Highly-Scalable Protected Graph Database Search with Oblivious Filter

Jamie Cui
Ant Group
Hangzhou, China
shanzhu.cjm@antgroup.com

Chaochao Chen
Ant Group
Hangzhou, China
chaochao.ccc@antgroup.com

Li Wang
Ant Group
Hangzhou, China
raymond.wangl@antgroup.com

ABSTRACT
With the emerging popularity of cloud computing, the problem of how to query over cryptographically-protected data has been widely studied. However, most existing works focus on querying protected relational databases, few works have shown interests in graph databases. In this paper, we first investigate and summarise two single-instruction queries, namely Graph Pattern Matching (GPM) and Graph Navigation (GN). Then we follow their design intuitions and leverage secure Multi-Party Computation (MPC) to implement their functionalities in a privacy-preserving manner. Moreover, we propose a general framework for processing multi-instruction query on secret-shared graph databases and present a novel cryptographic primitive Oblivious Filter (OF) as a core building block. Nevertheless, we formalise the problem of OF and present its constructions using homomorphic encryption. We show that with OF, our framework has sub-linear complexity and is resilient to access-pattern attacks. Finally, we conduct an empirical study to evaluate the efficiency of our proposed OF protocol.

KEYWORDS
Protected Graph Database Search; Secret Sharing; Secure Multiparty Computation; Oblivious Filter

1 INTRODUCTION
Graph Database as-a-Service (GDaaS) has gained much popularity in the industry recently. Many cloud service providers (CSPs) offer built-in GDaaS such as Google’s Aura and Amazon’s Neptune. How to resolve people’s concern over the confidentiality of their data and provide search functionality has become one of the most imperative challenges for cloud databases.

This problem has been comprehensively studied for relational databases. First introduced in 2000 by Song [53], the problem of Protected Database Search (PDS) has motivated the research in different aspects such as Private Information Retrieval (PIR) [18, 27], Searchable Symmetric Encryption (SSE) [13], and Order-Preserving Encryption (OPE) [3]. Roughly, in PDS, data owner securely outsources its data onto cloud servers (presumably in an encrypted or secret-shared form), and then client performs searches on the protected data. The security of PDS requires that cloud servers eventually learn nothing about the securely-outsourced data except the search results. To date, existing work on PDS can be divided into three categories: encryption based solutions [42, 44, 50, 55, 58, 59], secret-sharing based solutions [6], and trusted-hardware based solutions [26]. Despite the extensive study of PDS problem, most of the work target on relational databases [6, 26, 42, 44, 50, 55, 58, 59]. There is a lack of study on searching over protected graph databases. Unlike relational databases, graph databases by design maintain the relationship between nodes, which allows fast and efficient connection check between two nodes [8]. To bridge the gap, in this work, we bring the PDS problem to graph databases, and study the Protected Graph Database Search (PGDS) problem.

Problem and Challenges. Similar to PDS, PGDS aims to provide search functionality over on a cryptographically-protected and outsourced graph database. Many works [6, 45] and industrial systems such as Crypten [1] and TFEncrypted [20] leverage secret-share secure computation model to provide protection for secret data. Therefore, we focus on the PGDS problem where the outsourced graph data is protected by a secret sharing scheme (e.g. [52]). Generally, in our solution of PGDS, we assume the existence of three types of participants:

- An honest data holder $\mathcal{H}$ who holds a private graph database but has limited storage and computation resources, and therefore $\mathcal{H}$ outsources its private data to servers;
- Two non-colluding cloud servers $\mathcal{S}_0, \mathcal{S}_1$ who have practical storage, computation, and networking capability;
- An honest client who $\mathcal{C}$ performs secure searching functionality by interacting with $\mathcal{S}_0$ and $\mathcal{S}_1$.

Intuitively, for privacy concern, cloud servers should not learn anything about the underlying graph data except for metadata (e.g., the number of vertices and edges). Also, since we facilitate two-party secret sharing for secure computation, we further restrict our threat model to a single semi-honest adversary.

There are two challenges towards the secret-sharing based PGDS problem. First, scalability. This is because graph data can contain millions or even trillions of edges and vertices in the real-world application, e.g. social graphs, interest graphs, and consumption graphs. Therefore it is essential for the solution to be highly-scalable. Second, security. Previous works on PDS leaks access patterns in order to boost performance [13], which further broadens the attack plane. Recent work [36] has demonstrated a practical attacks utilising access patterns. In this work, we aim at eliminating all information leakages, especially access patterns.

1.1 Existing Work
Apart from the vast amount of work on relational databases based PDS, recently, Lai et al. [39] proposed GraphSE$^2$, which addresses the problem of PGDS by leveraging a SSE scheme called Oblivious Cross-Tags (OXTs) [13] protocol. In particular, GraphSE$^2$ provides an encrypted structural data model to facilitate parallel and encrypted data access (like set operations and graph traversal). However, it also inherits the leakage from the original OXT protocol and inevitably leaks access patterns, and thus is prone to access pattern attacks, which contradicts the initiatives of this work.
1.2 Our Contribution

To resolve the above challenges, in this paper, we propose a novel query processing system for secret-shared PGDS problem. In particular, our system can be seen as a graph traversal machine working on secret-shared graph databases. There are two general features supported by our system: Graph Pattern Matching (GPM) and Graph Navigation (NG). We begin by introducing a novel graph sharing scheme which serves as the infrastructure of our system, and then present the secure evaluation of GPM based on Ullmann’s algorithm [56], and secure evaluation of GN based on Gremlin [49], a popular industrial query processing machine. Afterwards, we propose a secure evaluation framework for multi-instruction query, which allows the evaluation of composed single instructions. The proposed secure evaluation framework for multi-instruction query relies on a novel cryptographic primitive: Oblivious Filter. Similar to the famous oblivious transfer [17, 38], oblivious filter is a two-party functionality between a server and a client, where the client obliviously filters an input list from server, and obtains the result in a secret-shared form. During the protocol, both server and client know nothing about the result. We also present a variant of OF, i.e., Secret-Shared Oblivious Filter, which can be used as a secret-input and secret-output blackbox filter, improving the scalability of our framework. Afterwards, we provide a HE-based construction of OF and prove its security under the simulation-based security.

Contribution. We summarize our main contributions in this paper as follows.

