorem.

Theorem 2 (Security of *OBGE*). *If P is a secure PRF, `SKE` is INDCPA-secure, ORAM is secure according to definition in [27] then `OBGE`, as described above, is adaptively $\mathcal{L}$-semantically secure, where $\mathcal{L}$ is the leakage function.*

Proof. We define a simulator that works as follows. Given $\mathcal{L}$, the simulator generates keys $key := (K_1, K_2, K_3)$, constructs an ORAM tree T that holds N random blocks and a position map PM as specified in the real `Setup`. For each query $q = (u, v)$, the simulator receives $\mathcal{L}_Q(q)$. It checks the query pattern to see if the query is new. If not, it returns the token it generated before. Otherwise, it randomly picks a token tk that is not previously generated and a random path p and outputs (tk, p).

To show that this simulator satisfies our security definition, we first argue that the random choices of tk and p will be indistinguishable from the outputs of P and $Reveal$. Then we argue that since all entries of T and PM are random, (tk, p) will be randomly distributed and thus indistinguishable from the output of the PRF P and $Reveal$. The result follows directly.

7 Evaluation

7.1 Dataset

For our evaluation, we utilized a real world graph dataset constructed from the OpenStreetMap data for navigating Paris [13]. This dataset represents the city's infrastructure as a network of interconnected nodes and edges. The nodes in our graph dataset consist of road intersections, providing a detailed representation of the city's street layout. Additionally, points of interest, which encompass various locations searchable within Paris, serve as supplementary nodes. The total number of nodes in the constructed graph is $102,037$.

7.2 Experimental Setup

We developed SGX-based implementations of the proposed path ORAM and recursive path ORAM, leveraging Intel's SGX SDK version 2.2 and Intel's IPP cryptographic library. Our experiments were conducted on a machine equipped with 32 GB of RAM and an Intel(R) Core(TM) i7-6770HQ 2.60GHz CPU, with a maximum enclave size of 128 MB. Throughout our tests, we maintained Z=5, representing the number of blocks within each node of the recursive path ORAM.

Figure 1 reports the correlation between the length of the retrieved path and the time required for query response but also underscore the practicality of the proposed solution for finding the closest paths. As the length of the retrieved path increases, indicating a more extensive search space, the time required for

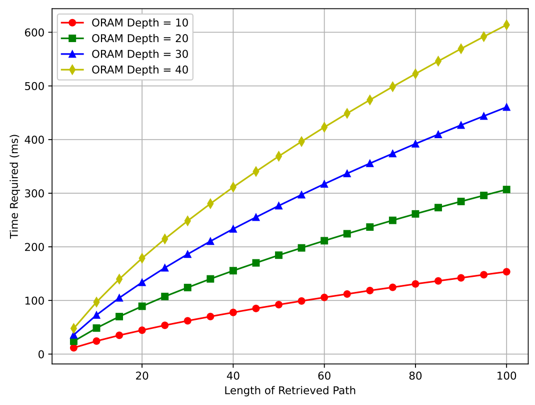

Fig. 1: Time Required for Path Query Response in Varying recursive ORAM Depths

query response escalates accordingly. Despite this escalation, the solution's efficiency remains evident, particularly in scenarios necessitating the exploration of longer paths. This suggests the solution's viability and effectiveness in practical applications where extensive pathfinding is required, demonstrating its potential utility in various real-world contexts.

8 Conclusion

In conclusion, the need for secure graph encryption schemes is paramount in various applications, including navigation systems, social networks, and infrastructure modeling. Existing methods, while efficient, suffer from vulnerabilities such as access pattern and query pattern leakage, which compromise security and privacy. To address these challenges, we propose a novel graph encryption scheme that leverages Oblivious RAM (ORAM) and Trusted Execution Environments (TEE) to conceal both access pattern and query pattern leakage. Our scheme builds upon the foundation of existing state-of-the-art solutions like the GKT scheme, enhancing security while maintaining efficiency. Through evaluations on real-world datasets, we demonstrate the practicality and effectiveness of our solution in protecting sensitive graph data while allowing secure shortest path queries.

Acknowledgments. This work were supported by the french national research agency funded project AUTOPSY (grant no. ANR-20-CYAL-0008). Additionally, part of this work was done as part of IRT SystemX project PFS (Security of Smart Ports).

