Private Compound Wildcard Queries using Fully Homomorphic Encryption

Myungsun Kim, Hyung Tae Lee, San Ling, Benjamin Hong Meng Tan, and Huaxiong Wang

***Abstract*—Fully homomorphic encryption (FHE) brings a paradigm shift in cryptographic engineering by enabling us to resolve various unsolved problems. Among them, this work solves the problem to design a private database query (PDQ) protocol that supports *compound queries with wildcard conditions* on encrypted databases using FHE. More precisely, we consider a setting where clients outsource an encrypted database using FHE to a remote server, and later request results of compound queries including a wildcard search condition–given a set of attribute values $\{A_1, A_2, \ldots, A_n\}$ and a search pattern W, retrieve a set of all attribute values A_i's in which the pattern W occurs. To this end, we first develop an algorithm for testing whether an encrypted string contains an encrypted pattern without revealing any information of the pattern, taking auxiliary encryptions as additional inputs. Then, using this algorithm, we design PDQ protocols on encrypted databases, which support compound queries using wildcard search conditions. Finally, we demonstrate proof-of-concept implementation results of our protocols by exploiting single-instruction-multiple-data (SIMD) operations and multi-threading techniques.**

***Index Terms*—Private database queries, Wildcard pattern matching, Homomorphic encryption, Encrypted databases**

I. Introduction

PRIVATELY processing queries over remote databases is a primary requirement of many real-world applications. It reduces the risks of outsourcing by enforcing data privacy and enhances the economies of scale that can result from cloud computing. Thus, practitioners as well as academics have proposed various solutions to make it possible. A bit more formally, the problem is that a client outsources an encrypted database to a server, and later retrieves records in the database that satisfy the conditions of a query, while revealing only the descriptions of database structures. We call it the private database query (PDQ) problem.

For a PDQ solution to be successfully deployed in some real-world applications, it is desirable to support as many functionalities as a traditional unencrypted database. Particularly, it is preferable to allow users to retrieve records from a table without revealing information about the values of its contents while using more than one condition in their retrieval queries. Our concerns about richness of queries is justified by observing that those queries are basic but present in some of the most commonly used structured query language (SQL) queries in practice, as evident from the benchmark queries Q2, Q9 and Q20 provided by Transaction Processing Performance Council (TPC) [1, Section 2.4].

The most common way to solve the PDQ problem is with symmetric searchable encryption (SSE) [2], [3], [4]. Unfortunately, most SSE-based PDQ techniques (e.g., [5], [6], [7], [8], [9], [10]) are restricted to a small number of query types (e.g., conjunctive boolean query). Support for richer queries (e.g., [11], [12]) are available in the literature but these techniques do not meet our security requirements since additional information besides the database structure must be given to the server. For instance, with Chase and Shen's SSE-based pattern matching technique [11], as auxiliary information, each encrypted string has its suffix tree representation stored at the server. When an encrypted pattern is matched, the suffix tree is traversed and this step leaks the length of the encrypted pattern to the server.

Another way to resolve the PDQ problem is to combine heterogeneous cryptographic primitives, e.g., order-preserving encryption, additive homomorphic encryption, and symmetric key encryption. Typical examples of this approach are CryptDB [13] and Monomi [14]. Unfortunately, these solutions are tightly coupled to pre-fixed specific settings, they have no scalability if there are deviations from the specified scenarios. For example, even if an encryption scheme for a new type of query becomes available, it is not clear on how to integrate the scheme into the pre-existing system. Furthermore, these solutions reveal some information about the database contents after several queries.

The appearance of fully homomorphic encryption (FHE) [15] enables us to resolve the problem in a different manner, as FHE schemes allow arbitrary computations on encrypted data. Despite that, efficient realization of protocols supporting various types of queries is another issue and most FHE-based techniques (e.g., [16], [17], [18]) only consider simple boolean queries. Taking a different approach with FHE, Yasuda et al.'s work [19] and its functionality-enhanced version, presented by Saha and Koshiba [20], provide efficient PDQ protocols for wildcard pattern matching between an encrypted pattern and unencrypted records in the database. However, both techniques require additional interactions or heavy computations to be used in general PDQ settings.