- We introduce a new graph sharing scheme which allows efficient connectivity check between two nodes, and has practical storage complexity. The graph sharing scheme only leaks the metadata.
- We summarize exiting queries on graph database into two single instruction queries, i.e., GPM and GN, and propose their secure evaluations.
- We propose a secure graph traversal machine based on oblivious filter. Our traversal machine works on secret-shared graph databases, and allows evaluation on the sequential composition of multiple instructions.
- We introduce a new cryptographic primitive, i.e., oblivious filter, allowing obliviously filtering the database with conditions. We also provide its HE-based construction, which is secure under a semi-honest non-adaptive adversary.
- We conduct an empirical study to evaluate the performance of our proposed building block — oblivious filter.

2 BACKGROUND AND TOOLS

In this section, we present some background knowledge and definitions about graphs, databases, and some cryptographic primitives. In summary, Section 2.1 first introduces two secure computation techniques: secret sharing and additive homomorphic encryption. Then Section 2.2 shows how to model and represent graph data. Finally Section 2.3 takes an in-depth look at two fundamental query functionalities, i.e., graph pattern matching and graph navigation.

Syntax. Our solution involves many interactions among a data holder $\mathcal{H}$, two cloud servers $S_0$, $S_1$, and a client $C$. In the following of this paper, we use a pair with angle brackets, i.e., $\langle \cdot \rangle$, to denote the data that is only visible to the corresponding party. The ordering that we use is $\langle H, S_0, S_1, C \rangle$. For example $\langle x, y, z, k \rangle$ means that $x$ is only visible to $H$, $y$ is only visible to $S_0$, $z$ is only visible to $S_1$, and $k$ is only visible to $C$. Also we use $\bot$ to represent empty. We further simplify the notation and omit the empty symbol $\bot$. That is, we use $\langle x \rangle$ or $\langle \#_0x \rangle$ as an abbreviation of $\langle \bot, \#_0x, \bot \rangle$, $\langle x, \circ \rangle$ as an abbreviation of $\langle \bot, \bot, \bot, x \rangle$, and $\langle x, y, \circ \rangle$ as an abbreviation of $\langle \bot, \bot, x, y \rangle$.

2.1 Secure Computation Techniques

Secure computation addresses the main challenge of the secret-share based PGDS problem. We introduce the basic concepts and notations of two secure computation techniques, i.e., secret-sharing and additive homomorphic encryption.

Secret sharing. Secret sharing is a cryptography primitive aiming to distribute a secret among a group of parties (participants), such that each party holds a share of the secret [52]. And secret sharing ensures that only with a sufficient amount of participants, the secret can be revealed. More formally, a secret sharing scheme is a pair of algorithms between $t$ parties: $(\text{Shr}, \text{Rec})$. Since in Secret Shared Graph Database (SSGD), we only consider the situation with a distributor and two receivers, the functionalities of secret sharing from $H$ to $S_0$ and $S_1$ could be denoted as:

- $\langle \#_0x \rangle \leftarrow \text{Shr}(\langle x, \circ \rangle)$,
- $\langle x, \circ \rangle \leftarrow \text{Rec}(\langle \#_0x \rangle)$.

Additionally, secret-sharing scheme has been the building block for many multiparty computation (MPC) protocols [24, 29]. We use additive secret sharing with the field of power of two to facilitate efficient arithmetic and boolean operations.

Additive homomorphic encryption. Additive homomorphic encryption is an asymmetric encryption scheme which allows addition operation on the encrypted data [22, 47, 48]. With a formal description, additive homomorphic encryption is a tuple of algorithms $(\text{Gen, Enc, Dec, Eval})$, with “+” as its homomorphic operation. Briefly,

- $(\text{pk, sk}) \leftarrow \text{Gen}(\lambda)$: Taking a security parameter $\lambda$, Gen generates the public key pk and secret key sk,
- $c \leftarrow \text{Enc}_{pk}(m)$: Enc encrypts a message $m$ with public key, and returns a cipher c,
- $m \leftarrow \text{Dec}_{sk}(c)$: Dec decrypts a cipher c with secret key, and reveals the message $m$.

As for additive homomorphism, those schemes allow the secure evaluation of $\text{Enc}(m_1 + m_2)$ using two ciphers $c_1, c_2$ and the public key, that is, $\text{Enc}_{pk}(m_1 + m_2) = \text{Eval}([\text{Enc}_{pk}(m_1), \text{Enc}_{pk}(m_2)])$. The security of additive HE schemes is defined by the generally-adopted semantic security [30] with a computationally-bounded adversary. Appendix A describes the detailed construction of an additive homomorphic encryption scheme — paillier encryption [48].

2.2 Graph Model and Representation

Graph databases use graph model as the basic data structure for graphs. To begin with, there are two most popular graph models in the literature: Resource Description Framework (RDF) [9] and Labeled Property Graph (LPG) [5]. For simplicity, in this work, we choose a limited, yet simple and widely-adopted variant of LPG model — Edge-labeled Graph Model [4], where edges are assigned
with labels to indicate different relationships between nodes. In the following, we give a formal definition of edge-labeled graph model.

**Definition 2.1 (Edge-labeled graph).** An edge-labeled graph \( G \) is a pair \((V, E)\), where \( V = \{v_1, v_2, ..., v_n\} \) is a finite set of vertices, and \( E = \{e_1, e_2, ..., e_m\} \) is a finite set of edges, with \( n \) and \( m \) denoting the number of vertices and edges, respectively. In addition, \( E \subseteq V \times L \times V \), where \( L \) is a set of labels.

![Figure 1: A simple example of edge-labeled graph, where works_in and customer_of are two labels between vertices 'Jane' and 'Bank'.](image)

Most existing graph databases use either adjacency matrix or adjacency list to represent edge-labeled graph, e.g. OrientDB, Neo4j, and MS Graph Engine. Theoretically, adjacency matrix has storage complexity of \( O(n^2) \), and can check connectivity in \( O(1) \). In contrast, adjacency list has storage complexity of \( O(n + m) \), and can check connectivity in \( O(\min\{|A_i|\} \leq O(d)) \), where \( A_i \) stands for the adjacency list for vertex \( u \), and \( d \) denotes the maximum degree in \( G \). We notice that the storage requirement of adjacency matrix grows squarely with the size of vertices, and thus is not an ideal solution for large-scale graph data. Thus, we adopt the adjacency list to represent the underlying graph data.