References

1. Aho, A.V., Hopcroft, J., Ullman, J.: Data structures and algorithms (1983). Google Scholar Google Scholar Digital Library Digital Library (1983)
2. Amjad, G., Kamara, S., Moataz, T.: Forward and backward private searchable encryption with sgx. In: Proceedings of the 12th European Workshop on Systems Security. pp. 1–6 (2019)
3. Anati, I., Gueron, S., Johnson, S., Scarlata, V.: Innovative technology for cpu based attestation and sealing. In: Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy. vol. 13. ACM New York, NY, USA (2013)
4. Cash, D., Jaeger, J., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M.C., Steiner, M.: Dynamic searchable encryption in very-large databases: Data structures and implementation. Cryptology ePrint Archive (2014)
5. Chan, T.H.H., Guo, Y., Lin, W.K., Shi, E.: Oblivious hashing revisited, and applications to asymptotically efficient oram and opram. In: Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23. pp. 660–690. Springer (2017)
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, Singapore, December 5-9, 2010. Proceedings 16. pp. 577–594. Springer (2010)
7. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., et al.: Introduction to algorithms, chapter 11 (2001)
8. Costan, V., Devadas, S.: Intel sgx explained. Cryptology ePrint Archive (2016)
9. Cui, S., Belguith, S., Zhang, M., Asghar, M.R., Russello, G.: Preserving access pattern privacy in sgx-assisted encrypted search. In: 2018 27th International Conference on Computer Communication and Networks (ICCCN). pp. 1–9. IEEE (2018)
10. Falzon, F., Paterson, K.G.: An efficient query recovery attack against a graph encryption scheme. In: European Symposium on Research in Computer Security. pp. 325–345. Springer (2022)
11. Fuhry, B., Bahmani, R., Brasser, F., Hahn, F., Kerschbaum, F., Sadeghi, A.R.: Hardidx: Practical and secure index with sgx. In: Data and Applications Security and Privacy XXXI: 31st Annual IFIP WG 11.3 Conference, DBSec 2017, Philadelphia, PA, USA, July 19-21, 2017, Proceedings 31. pp. 386–408. Springer (2017)
12. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the forty-first annual ACM symposium on Theory of computing. pp. 169–178 (2009)
13. Geofabrik: Openstreetmap data for this region: Ile-de-france. https://download.geofabrik.de/europe/france/ile-de-france.html, accessed: 15th April, 2024
14. Ghosh, E., Kamara, S., Tamassia, R.: Efficient graph encryption scheme for shortest path queries. In: Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security. pp. 516–525 (2021)
15. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. Journal of the ACM (JACM) **43**(3), 431–473 (1996)
16. Lai, S., Yuan, X., Sun, S.F., Liu, J.K., Liu, Y., Liu, D.: Graphse^2: An encrypted graph database for privacy-preserving social search. In: Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security. pp. 41–54 (2019)

17. McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C.V., Shafi, H., Shanbhogue, V., Savagaonkar, U.R.: Innovative instructions and software model for isolated execution. Hasp@ isca **10**(1) (2013)
18. 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. pp. 504–517 (2015)
19. Mishra, P., Poddar, R., Chen, J., Chiesa, A., Popa, R.A.: Oblix: An efficient oblivious search index. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 279–296. IEEE (2018)
20. Moataz, T., Blass, E.O., Noubir, G.: Recursive trees for practical oram. Cryptology ePrint Archive (2014)
21. Mouratidis, K.: Strong location privacy: A case study on shortest path queries. In: 2013 IEEE 29th International Conference on Data Engineering Workshops (ICDEW). pp. 136–143. IEEE (2013)
22. Mouratidis, K., Yiu, M.L.: Shortest path computation with no information leakage. arXiv preprint arXiv:1204.6076 (2012)
23. Patel, S., Persiano, G., Yeo, K.: Recursive orams with practical constructions. Cryptology ePrint Archive, Paper 2017/964 (2017), https://eprint.iacr.org/2017/964, https://eprint.iacr.org/2017/964
24. Rachid, M.H., Riley, R., Malluhi, Q.: Enclave-based oblivious ram using intel's sgx. Computers & Security **91**, 101711 (2020)
25. Sasy, S., Gorbunov, S., Fletcher, C.W.: Zerotrace: Oblivious memory primitives from intel sgx. Cryptology ePrint Archive (2017)
26. Shi, E., Chan, T.H.H., Stefanov, E., Li, M.: Oblivious ram with o ((log n) 3) worst-case cost. In: Advances in Cryptology–ASIACRYPT 2011: 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings 17. pp. 197–214. Springer (2011)
27. Stefanov, E., Dijk, M.v., Shi, E., Chan, T.H.H., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path oram: an extremely simple oblivious ram protocol. Journal of the ACM (JACM) **65**(4), 1–26 (2018)
28. Will, N.C., Maziero, C.A.: Intel software guard extensions applications: A survey. ACM Computing Surveys (2023)
29. Wu, D.J., Zimmerman, J., Planul, J., Mitchell, J.C.: Privacy-preserving shortest path computation. arXiv preprint arXiv:1601.02281 (2016)
30. Wu, Z., Li, R.: Obi: a multi-path oblivious ram for forward-and-backward-secure searchable encryption. In: NDSS (2023)
31. Xi, Y., Schwiebert, L., Shi, W.: Privacy preserving shortest path routing with an application to navigation. Pervasive and Mobile Computing **13**, 142–149 (2014)
32. Yeo, K., Patel, S., Persiano, G., Raykova, M.: Oblivious ram with logarithmic overhead (Jun 1 2021), uS Patent 11,023,168