This work focuses on designing, constructing and implementing a PDQ protocol supporting wildcard searches with multiple boolean connectives over encrypted databases us-

M. Kim is with the Department of Information Security, The University of Suwon, Hwaseong, South Korea. Email: msunkim@suwon.ac.kr

H. T. Lee is with the Division of Computer Science and Engineering, College of Engineering, Chonbuk National University, Jeonju, South Korea. Email: hyungtaelee@chonbuk.ac.kr

S. Ling, B. H. M. Tan, and H. Wang are with the Division of Mathematical Sciences, School of Physical & Mathematical Sciences, Nanyang Technological University, Singapore. Email: lingsan@ntu.edu.sg, TANH0199@e.ntu.edu.sg, hxwang@ntu.edu.sg

ing FHE schemes.[1] More precisely, we consider substring matching queries with patterns containing wildcards and exact equality queries with boolean connectives. Following the cryptographic literature on wildcard queries [11], [12], [19], due to concerns over performance, we do not consider every single wildcard character that is available to retrieval queries in traditional unencrypted databases. Rather, we restrict our attention to the following three wildcard characters, \$, which stands for a single (non-empty) character, $!(\alpha)$ which stands for a single (non-empty) character that is not α and % which is used to denote either substrings ending ($\%W$), containing ($\%W\%$) or beginning ($W\%$) with query pattern W that may include several \$ and $!(\alpha)$ wildcards. Other wildcard characters that are not considered in this work include [] and $-$, which are for sets of characters.[2] For example, "%sp\$\$e%" is a valid wildcard pattern in this work that would match to "spice", "hospice", "space", "spaceship", "spore" and many more. Should we wish to filter out "space", "spaceship" and other words that include "spa", we can use the pattern "%sp!(a)\$e%".

A. Problem Formulation

Let $\mathsf{R}(\mathsf{A}_1, \mathsf{A}_2, \ldots, \mathsf{A}_\rho)$ be a relation schema (or simply a table) of $\deg(\mathsf{R}) = \rho$. In this work, we would like to privately handle compound queries like conjunctive and disjunctive queries, with search conditions that include some wildcards $\big(\%, \$ \text{ and } !(\cdot)\big)$. For $j, k, \ell \in \{1, \ldots, \rho\}$, a typical example of such queries is given by

$$\texttt{select } \mathsf{A}_\ell \texttt{ from } \mathsf{R} \texttt{ where } \mathsf{A}_j \texttt{ like } \text{“}\%W\%\text{”}^3 \texttt{ and } \mathsf{A}_k = Q,$$

where W is a pattern for wildcard matching test, possibly containing several \$ and $\{!(c_j)\}$ wildcards and Q is a constant for equality test. Throughout the paper, unless stated explicitly otherwise, attributes for wildcard matching with a pattern string W of length τ are also declared with a string variable of maximum length η. Note that τ and η mean the number of characters in strings and not the number of bits. To simplify the exposition, we restrict the alphabet to the set of characters of 8-bit length in this paper. We note that this restriction is arbitrary and can be easily replaced by any reasonable alphabet. In this paper, we aim to keep the size (τ) of input patterns as well as attribute values secret, but we allow the maximum length (η) of possible attribute values to be revealed.

B. Overview of Results

Our technical results are summarized as follows (we describe each in more detail below):

1) We provide a wildcard pattern matching algorithm that takes an encrypted string and an encrypted pattern along with auxiliary inputs.
2) On top of this, we present the first FHE-based PDQ protocol in the semi-honest model that supports compound queries on encrypted databases with wildcard conditions.
3) We finally provide experimental results of our solutions by applying several techniques to improve performance.

Wildcard Pattern Matching on Encryptions. We first develop a pattern matching algorithm that takes an encrypted string and an encrypted pattern along with auxiliary encryptions as inputs and returns an encryption of 1 if the input pattern occurs in the input string and an encryption of 0 otherwise. We stress that our wildcard matching algorithm can be easily integrated into a high-level protocol processing compound queries with wildcard search conditions, since our algorithm outputs an encryption of 1 or 0 to represent the result of the matching. On the other hand, in the previous work by Yasuda et al. [19] and its variant [20], the client receives an encrypted polynomial whose coefficients include 0 if the input pattern occurs in the input string and do not include 0 otherwise. This precludes their protocols from being applied for more general purposes, e.g., PDQ settings, without additional heavy processing. (See Section II for the details.)