More concretely, to represent a string-format graph, we first use two string lists \( V \) and \( L \) to store the mappings between a valid string and its ID. That is, \( V : \Sigma^* \rightarrow \text{ind}(V) \) and \( L : \Sigma^* \rightarrow \text{ind}(L) \), where we use \( \Sigma^* \) to denote a valid string, and \( \text{ind}(\cdot) \) to denote the indexes of a certain type. Afterwards, we use indexed adjacency list to denote the relationships between vertex-IDs. For the example in Figure 1, let \( V = \{"Jane": v_1, "Bank": v_2\} \), \( L = \{"works_in": l_1, "customer_of": l_2\} \), the above example can be represented in the following indexed adjacency list:

\[
\begin{align*}
u_1 &: (l_1, v_2), (l_2, v_2), \vphantom{v_1} \\
u_2 &: \vphantom{v_1}
\end{align*}
\]

**2.3 Graph Query Functionalities**

Graph query languages express the searching functionality and serve as the core component of graph database systems. Currently, there are many practical graph query languages in the industry, e.g. Cypher, SPARQL, and Gremlin. Even though these languages differ enormously in purpose, expressivity, implementation, and so on. On the high level, graph query languages share two most fundamental functionalities, namely Graph Pattern Matching (GPM) and Graph Navigation (GN).

**2.3.1 Graph Pattern Matching (GPM).** In the theory of graph query languages, GPM refers to the problem of finding exact graph pattern matches for a given graph. Formally, we define an edge-labeled graph pattern as a graph \( \tilde{G} \) comprises of constants and variables, where constants are denoted as \( \text{Const}(G) \subseteq V \cup E \cup L \) and variables are denoted as \( \text{Var}(G) \). For instance, a graph pattern query “Search for the people that Marko knows” towards graph \( G = (V, E) \) could be turned into a graph pattern \( \tilde{G} = (\tilde{V}, \tilde{E}) \), where \( \tilde{V} = \{ "Marko" \}, \delta \), and \( \tilde{E} = \{ ("Marko", "knows", \delta) \} \), where \( \delta \) is the variable. To find a match, the graph pattern \( \tilde{G} \) is first matched to the graph \( G \), and then the graph database searches for the occurrences of the pattern. The GPM problem is known to be NP-hard theoretically by reduction from the graph homomorphism problem [31]. However, in reality, the size of the graph pattern query is usually much smaller than the size of graph database, and many existing works [2, 16, 43] have showed that the GPM problem can be solved in polynomial time and even subpolynomial time with a fixed-size query. More formally, a match for a graph pattern is defined as follows:

**Definition 2.2 (Match).** Given an edge-labeled graph \( G = (V, E) \) and a graph pattern \( \tilde{G} = (\tilde{V}, \tilde{E}) \), a match \( h \) of \( \tilde{G} \) in \( G \) is a mapping from \( \text{Const} \cup \text{Var} \) to Const such that the mapping \( h \) maps constants to themselves and variables to constants; if the image of \( \tilde{G} \) under \( h \) is contained within \( G \), then \( h \) is a match.

Technically, we follow Ullmann’s basic approach [56] to find the matches. It requires to list all possible mappings of vertices in the graph pattern and invoke a depth-first tree search algorithm. Since Ullmann’s algorithm has exponential search space, it is common to involve a pre-processing phase to prune unpromising subtrees. We denote the match search tree of graph \( G \) as Tree\((G)\), and show the details of Ullmann’s match algorithm in Figure 2.

![Figure 2: Ullmann’s match algorithm [56] for edge-labeled graph.](image)
the possible search tree for graph pattern $\tilde{G}$ and prune the repeated values. As the result, we obtain Figure 4 with a total of six possible matches.

![Graph data](image)

(a) A graph data

![GPM query](image)

(b) A GPM query with variable $\delta_1, \delta_2$

**Figure 3: An example of the GPM problem.**

To find the match of $\tilde{G}$ in the depth-first manner, we begin by looking at the first value of $\delta_1$ on level 1 (that is “Jim”). In the graph pattern $\tilde{G} = (V, E)$, we extract the edges containing $\delta_1$ only, that is, (“Jim”, “knows”, $\delta_1$). By substituting $\delta_1$ with its value “Jim”, we obtain (“Jim”, “knows”, “Jim”), which is not an edge in the original graph, therefore, we delete this node and its subtrees. After many iterations, the algorithm reaches the end, and the remaining paths are the matches. In this example, we have $\delta_1 = “Jane$” and $\delta_2 = “Bank”$ (the dash line in Figure 4).

![Matching tree](image)

**Figure 4: Possible matching tree.**

2.3.2 Graph Navigation (GN). In general, GN allows navigation towards the graph topology. GN queries have long been established as the core of navigational querying in graphs by the research community [7, 11, 57] and are widely adopted in graph query languages (e.g. SPARQL, Cypher and Gremlin). One typical example of such a query is “finding all friends-of-a-friend of Marko”. Here we are not only interested in the immediate acquaintances of a person, but also the people she might know through other people; namely, her friends-of-a-friend, their friends, and so on. Traditionally, the functionality of graph navigation uses a basic component called Path Query, where path query navigates through arbitrary number of edges in the graph. Formally, we define path query as follows.

**Definition 2.3 (Path Query).** Given an edge-labeled graph $G = (V, E)$, a path query has the general form $P = x \rightarrow^{\alpha} y$, where $x, y$ specify the beginning and the ending nodes (vertices) of the path, and $\alpha$ denotes the condition (traversal instruction) on the paths.

Generally, the evaluation of a path query $P = x \rightarrow^{\alpha} y$ over $G$, denoted as $P(G)$, consists of all the paths in $G$ who conform the condition $\alpha$. Due to the fact that the evaluation of a path query differs vastly from different systems, we focus on Gremlin’s processing system, and introduce the details in Section 2.3.3.

2.3.3 Graph Query Language Processing. Industrial graph query processing systems use different mechanisms and semantics to process GPM and GN queries. For instance, SPARQL [33] processes graph queries in a non-instructive and expressive fashion, while Gremlin [49] processes graph queries through graph traversal.

In this work, we focus on the more general framework — Gremlin’s graph traversal machine. A traversal machine in Gremlin is defined as $t \in T$, which traverses over a graph $G$ according to an instruction set $\Psi$. Here, we take a simplified formalization of a graph traversal machine for edge-labeled graph, that is,

$$T = (U \times \Psi),$$

where $U = V \cup E \cup L$ is the traverser’s location in the graph $G$. Evaluating a single step traversal instruction $\psi \in \Psi$ in Gremlin can be taken as a specification of one of the following maps:

1. **flatMap**
   - $P(V) \rightarrow P(E)$: Move the traverser from vertices to edges;
   - $P(E) \rightarrow P(V)$: Move the traverser from edges to vertices;
   - $P(U) \times G \rightarrow P(U)$: Move the traverser according to graph pattern $G$;

2. **filter**
   - $P(E) \times I_c \rightarrow P(E)$: Filter edges with the index set $I_c$;
   - $P(V) \times I_c \rightarrow P(V)$: Filter vertices with the index set $I_c$;
   - $P(E) \times L \rightarrow P(E)$: Filter edges with the label $l \in L$;

where we use $P(\ast)$ to denote the power set of $\ast$. Since we use a simplified representation for the graph traverser, it only supports the most basic graph traversal — Simple Traversal, where traversal instructions are processed in a sequential order ($\psi_1 \sim \psi_2 \sim \ldots \sim \psi_n$). For the rest of this paper, we use the term “instruction” or “traversal instruction” to short for a single step traversal instruction.