An Extension to Compound Wildcard Query. By exploiting the proposed algorithm, we present the first FHE-based PDQ protocol in the semi-honest model that supports compound queries on encrypted databases with wildcard conditions. To the best of our knowledge, such a protocol has not been reported in the PDQ community, though the definition of FHE naturally implies its feasibility. Especially, to the best of our knowledge, our proposed protocol is the first PDQ protocol of any kind to support some type of exclusion wildcard, $!(\cdot)$.

To encrypt a string character-wise, we use an FHE scheme of plaintext space $\mathbb{F}_{2^8}$ whose security relies on learning with errors on rings (ring-LWE).[4] In this setting, the proposed wildcard matching algorithm requires a multiplicative depth of $4+2\lceil\log\eta\rceil$. The multiplicative depth required for the conjunctive query with wildcard and exact matching conditions is up to $4 + 2\lceil\log\eta\rceil + \lceil\log(\delta+1)\rceil$, where δ is the number of connectives in the conjunctive query. The transmission cost on the client side is at most $3\delta\eta$ FHE ciphertexts of characters, while the server needs to return $n\eta$ FHE ciphertexts of characters where n is the maximum number of tuples of η characters in the table R.

Implementation and Micro-Benchmarks. To illustrate the potential of the protocol, we provide an implementation of our protocol with several techniques to improve performance, such as careful packing of words into plaintext slots, single-instruction-multiple-data (SIMD) operations, and depth-free automorphism evaluations. Benchmarks for the prototype show that our efforts effectively boosted the performance of our protocol. According to the experimental results, our

[1] Specifically, we exploit leveled FHE scheme (e.g., [21]), but will omit the term "leveled" in the rest of this work.

[2] [] stands for any single character within the brackets and $-$ stands for any single character in a range of characters.

[3] "$\%W$" and "$W\%$" are the other possible wildcard patterns with %. We note that the work in this paper can also support these patterns, except the case that W ends with an exclusion $!(\cdot)$ for the pattern "$\%W$".

[4] Since the alphabet has been restricted to the set of characters of 8-bit length in this paper, $\mathbb{F}_{2^8}$ is an appropriate choice for the size of plaintext space, but it is easily adjustable with respect to the alphabet. We note that the required multiplicative depth for the proposed algorithm should be also adjusted to $2\lceil\log\eta\rceil + \lceil\log\ell\rceil + 1$, when the plaintext space of the exploited FHE scheme is $\mathbb{F}_{2^\ell}$ and the maximum length of attributes is η characters of ℓ-bit length.

protocol takes amortized time of about 2.3 seconds per element to perform a conjunctive query consisting of one wildcard matching condition and 5 exact equality conditions on 35-byte attributes by exploiting 29-level Brakerski-Gentry-Vaikuntanathan (BGV) FHE scheme [21] for 83-bit security with 24 threads (see Section V).

C. Solution Sketch

Before explaining our protocol, we introduce some basic notation. We use a (ring-LWE based) FHE scheme of plaintext space $\mathbb{F}_{2^8}$ and encrypt a string character-wise. Denoting its encryption algorithm by Enc, $\overline{m}$ refers to an FHE ciphertext of a message m, i.e., $\overline{m} = \mathsf{Enc}(m)$ omitting the public key and randomness used. Let $\mathtt{EQ}$ be an algorithm that takes two ciphertexts as inputs and outputs $\overline{1}$ if the two ciphertexts encrypt the same plaintext; otherwise $\overline{0}$.

In the setup phase, a client stores an encrypted table $\langle \mathsf{R}(\mathsf{A}_1, \ldots, \mathsf{A}_\rho), \{\overline{A}_1, \ldots, \overline{A}_n\}\rangle$ at the server side, where $\overline{A}_i$ is a ρ-tuple of encrypted attribute values. While defining a string-typed attribute A_j for some $j \in \{1, \ldots, \rho\}$, for simplicity, we restrict the alphabet Σ to the set of ASCII characters. Thus, each string is assumed to be a sequence of at most length η over Σ.

To simply our exposition, we consider a query with only one wildcard condition "$\%W\%$"[5], which can be written in SQL as follows:

$\mathtt{select}\ \mathsf{A}_\ell\ \mathtt{from}\ \mathsf{R}\ \mathtt{where}\ \mathsf{A}_j\ \mathtt{like}$ "$\%W\%$".

The key idea to handle wildcard queries is quite simple: We observe that a wildcard test can be regarded as a disjunction of results of (non-)equality tests between all possible substrings of the value A in tuples for attribute A_j and a given pattern W. More precisely, let $\overline{A} = (\overline{a}_1, \ldots, \overline{a}_\eta)$ be an encrypted string of η characters a_i's and $\overline{W} = (\overline{w}_1, \ldots, \overline{w}_\tau)$ be a pattern of length τ ($\leq \eta$). Note that no a_i can be one of the wildcard characters $!(c)$, \$ and %. Then, a wildcard test between $\overline{A}$ and $\overline{W}$ can be performed as follows:

1) Check if a τ-character substring $(\overline{a}_{i+1}, \ldots, \overline{a}_{i+\tau})$ contains pattern $(\overline{w}_1, \ldots, \overline{w}_\tau)$ for each $0 \leq i \leq \eta - \tau$.
2) Gather all results of the checks on each substring and check if any of them contain the pattern W.

Roughly speaking, the first step is obtained from a product of equalities and possibly non-equalities

$$\overline{z}_i = \prod_{j \in S} \Big(\mathtt{EQ}(\overline{a}_{i+j}, \overline{w}_j) + \mathtt{EQ}(\overline{w}_j, \$)\Big) \times \prod_{j \notin S} \Big(\mathtt{EQ}(\overline{a}_{i+j}, \overline{c}_j) + \overline{1}\Big) \quad (1)$$

for $0 \leq i \leq \eta - \tau$, where the set $S \subseteq \{1, 2, \ldots, \tau\}$ consists of positions j's between 1 and τ such that j-th position is not an exclusion wildcard. This means that for $1 \leq j \leq \tau$ we are performing equality tests if w_j is either an ordinary character or \$ and non-equality tests if position j has an exclusion wildcard.

[5]Refer to Section IV-A for the techniques to handle the patterns $\%W$ and $W\%$.

On top of this simple pattern test, we should look to support any number of wildcard characters $\big($denoted by \$ and $!(c)\big)$ and hide τ. Hiding all information about the pattern in the computation above means that we must handle three issues: (1) any number of wildcard characters can be included in the pattern W, (2) the non-equality tests that come with using the exclusion wildcard $!(\cdot)$ should be integrated with the equality tests, and (3) τ should not be revealed. To address the first issue, we re-purpose unused values in the plaintext space to represent \$ and encode W as a η-character string by padding $(\eta - \tau)$ \$'s after w_τ (see Section IV-A). The second issue is not that difficult, instead of setting w_j to be $!(c_j)$, we set $w_j = c_j$ so that we can perform the equality tests on every slot. Then, the exclusion wildcard is handled along with some auxiliary information which we use to address the last issue as well.

Our idea in tackling the last issue (i.e., hiding the length of a wildcard pattern) is to introduce a set of constants e_j's to represent the presence (or absence) of a wildcard character in each slot, which can be done by the client as this information is known to him or her. In our protocol, the client additionally provides encrypted constants $\overline{e}_j$'s to the server where $\overline{e}_j = \overline{1}$ if $\overline{w}_j = \overline{\$}$ or $!(c_j)$ and $\overline{e}_j = \overline{0}$, otherwise. When given an encrypted string $\overline{A} = (\overline{a}_1, \ldots, \overline{a}_\eta)$, the server sets $\overline{A}^{(i)} = (\overline{a}_{i+1}, \ldots, \overline{a}_\eta, \overline{\#}, \ldots, \overline{\#})$ for $1 \leq i \leq \eta$ by shifting i characters of $\overline{A}$ to the left and adding i $\overline{\#}$'s to the right, where $\#$ is a character that represents an empty slot. Denote by $\overline{A}^{(i)} = (\overline{a}_1^{(i)}, \ldots, \overline{a}_\eta^{(i)})$ for notational convenience. Then, without the information of τ, the server can compute Equation (1) by evaluating

$$\overline{z}_i = \prod_{j=1}^{\eta} \Big(\mathtt{EQ}(\overline{a}_j^{(i)}, \overline{w}_j) + \overline{e}_j\Big)$$

which corresponds to the result of checking if a τ-character substring $(a_{i+1}, \ldots, a_{i+\tau})$ contains W or not.