Following this intuition, we further detail the categorization of our graph query languages, namely single-instruction query and multi-instruction query, where a single-instruction query contains only one traversal instruction and multi-instruction query consists of multiple instructions. In Section 4 and Section 5, we will present how to securely evaluate single-instruction query and multi-instruction query on secret shared graph database, respectively.

3 GRAPH SHARING SCHEME

Though we specify the plaintext graph model and representation, it still remains a challenge of how to securely and efficiently share the graph between two parties. One the one side, the graph sharing scheme should be secure against a semi-honest adversary, and on the other side, it should also support: (1) secure validity check for a constant string (whether the string exists in vertices set or label set); (2) secure connectivity check for two string-format vertices, including the case of shared string vs. public string, shared string vs. shared string, and public string vs. public string. Here, shared means that the string is secret shared between $S_0, S_1$, and public means that the string is public to both $S_0, S_1$.

Intuitively, as is shown in the previous section, an edge-labeled graph $G$ could be sufficiently represented by a vertex list $V$, a label list $L$, and an adjacency list $A$. This graph representation method uses $V$ and $L$ for validity check of a string and retrieves the string’s index if it is valid. Afterwards, given two vertex-IDs, the adjacency list $A$ checks the connectivity of the two vertices.
In our solution, we adopt such design intuition and utilize two different data structures for the graph sharing scheme, namely shared lookup table and shared adjacency list. To begin with, we use hash(v) to represent hashed strings, where hash(*) is a cryptographic hash function. First, we hash all the strings in the original graph, and share the hashed values S = hash(V) ∪ hash(L), i.e., (⟨S0⟩, ⟨S1⟩) ← Shr((S, v)). The shared string list is later used for string’s validity checking. Then, to allow secure connectivity check of two vertices, we build a shared index lookup table [T], which allows index retrieving for a vertex string or a label string. Finally, we share the indexed adjacency list [A] to allow the connectivity check between two vertices. We will describe the details of shared lookup table [T] and shared adjacency list [A] later. In summary, a shared graph [G] is represented as:

\[ [G] = ([S], [T], [A]). \]

Shared lookup tables \([T]\). Formally, a shared lookup table associates with algorithms BuildT and LookupT. Let \(X = \{ x₀, ..., x_{d-1} \}\) and \(Y = \{ y₀, ..., y_{d-1} \}\) be two sets with the same length \(d\), and we want to share the mapping \(X \mapsto Y\). Additionally, we define the final result \([T]\) as a set of polynomial coefficients \([a₀], ..., [a_{d-1}]\), where the shared \([a_i]\) are the coefficients used to retrieve the mapping results. For sharing the graph, we build a shared index lookup table by \([T]\) ← BuildT(hash(V), ind(V)).

Notation. \(X = \{ x₀, ..., x_{d-1} \}\) and \(Y = \{ y₀, ..., y_{d-1} \}\) have \(d\) elements, and we also define \([T]\) = \([a₀], ..., [a_{d-1}]\).

Algorithm 1. \([T]\) ← BuildT(X, Y).

1. Data holder gets the polynomial coefficients from the input mapping: \(a₀, ..., a_{d-1} ← \text{LagrangeInterpolation}(X \mapsto Y)\);
2. For every \(a \in \{a₀, ..., a_{d-1}\}\), data holder shares the coefficients to servers: \([a] ← \text{Shr}((a, v))\).

Algorithm 2. \([y]\) ← LookupT(x, \([T]\)).

1. \(S_i\) calculates \([y] = [a₀]₀ + [a₁]₀x + ... + [a_{d-1}]₀x^{d-1}.\)

Figure 5: Algorithms for shared lookup table.

In Figure 5, we use the term “LagrangeInterpolation” to denote the function that outputs a polynomial (degree \(d - 1\)) that interpolates the input mapping. The resulting coefficients \(a₀, ..., a_{d-1}\) and the polynomial \(f(x) = \sum_{i=0}^{d-1} a_i x^i\) satisfies that \(f(x_i) = y_i\) for all \(x_i \in X\) and \(y_i \in Y\). The computation complexity of computing all coefficients of LagrangeInterpolation is \(O(d \log d)\) according to [54], and sharing the coefficients requires \(d\)\-bit communication, where \([x]\) indicates the bit size of the shares. As for the lookup algorithm, evaluating the polynomial only requires \(d\) additions and \(d\) multiplications using Horner’s Method [54], and lookup algorithm requires no extra communication cost.

Shared adjacency list \([A]\). The objective of a shared adjacency list is to allow the connectivity check for two vertices with a given label. A simple way is to reconstruct all the indexes and apply a plaintext lookup, but unfortunately this leaks access patterns. To share the adjacency list, first we use an represented index for the head node (the \(u\) of the \(A_u\) of the adjacency list, and then we include the tail node’s index in the edge representation, and pad every line in adjacency list to the maximum degree \(\hat{d}\) with “0”s. Finally, we share the mapped element in the adjacency list (notice that we do not need to share the head vertex indexes as long as they are stored in order). For example, the shared adjacency list of Figure 1’s case is:

\[ \text{ind}(a₀) : \langle [\text{ind}(a₀)]₀, [\text{ind}(a₁)]₀, [\text{ind}(a₂)]₀, [\text{ind}(a₃)]₀ \rangle \]
\[ \text{ind}(a₁) : \langle [\text{ind}(a₀)]₀, [\text{ind}(a₀)]₀, [\text{ind}(a₀)]₀, [\text{ind}(a₀)]₀ \rangle \]

where \(S₀\) holds the list with \(i = 0\) and \(S₁\) holds the list with \(i = 1\). This secret sharing scheme has storage complexity of \(O(\text{ind} + 2n + m)\) on both server’s size and checks connectivity of two vertices with \(O(\hat{d})\) secure equality test operations. We denote the adjacency list as \(A\), and shared adjacency list as \([A]\).

Security. This sharing scheme for graphs inevitably inherits the leakage profile from shared lookup table and shared adjacency list. All those leakages are leaked to servers \(S₀\) and \(S₁\). Specifically, \([S]\) leaks the total number of vertices and labels \(|V| + |L|\), \([T]\) leaks the number of vertices \(|V|\), and the shared adjacency list \([A]\) additionally leaks the maximum degree of the graph \(\hat{d}\). Here, we only discuss the leakage of those data structures, and we leave the security of connectivity check in Section 4. In summary, the leakage profile of this sharing scheme is defined as:

\[ \mathcal{L}_{\text{share}} = (|V|, |L|, \hat{d}). \]

4 SECURE EVALUATION OF A SINGLE-INSTRUCTION QUERY

In the previous section, we introduce how to convert GPM (graph pattern) queries and GN (path) queries into traversal instructions. In this section, we show the secure processing of single-instruction queries (e.g. a GPM query or a GN query which could be translated into only a single instruction). The security of our protocols follows the standard semi-honest definition using simulation-based technique [12, 28], where security says the behaviour of the adversary can be simulated given only the view of an honest participant.

Methodology. Aiming at secure and efficient single-instruction query evaluation, we first formalise the ideal functionalities of answering GPM and GN queries. We then present and analyse the secure constructions using real world vs. ideal world simulation paradigm. To allow an easy analysis, we describe functionalities with a leakage profile \(\mathcal{L}\), which models the information leakage. Also, some of our proposed protocols leverage a known secure equality test protocol in MPC. To give an abstraction of the whole protocol, we prove the security in the MPC-hybrid model, where we assume the presence of a secure equality test functionality\(\mathcal{E}_Q\). The detailed constructions are shown in [14].

4.1 Securely Evaluating A GPM Query

Recall that a GPM query is essentially a graph pattern that could be represented as a graph \(\hat{G} = (\hat{V}, \hat{E})\), containing constants and variables. Formally, we adopt the graph pattern matching algorithm from Ullmann[56], and describe the functionality of GPM as \(T_{\text{GPM}}\)
in Figure 6, where we use $\ln(x) \in X \times X^*$ to denote a
algorithm which checks if the element $x$ belongs to the set $X$. This
functionality works between $C, S_0$, and $S_1$ and allows the secure
evaluation of a graph pattern query (containing a single instruc-
tion) over the secret shared graph database. The ideal functionality
$\mathcal{F}_{GPM}$ could be split into three phases:

1. **Check phase**, where the client checks the validity of a given
   graph pattern by sending all strings in graph pattern to the
   ideal functionality;
2. **GetID phase**, where the client retrieves IDs for every query
   string;
3. **Query phase** where the client first builds Tree($\tilde{G}$), and then
   checks the existence of an edge in Tree($\tilde{G}$) in a depth-first-
   search manner.

**Global Parameters.** Security parameter $\lambda$.

**Check.** Given a hash set $Hset$ from $C$, and $Hset$'s, from $S_i$,
(1) Invokes $(S, T, A) \leftarrow \text{Rec}(\tilde{G})$;
(2) For all $\sigma_j \in Hset$, invokes $b_j \leftarrow \ln((\sigma_j, S)$.
Leaks $|Hset|$ to $S_i$;
(3) Returns $b_1, ..., b_{|Hset|}$ to $C$.

**GetID.** Given a hash set $Hset$ from $C$, and $Hset$'s, from $S_i$,
(1) Invokes $(S, T, A) \leftarrow \text{Rec}(\tilde{G})$;
(2) For all $\sigma_j \in Hset$, invokes $b_j \leftarrow \ln((\sigma_j, S)$.
Leaks $|Hset|$ to $S_i$;
(3) Returns $b_1, ..., b_{|Hset|}$ to $C$.

**Query.** Given a graph pattern $\tilde{G}$, client $C$:
(1) Initiates $Hset = \emptyset$;
(2) For all $\phi_j \in \text{Const}(\tilde{G})$, calculates and pushes $(\phi_j)$
to Hset;
(3) Sends Hset to servers $S_i$;
(4) On receiving $S_i$'s messages, invokes $\phi_j \leftarrow \text{Rec}(\phi_j)$
and $\phi_j \leftarrow \text{Rec}(\phi_j)$.

**Lemma 4.1.** Protocol $\Pi_{GPM}$ is secure against non-adaptive adversary
in the MPC-hybrid model.

**Proof sketch.** For the security proof of $\Pi_{GPM}$, we use a simulator
for each server in an invocation of $\Pi_{GPM}$, Check, $\Pi_{GPM}$, GetID and
$\Pi_{GPM}$, Query, and prove its security in the MPC-hybrid model.

**Simulator for $S_i$:** During the Check phase, $S_i$ eventually receives
nothing, therefore the simulation for Check phase is trivial. To
simulate the getID phase and query phase, the simulator uniformly
samples random IDs for every edge header $\phi_j$, and then obtains
the final result. This simulation is indistinguishable from the real
execution under the condition that the simulator cannot distinguish
between a randomly sampled vertex-ID and the real vertex-ID for
a specific string. Since we only assume a non-adaptive semi-
honest adversary and the real vertex-IDs are no-repeatable and
independently distributed, the simulation completes.

**Communication efficiency.** The check phase only runs in one
round and has communication complexity of $O(|\text{Const}(G)|)$ for $C$, $S_0$, and $S_1$. And the getID phase has communication complexity of
4.2 Securely Evaluating A Single-Instruction

GN Query

Recall that in Section 2.3.2, GN queries allow the navigation towards the topology of the graph. Similar to $\mathcal{F}_{\text{GPM}}$, the ideal functionality $\mathcal{F}_{\text{GN}}$ could be divided into three phases:

(1) Check phase, where the client checks if a rich-text query is valid. This phase is identical to $\mathcal{F}_{\text{GPM}}, \text{Check};$

(2) GetID phase, where the client retrieves IDs for every query string. This phase is identical to $\mathcal{F}_{\text{GPM}}, \text{GetID};$

(3) Query phase, which takes a shared traverser as input and outputs a new shared traverser based on the latest traversal instruction.

C.Query.filter Given a single-instruction path query with condition $\alpha$, $\mathcal{C}$:

(1) Sends $\text{ind}(U), \psi_{\gamma \rightarrow \alpha}$ or $\psi_{\gamma \leftarrow \alpha}$ to $S_i$

(2) On receiving server’s messages, invokes $b_1, ..., b_k \leftarrow \text{Rec}(\langle b_1 \mid 0, ..., b_k \mid 0, b_1 \mid 1, ..., b_k \mid 1 \rangle)$

$S_i$.Query.filter

On receiving $\text{ind}(U)$ and an instruction $\psi_{\gamma \rightarrow \alpha}$ from $\mathcal{C}$,

(1) Extracts current candidate set $\langle U_i \rangle \in \langle U \rangle_i$, and its corresponding shared adjacency list $\langle A_U \rangle_i$

(2) For all $\langle A_U \rangle_i \in \langle A_U \rangle_i$, filters and gets the indication vector for every edge in $\langle A_U \rangle_i$, that is $\langle b_1 \mid i, ..., b_{|U|d} \rangle_i \leftarrow \text{EQ}(\langle A_U \rangle_i, \phi_{\gamma \rightarrow \alpha})$

(3) Sends $\langle b_1 \mid i, ..., b_{|U|d} \rangle_i$ to $C$.