Next, to consolidate the results of the (non-)equalities in the second step, we introduce another set of constants d_i's such that $d_i = 1$ if $0 \leq i \leq \eta - \tau$ and 0 if $\eta - \tau + 1 \leq i \leq \eta - 1$. Similarly, we let the client send encrypted d_i's to the server and make the server compute

$$\overline{t} = \overline{1} + \prod_{i=0}^{\eta-1} (\overline{1} + \overline{z}_i \cdot \overline{d}_i).$$

As a result, $\overline{t}$ is an encryption of $\overline{1}$ if the pattern W occurs within the attribute value A and an encryption of $\overline{0}$, otherwise.

We can apply this idea to deal with compound queries with wildcard conditions by utilizing the result $\overline{t}$, since the result $\overline{t}$ directly indicates whether an attribute value A contains the pattern W or not.

D. Organization of the Paper

Section II introduces previous results that are related to wildcard queries on encrypted data. Following that, preliminary definitions and elementary tools for our constructions are given in Section III. Then, our PDQ protocols are given in Section IV along with detailed analysis and finally, we report on our implementation results in Section V.

II. Related Work

The goal of this section is to review concurrent work related to wildcard queries on encrypted data, which we classify according to the cryptographic primitive used. Although there are many proposals for wildcard queries with \$ and %, we note that none of them support exclusion wildcards, either for single character or sets, at all.

Homomorphic Encryption. There is a lot of prior research on the construction of private query protocols that exploit the properties of homomorphic encryption with examples including [16], [17], [22], [23]. However, they only deal with simple retrieval queries, which is one main difference from our proposal.

The most closely related works to ours is the private wildcard pattern matching protocol proposed by Yasuda et al. [19] and its functionality-enhanced version [20]. They use a ring-LWE based somewhat homomorphic encryption scheme whose plaintext space is a polynomial quotient ring $R_q = \mathbb{Z}_q[x]/\langle x^N+1\rangle$, where N is a power of 2 and q is a sufficiently large positive integer. In the protocol proposed by Yasuda et al., they first introduce two types of encodings for a vector $A = (a_1, \ldots, a_\eta) \in \Sigma^\eta$ by borrowing the packing method of [24]:

$$\begin{aligned} p_1(A) &= \sum_{i=0}^{\eta-1} a_{i+1}x^i \in R_q, \\ p_2(A) &= \sum_{i=0}^{\eta-1} a_{i+1}x^{N-i} = a_1 - \sum_{i=1}^{\eta-1} a_{i+1}x^{N-i} \in R_q. \end{aligned}$$

The client sends encrypted polynomials $\overline{p_2(W)}$, $\overline{p_2(W^2)}$, and $\overline{p_2(W^3)}$ where $W = (w_1, \ldots, w_\tau) \in \Sigma^\tau$ is a pattern of length τ, $W^2 = (w_1^2, \ldots, w_\tau^2)$, and $W^3 = (w_1^3, \ldots, w_\tau^3)$. Once received from the client, the server who stores a clear attribute $A = (a_1, \ldots, a_\eta) \in \Sigma^\eta$ of a length η, computes

$$\begin{aligned} \overline{R} = p_1(A) \times \overline{p_2(W^3)} \\ + p_1(A^3) \times \overline{p_2(W)} + (-2p_1(A^2)) \times \overline{p_2(W^2)} \end{aligned} \quad (2)$$

where $A^2 = (a_1^2, \ldots, a_\eta^2)$ and $A^3 = (a_1^3, \ldots, a_\eta^3)$. Then, the server returns the result of Equation (2). We note that coefficients of x^i in the resulting polynomial R for $0 \le i \le \eta - \tau$ is equal to

$$\sum_{j=0}^{\tau-1} w_{j+1} \cdot a_{i+j+1}(w_{j+1} - a_{i+j+1})^2$$

which is 0 if $A^{(i)} = (a_1^{(i)}, \ldots, a_\eta^{(i)}) = (a_{i+1}, \ldots, a_\eta, \#, \ldots, \#)$ has a pattern W and non-zero, otherwise. This is because at least one of w_{j+1}, a_{i+j+1}, and $w_{j+1} - a_{i+j+1}$ is 0 when w_{j+1} or a_{i+j+1} is the wildcard character \$, which corresponds to $0 \in \Sigma$, or $w_{j+1} = a_{i+j+1}$. Thus, the client confirms whether A contains a pattern W by decrypting $\overline{R}$ and then checking coefficients of x^i in R for $0 \le i \le \eta - \tau$. As a result, Yasuda et al. [19] provide a very efficient wildcard pattern matching protocol that requires several ciphertext additions and scalar multiplications only.

We may try to extend their protocol to our setting by storing encrypted polynomials $\overline{p_1(A)}$, $\overline{p_1(A^2)}$, and $\overline{p_1(A^3)}$ at the server and computing Equation (2) with them rather than the plaintext polynomials $p_1(A)$, $p_1(A^2)$, and $p_1(A^3)$. However, to perform compound queries with wildcard search conditions, the server has to obtain an encrypted result of the wildcard pattern matching for further computation. Thus, the server has to perform additional operations to check whether the resulting polynomial R includes a monomial with zero coefficient for some monomial x^i, $0 \le i \le \eta - \tau$. To complete it without knowledge of the length of the pattern, τ, the server may exploit expensive operations on encrypted data, such as extracting the coefficients of an encrypted polynomial of high degree ($N-1$ in this case). Hence, the straightforward extension of their protocol [19] is not suitable in our setting where compound queries, in addition to simple queries, are handled. Furthermore, the correctness of their protocol relies on the fact that the quotient polynomial of the plaintext space is of form $x^N + 1$, so existing packing and SIMD techniques for ring-LWE based FHE schemes cannot be easily applied to the modified protocol. This is because plaintext spaces of all currently existing packing techniques for ring-LWE based FHE schemes are $\mathbb{Z}_q[x]/\langle f_i(x)\rangle$ where $f_i(x)$ is a factor of $x^N + 1$ or the N-th cyclotomic polynomial, not $x^N + 1$ itself and is most likely not of the form $x^k + 1$ for some integer k. Thus, each attribute value should occupy one ciphertext and the modified protocol suffers from both space and computational inefficiencies, because N should be quite large for security reasons, but the length of each attribute is much smaller than N in general PDQ settings.

Very recently, Saha and Koshiba [20] presented an improved version of Yasuda et al.'s protocol that supports to replace several letters by one wildcard \$, whereas Yasuda et al.'s original work allows to replace a single letter by a single wildcard \$. They obtained this improvement by proposing a new packing method. Unfortunately, the same problems as Yasuda et al.'s protocol surface when their protocol is extended to our PDQ setting; the server has to perform additional heavy operations to check the coefficients of the encrypted resulting polynomial and existing packing and SIMD techniques cannot be easily applied.

Symmetric Searchable Encryption. An immediate solution for our goal is based on symmetric searchable encryption (SSE). SSE allows a user to retrieve indices for records matching a given keyword after storing data at an untrusted server, while preserving privacy. Since Song et al.'s first suggestion [2], lots of research has considered SSE to enrich functionality (e.g., [3], [4], [5], [6], [7], [8], [9], [10]). Faber et al. [12] recently proposed a substring query protocol on the top of Cash et al.'s SSE [9]. The scheme may not hide the length of substrings in a query and may have worse security; if its underlying SSE scheme is compromised by the file-injection attack investigated by Zhang et al. at USENIX 2016 [25], then the server can learn a significant portion of queried strings.

Fortunately, the above is not the only possible design; Chase and Shen [11] presented a substring query protocol using a different approach. In their scheme, a user first represents each string as a suffix tree [26], [27] whose nodes include suffix substrings, their length and position. Then, the user encrypts

and stores the tree at a server. Later, when the user runs a wildcard query by interacting with the server, the user sends encryptions of all prefixes of his queried string to the server, which is then searched for any match over the suffix trees using additional information stored in each node. This scheme reveals the length of keyword strings and the determinism of its encryption and extra information to support matching could allow an adversary to mount Zhang et al.'s file-injection attacks.

Predicate Encryption. Besides the few common techniques mentioned above, there are other approaches for substring search. For example, an inner product encryption (IPE) scheme (e.g., [28]) or a hidden vector encryption scheme (e.g., [5]) can be applied to this setting. Consider this scenario: Strings of length n are encrypted with an IPE scheme and stored in a cloud. A client wishes to perform a query on the strings with wildcard conditions and submits a pattern encrypted with an IPE scheme. For the cloud to evaluate a predicate at encryptions of a string and a pattern, the client has to provide a set of tokens derived from its secret key. In this case, the client needs to prepare $O(2^n)$ tokens because a string of length n has 2^n substrings, which leads to query times that are $O(2^n)$. Therefore, we think that this approach is clearly impractical.