C.Query.flatMap Given a single-instruction path query with condition $\alpha$, $\mathcal{C}$:

(1) Sends $\text{ind}(U), \psi_{\gamma \rightarrow \alpha}$ or $\psi_{\gamma \leftarrow \alpha}$ to $S_i$

(2) On receiving server’s messages, invokes $b_1, ..., b_k \leftarrow \text{Rec}(\langle b_1 \mid 0, ..., b_k \mid 0, b_1 \mid 1, ..., b_k \mid 1 \rangle)$

$S_i$.Query.flatMap

On receiving $\text{ind}(U)$ and an instruction $\psi_{\gamma \rightarrow \alpha}$ from $\mathcal{C}$,

(1) Extracts current candidate set $\langle U_i \rangle \in \langle U \rangle_i$, and its corresponding shared adjacency list $\langle A_U \rangle_i$

(2) For all $\langle A_U \rangle_i \in \langle A_U \rangle_i$, filters and gets the indication vector for every edge in $\langle U \rangle_i$, that is $\langle b_1 \mid i, ..., b_{|U|d} \rangle_i \leftarrow \text{EQ}(\langle A_U \rangle_i, \phi_{\gamma \rightarrow \alpha})$

(3) Sends $\langle b_1 \mid i, ..., b_{|U|d} \rangle_i$ to $C$.
5 SECURE EVALUATION OF A MULTI-INSTRUCTION QUERY

Now that we have introduced how to securely evaluate a single-instruction query in Section 4. It still remains a question: can protocols πGPM and πCN preserve efficiency and security under the compositions of single instruction queries?

To answer this question, we first observe that secure Single Instruction Evaluation (SIE) protocols (πGPM.Query and πCN.Query) share a common structure. We show the general structure of SIE in Figure 13, where π.Query takes input of a single instruction (ΠGPM or ΠCN) and a traversal location set (not required for πGPM), and outputs a shared indicator vector [b] (when client and server do not invoke reconstruction function). For the case of SIE, [b] is first reconstructed to client, and client then filters the ind(G) (which equals to ind(V)∪ind(E)∪ind(L)), and finally gets the query result. However, this interactive approach is not desired when facing a multi-instruction query, since the whole evaluation process will involve O(|Ψ|) interactions between servers and client. Also such naive composition of SIE will have linear communication complexity, which also contradicts the goal (sub-linear) of this work. Moreover, reconstructing [b] to servers inevitably leak internal traversal locations. Therefore, how to perform filter in an oblivious manner, without reconstructing [b], is the key to achieve sublinear complexity.

![Figure 13: Single Instruction Evaluation (SIE).](image)

In this work, we propose a novel cryptographic primitive called Oblivious Filter, which we will present in details in Section 6. We denote the Single Instruction Evaluation with Oblivious Filter as OF-SIE, and show the framework of secure multi-instruction evaluation in Figure 14, where OF-SIE is sequentially executed. Informally, for a traverser t = (U, Ψ) with instruction set Ψ = {ψ1, ..., ψ|Ψ|}, the evaluation routine works as follows:

result ← OF-SIE(...OF-SIE(OF-SIE(t = (U, Ψ)))...),

where we denote the initial location as [U], and result location as [U′]. Since each shared location set for OF-SIE is a strict subset of ind(G), sub-linear communication complexity is guaranteed.

![Figure 14: Multi-Instruction Evaluation Framework.](image)

6 OBLIVIOUS FILTER

In this section, we present the building block for our proposed general query processing system, i.e., oblivious filter.

Definition. We define the oblivious filter primitive as a 2-party functionality between a server and a client. A server holds a list of pairs (t, v), and every pair consists of an indication bit ti ∈ {0, 1} and a fixed finite value vi ∈ V. The client holds a choice bit c ∈ {0, 1}. Intuitively, oblivious filter performs a secure filtering on the list of pairs (t, v), and outputs vi when client and server do not invoke reconstruction function. For the case of SIE, [b] is first reconstructed to client, and client then filters the ind(G) (which equals to ind(V)∪ind(E)∪ind(L)), and finally gets the query result.

Inputs. A list of m pairs (t, v) from server, where t ∈ {0, 1}m, v ∈ Vm. A choice bit vector c ∈ {0, 1}m from client.

Outputs (shared). [v′], s.t. v′ ⊆ v, and for every element v′ ∈ V, its corresponding ti equals to c.

![Figure 15: Functionality of Oblivious Filter.](image)

Relations with other primitives. Oblivious filter can be seen as a instantiation of Private Function Evaluation (PFE) [32, 35] with shared outputs. Moreover, one building block for PFE — Oblivious Extended Permutation (OEP) [40] could be seen as a relaxation of oblivious filter, where the filter function is held by client instead of both parties. Roughly, OEP assumes that server holds an extended permutation π : {1, ..., m} → {1, ..., n}, a mask r ∈ Vn, while client holds a private input v ∈ Vm. At the end of OEP protocol, client learns {vπ−1(1) + r1, ..., vπ−1(n) + rn}, and server holds {r1, ..., rn}. Though the design intuition is similar, oblivious filter takes a simpler construction from homomorphic encryption (other than universal circuits), which makes it more communication-efficient than the OEP protocols.

Additionally, oblivious filter can also be seen as a simpler version of Private Secret-Shared Set Intersection (PS3I) [10] and therefore has a more efficient construction comparing with PSI, which assumes two parties hold a list (t1, v1) and (t2, v2) respectively, and the protocol finally outputs the shared intersection of v′1 and v′2. Notice that for all v′ ∈ V and v′ ∈ V, lci = lci holds. PS3I ensures that at the end of the protocol, two parties only learn the shares the intersection and nothing else.

Our work is also related to other primitives, such as Oblivious Data Structures [51] and Oblivious Shuffling [15, 35, 41, 46]. In the literature, there are three approaches for PFE protocol construction [35]: homomorphic encryption, universal circuit, and oblivious switching network. We follow such design intuition and try to build oblivious filter based on HE.