Secure Two-Party Computation. There are several previous works on secure two-party (or multi-party) computation, particularly on secure substring matching and other string processing operations in the two party-setting (e.g., [29], [30], [31], [32]). Their approaches are interesting, but they work in different settings from ours: Ours has Alice outsourcing storage of her encrypted databases to Bob and then querying its table with an encrypted pattern string at any time she wishes, whereas in their settings, Bob works on an unencrypted string. This is the reason why we cannot directly apply secure multi-party computation to our setting.

III. Preliminaries

In this section, we recall some fundamentals of FHE and databases, along with an FHE-based primitive for equality test (see Section III-B). On top of those, we will design our algorithms (see Section IV-A) for wildcard pattern matching and PDQ protocols (see Section IV-B).

We begin by introducing notation used in the paper.

Notation. The symbol Σ denotes the set of ASCII characters of 8-bit length. We use uppercase letters to denote a string over Σ and lowercase letters to denote characters in Σ. If S is a character string over Σ, then s_i denotes the i-th character of S. Throughout the paper, $\overline{S}$ (resp. $\overline{s}$) denotes an encryption of a string S (resp. a character s) and we assume that a string S is encrypted character-wise, i.e., $\overline{S} = (\overline{s}_1, \ldots, \overline{s}_\eta)$ for $S = (s_1, \ldots, s_\eta)$ with characters s_i's.

For an algorithm Alg, $a \leftarrow \mathsf{Alg}$ denotes that the algorithm Alg outputs a. We say that a function $f(\lambda)$ is negligible if it is $O(\lambda^{-\alpha})$ for all $\alpha > 0$, and we use $\mathsf{negl}(\lambda)$ to denote a negligible function of λ.

A. Fully Homomorphic Encryption

Definitions. An FHE scheme consists of the following four probabilistic polynomial-time (PPT) algorithms, $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$:

- $(pk, sk) \leftarrow \mathsf{KeyGen}(1^\lambda)$: It takes a security parameter λ as an input and outputs a public key pk and a secret key sk. We assume that a public key pk specifies both the plaintext space $\mathcal{P}$ and the ciphertext space $\mathcal{C}$.
- $\overline{m} \leftarrow \mathsf{Enc}(pk, m)$: Given the public key pk and a plaintext m, it outputs a ciphertext $\overline{m}$. For simplicity, we omit the randomness used for encryption.
- $m' \leftarrow \mathsf{Dec}(sk, \overline{m})$: Given the secret key sk and a ciphertext $\overline{m}$, it outputs a message m'.
- $\overline{m}_\varphi \leftarrow \mathsf{Eval}(pk, \varphi, \overline{m}_1, \ldots, \overline{m}_n)$: It takes the public key pk, a function $\varphi : \mathcal{P}^n \to \mathcal{P}$, and a set of n ciphertexts $\overline{m}_1, \ldots, \overline{m}_n$ as inputs and outputs a ciphertext $\overline{m}_\varphi$.

For correctness, we require that for any $m_i \in \mathcal{P}$ with $1 \le i \le n$ and any function $\varphi : \mathcal{P}^n \to \mathcal{P}$, if $(pk, sk) \leftarrow \mathsf{KeyGen}(\lambda)$, $\overline{m}_i \leftarrow \mathsf{Enc}(pk, m_i)$ for $1 \le i \le n$, and $\overline{m}_\varphi \leftarrow \mathsf{Eval}(pk, \varphi, \overline{m}_1, \ldots, \overline{m}_n)$ then it always holds that $\mathsf{Dec}(sk, \overline{m}_\varphi) = \varphi(m_1, \ldots, m_n)$. For security, we say that an FHE scheme is semantically secure if for any PPT adversary $\mathcal{A}$,

$$\Big| \Pr[\mathcal{A}(pk, \mathsf{Enc}(pk, m_0)) = 1] - \Pr[\mathcal{A}(pk, \mathsf{Enc}(pk, m_1)) = 1] \Big|$$

is negligible in the security parameter λ where $m_0, m_1 \in \mathcal{P}$ are chosen by $\mathcal{A}$ and $(pk, sk) \leftarrow \mathsf{KeyGen}(1^\lambda)$.