Security of oblivious filter. Roughly, we define that an instantiation of oblivious filter should follow simulation-based security with the presence of a semi-honest adversary. That is, at the end of the protocol, server should know nothing about c and client should know nothing about (t, v). Formally, security holds that any party’s view during the attack can be simulated given only its own input and output.
Definition 6.1 (semi-honest security). Let $\Pi_{OF}$ be an oblivious filter protocol. We say that $\Pi_{OF}$ is secure against semi-honest adversary in the two-party settings if for all adversaries $A$, there exists a polynomial-time simulator $S$, such that

$$\Pr[\text{Rea}_{A}^{\Pi_{OF}}(\lambda) = 1] - \Pr[\text{Ideal}_{A}^{\Pi_{OF}}(\lambda) = 1] \leq \text{negl}(\lambda).$$

In the following of this section, we will present a construction of oblivious filter using HE, and then introduce a variant of oblivious filter, namely secret-shared oblivious filter.

6.1 Construction of OF from HE

We first present a simple construction of oblivious filter based on partially homomorphic encryption scheme. We present the description of this protocol (OF-HE) in Figure 16. The main idea is that server first performs a secure evaluation to shuffled indicator vector, without knowing additional information about $t$.

During the setup phase, both parties generate encryption key pairs by $(pk, sk) \leftarrow \text{Gen}(\lambda)$, and exchange public key with each other. At the beginning of the protocol, client first uses its public key to encrypt its choice vector $c$, and sends it to server. Then server performs secure evaluation between the encrypted $c$ and $t$, and gets the encrypted $c - t$. Afterwards, server generates a random permutation $\pi_{s} \in S^{m}$, and encrypts and shuffles the value $v$. After it, server sends it to client $(\pi_{s}(Enc_{pk}(c - t)), \pi_{s}(Enc_{pk}(v)))$.

Then client filters $Enc_{pk}(\pi_{s}(v))$ by decrypting the first part of server’s message $\pi_{s}(Enc_{pk}(c - t))$. As a result, client gets a new vector $Enc_{pk}(\pi_{s}(v'))$. Then client samples a new vector $r$ and gets $Enc_{pk}(v' - r)$. Next, client applies a new permutation $\pi_{c} \in S^{m}$ to the encrypted result and sends $Enc_{pk}(\pi_{c}(v' - r))$ back to server. After decryption, server sets $\|v'\|_{0} = v' - r$ and client sets $\|v\|_{1} = r$.

Lemma 6.2. The protocol described in Figure 16 is a secure instantiation for oblivious filter functionality.

The correctness of the simple construction could be easily verified. As for the security of this construction, we prove the security of OF-HE in Appendix B.

Now we analyse the complexity of this protocol.

- Round complexity: Constant-round and communicating $m+n$ ciphers in total.
- Computation complexity: Server performs $m+n$ public-key operations and Client performs $n$ times HE evaluations.

6.2 Secret-Shared Oblivious Filter

Since we use a shared input and shared output format in our querying framework, in this subsection, we present a variant of OF, i.e., Secret Shared Oblivious Filter (SS-OF) input. In this variant of OF, we assume that server and client both hold the shares of a list $(t, v)$, and they both know a public choice bit $c \in \{0, 1\}$. We formally describe this functionality in Figure 16. In the following, we will present two different construction of SS-OF.

6.2.1 Construction of SS-OF from Oblivious Filter. SS-OF functionality can be achieved from a two-fold oblivious filter (see Figure 18). For the first round, $P_{0}$ inputs $[t]_{0}$ and $[v]_{0}$ to $\Pi_{OF}$, and $P_{1}$ inputs $[1]_{1}$. Then $\Pi_{OF}$ outputs $[v'_{0}]_{0}$ to $P_{0}$, and $[v'_{1}]_{1}$ to $P_{1}$. For the second round, $P_{0}$ inputs $[v]_{0}$ to $\Pi_{OF}$, and $P_{1}$ inputs $[t]_{1}$. Then $\Pi_{OF}$ outputs $[v'_{0}]_{0}$ to $P_{0}$, and $[v'_{1}]_{1}$ to $P_{1}$. Finally, $\|v'_{0}\|_{0} + \|v'_{1}\|_{1}$ and $\|v'_{0}\|_{0} + \|v'_{1}\|_{1}$ are the reshares of the result $v'$.

6.2.2 Construction of SS-OF from Oblivious Shuffle. We borrow the idea from [41], and propose a generic construction of SS-OF from Oblivious Shuffle in Figure 19. The communication and computation complexities of this construction mainly depend on the oblivious shuffle protocol, and to the best of our knowledge, the most efficient secret-shared shuffle protocol [15] achieves $O(N \log N \cdot \lambda)$ communication, where $N$ is the set size and $\lambda$ is the security parameter. With an efficient secret-shared shuffle protocol such as [15], constructing SS-OF from oblivious shuffle is also efficient.

7 EXPERIMENT

We implement the additive homomorphic encryption based oblivious filter using Paillier encryption scheme, which is implemented using C++ with GMP library. Our test environment is Quad-Core.
Table 1: Running time of OF-HE by filtering 50% of client’s input data.

<table>
<thead>
<tr>
<th>Key size</th>
<th>1024 bit</th>
<th>2048 bit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Input data size</td>
<td></td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>1,000</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>1,000</td>
</tr>
<tr>
<td>Offline time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Server</td>
<td>0.366872</td>
<td>3.73717</td>
</tr>
<tr>
<td>Client</td>
<td>0.373998</td>
<td>3.74880</td>
</tr>
<tr>
<td>Online time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Server</td>
<td>0.062351</td>
<td>0.647129</td>
</tr>
<tr>
<td>Client</td>
<td>0.131378</td>
<td>1.28905</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Inputs (shared). A list of $m$ pairs of shares $(\{t\}, \{v\})$, where $t \in \{0, 1\}^m, v \in \{0, 1\}^m$. A public choice bit $c \in \{0, 1\}$.

Outputs (shared). $\{v'\}$, s.t. for all $v' \in v'$, its corresponding $t'$ equals to $c$.

Setup. $P_0$ runs $(pk, sk) \leftarrow \text{Gen}(1^k)$, and sends $pk$ to $P_1$.

Protocol.

1. $P_0$ and $P_1$ performs an OI protocol, where $P_0$ inputs $(\{t\})$ and $(\{v\})$. $P_1$ inputs $(\{t\})$.
2. At the end, $P_0$ receives share $(\{v'\})_0$ and $P_1$ receives share $(\{v'\})_1$.
3. Switch roles of $P_0$ and $P_1$, where $P_0$ inputs $(\{t\})$, and $P_1$ inputs $(\{t\})$.
4. At the end, $P_0$ receives share $(\{v'\})_0$ and $P_1$ receives share $(\{v'\})_1$.
5. $P_0$ calculates the reshared result as $\{v'\}_0 = (\{v'\})_0 + (\{v'\})_1$ and $P_1$ calculates the reshared result as $\{v'\}_1 = (\{v'\})_0 + (\{v'\})_1$.

Figure 18: Constructing SS-OF from OF.

(1) Both parties apply a secret-shared shuffling protocol to permute the original data $(\{\pi(t)\}, \{\pi(v)\}) \leftarrow \text{Permute}(\{\pi\}, (\{t\}, \{v\}))$, where the $i$-th permuted element is denoted as $(\{\pi(i)\}, \{\pi(i)'\})$.

(2) Both parties reconstruct the permuted vector $(\{\pi(t)\}$ of the resulting database $(\{\pi(t)\}, \{\pi(v)\})$

(3) Both parties keep the shares of $(\{\pi(i)\})$ for which the corresponding indicator bit $t(i) = c$.

Figure 19: Constructing SS-OF from oblivious shuffle.

Intel Core i5 2.40GHz CPU with 16G RAM, and we have run the test on different size of databases in LAN. Specifically in our experiment, our protocol can finish obivious filter on $10^5$ data within about 30 minutes. We report the experimental result of OF using HE (OF-HE) in Table 1, where we set the modulus to the 1,024 bit and 2,048 bit, respectively. From them, we can find that the evaluation time of OF is linear with data size on both server and client’s sides, which indicates its scalability.

8 CONCLUSION

In this paper, we focus on the problem of how to perform scalable and secure query on secret shared graph databases. To do this, we first summarized the queries on secret sharing graph database into two single instruction queries, i.e., Graph Pattern Matching (GPM) and Graph Navigation (GN), and a multi-instruction that is composed by single instruction queries. We then leveraged secure multiparty computation technique to securely evaluate GPM and GN. Next, we proposed a general framework for processing multi-instruction query and introduced a novel cryptographic primitive Oblivious Filter (OF) as a core building block. We constructed OF with homomorphic encryption and proved that our proposed framework has sub-linear complexity and is resilient to access-pattern attacks. Finally, empirical study demonstrated the efficiency of our proposed OF protocol.

REFERENCES


A PAILLIER ENCRYPTION

We introduce the paillier encryption scheme [48] which is used to construct oblivious filter. Paillier is an instantiation of additive homomorphic encryption with algorithms (Gen, Enc, Dec, Eval). We introduce the details of these algorithms below.

Key Generation. (pk, sk) ← Gen(1^λ): Taking a security parameter λ as input, generates the public key pk for encryption, and secret key sk for decryption.

- First, generate two large prime numbers with bit size equal to the security parameter q, p ← large prime, and ensure that gcd(pq, (p − 1)(q − 1)) = 1.
- Let n = pq and calculate λ = lcm(p − 1, q − 1).
- Randomly select a group generator for n^2, g ← Z_{n^2}^*, and ensure that n divides the order of g, if not, repeat this step.
- Calculate μ = (L(g^x mod n^2))^{-1} mod n, where L(x) = (x − 1)/n.
- Let pk = (n, g) and sk = (λ, μ).

Encryption. c ← Encpk(m): Taking a message m and public key pk as input, return the encrypted cipher c.

- Randomly select a number r ← Z_{n^2},
- Generate the ciphertext c = g^m r^n mod n^2.

Decryption. m ← Decsk(c): Taking a ciphertext c and secret key sk as input, return the decrypted message m.

- m = c^λ μ mod n.

Evaluation. c = Evalpk(c₀, c₁): Calculate c = c₀ c₁ mod n^2.

Here we do not illustrate the details of the correctness and the security properties of paillier, for those who are interested we refer to the original work [48]. Other additive homomorphic encryption schemes can be found in [22, 47].

B SECURITY PROOF OF LEMMA 6.2

In this part, we prove the security of Lemma 6.2 using real world vs. ideal world simulation-based technique [12, 28]. First, recall that

Lemma 6.2. The protocol described in Figure 16 is a secure instantiation for oblivious filter functionality against adaptive semi-honest adversary assuming the existence of a IND-CCA2 secure additive homomorphic encryption scheme.

To start the proof, we first introduce the real/ideal models for the construction of oblivious filter functionality from HE, which we denote as Π_{HE}. Since OF is a two-party functionality between
a server and a client, where the server holds the private list \((t, v)\), and the client holds the private choice vector \(c\), where \(t, c \in \{0, 1\}^n\), and \(v \in \mathcal{V}_m\). We denote \(\mathcal{V}\) as the value space.

**Definition B.1.** Let \(f = (f_s, f_c)\) be two functionality, we say a protocol \(\Pi\) securely computes \(f\) in the presence of static semi-honest adversaries if there exists a probabilistic polynomial-time algorithm \(S_s\) and \(S_c\) such that

\[
\{(S_s(1^\lambda, t, v, \llbracket v' \rrbracket_s), v)\} \equiv \{(\text{view}_s^\Pi(\lambda, t, c, \text{output}^\Pi(\lambda, t, c)))\},
\]

\[
\{(S_c(1^\lambda, c, \llbracket v' \rrbracket_c), v)\} \equiv \{(\text{view}_c^\Pi(\lambda, v, \text{output}^\Pi(\lambda, t, c)))\}.
\]

**IND-CCA2 Security.** Indistinguishability under adaptive Chosen Ciphertext Attack (IND-CCA2) is defined as an adversary game, where the adversary is given access to both encryption and decryption oracle. The game is defined as follows:

**Simulator** \(S_s\). To simulate the server in OF, extract the output size \(n\) from the ideal output \(\llbracket v' \rrbracket_s\). Then uniformly sample a boolean list \(c \leftarrow \{0, 1\}^m\), a random permutation \(\pi_s\), securely evaluate and send \(\pi_s(\text{Enc}_{pk_s}(c - t))\) and \(\pi_s(\text{Enc}_{pk_s}(v))\) to the client. This simulation is computationally indistinguishable from the real execution assuming the encryption scheme is IND-CCA2 secure. To this end, the simulation is finished since it only receives the filter result afterwards.

**Simulator** \(S_c\). To simulate the client in OF, first send \(\text{Enc}_{pk_c}(c)\) to the server. Afterwards, extract the output size \(n\) from the ideal output \(\llbracket v' \rrbracket_c\) and uniformly sample a list \(r \leftarrow \mathcal{V}_c\) and send to the client. This simulation is computationally indistinguishable from the real execution assuming the encryption scheme is IND-CCA2 secure.