Additionally, a leveled FHE scheme is an encryption scheme that only allows evaluations of functions of bounded depth (L). In leveled FHE schemes, a level L is additionally given to the key generation algorithm as an input. Then, for correctness, φ is restricted to a function which can be evaluated with a circuit with depth at most L. Strictly, a leveled FHE scheme is somewhat homomorphic encryption, not FHE. However, in the FHE literature we use the term, "leveled FHE", without precise classification. See [15], [21] for the formal definition of leveled FHE.

Choosing a Baseline FHE Scheme. For the scope of our protocol, the full power of FHE to compute arbitrary functions is not required. Instead, we use a leveled FHE scheme (e.g. [21], [33], [34]) which brings significant performance improvements compared to Gentry's FHE scheme [15] and its variants. Among them, our protocol can work on any ring-LWE based FHE schemes (e.g., [21], [33]) which support depth-free Frobenius map evaluations. Because we only need the high-level operations that FHE schemes provide to understand this work, we omit detailed descriptions of the schemes.

We end this subsection with a remark about plaintext spaces of existing ring-LWE based FHE schemes, since they are closely related to packing techniques and SIMD operations. In the existing ring-LWE based FHE schemes, the plaintext space is $R_p = R/pR = \mathbb{Z}[x]/\langle \Phi_m(x), p \rangle$ for a positive integer $p \ge 2$ where $R = \mathbb{Z}[x]/\langle \Phi_m(x) \rangle$ and $\Phi_m(x)$ is the m-th cyclotomic polynomial. Here, since $\Phi_m(x)$ factors modulo p into $\phi(m)/l$ irreducible factors $\{f_i(x)\}_{i=1}^{\phi(m)/l}$, all of degree l, R_p can be decomposed into $\prod_{i=1}^{\phi(m)/l} \mathbb{Z}[x]/\langle f_i(x), p \rangle$ which is

each isomorphic to a field $\mathbb{F}_{p^l}$ if p is a prime. This enables SIMD operations and depth-free automorphism evaluations by packing a message into each slot which corresponds to a factor of R_p. Unlike typical ring-LWE based FHE schemes which set $p = 2$ and pack a bit into each slot, in this paper we pack a character into each slot. See [21], [33], [35], [36] for concrete ring-LWE based FHE schemes and their packing methods.

B. Equality Test on FHE Ciphertexts

In our proposed algorithms and protocols, we will exploit an equality test algorithm on encrypted data using FHE schemes as a building block. We define an equality test algorithm between two encrypted strings as follows: For strings S_1, S_2 of the same length, $\overline{1} \leftarrow \texttt{EQ}(\overline{S}_1, \overline{S}_2)$ if $S_1 = S_2$; otherwise $\overline{0} \leftarrow \texttt{EQ}(\overline{S}_1, \overline{S}_2)$.

There are several previous works [18], [22], [37] that measure the required multiplicative depth to perform the `EQ` algorithm on l-bit encrypted elements. According to the analysis by Kim et al. [18], when the plaintext space of the FHE is $\mathbb{Z}_2$ or a binary extension field where the scheme supports depth-free Frobenius map evaluation, the `EQ` algorithm on l-bit encrypted elements consumes an optimal multiplicative depth of $\lceil \log_2 l \rceil$. Hence, we employ FHE schemes, whose plaintext space is $\mathbb{F}_{2^8}$ and support depth-free Frobenius map evaluation, to encrypt a string character-wise. We note that the `EQ` algorithm on two encrypted characters in our setting consumes $3 (= \log_2 8)$ multiplicative depth. Readers may refer to [18], [22], [37] for concrete `EQ` algorithms.

C. Database Structures

In this work, we only consider a database with a single table R without loss of generality. This is because we may consider to simply join the results from multiple tables if that is the case, but unfortunately there are currently no known efficient methods for joining tables encrypted with an FHE scheme.

Formally, we write the table R as the relation schema $\mathsf{R}(\mathsf{A}_1, \ldots, \mathsf{A}_\rho)$ of $\deg(\mathsf{R}) = \rho$. Without loss of generality, we assume that attributes A_j in R are fixed-length strings of length η with alphabet Σ. If the strings are less than η characters, we pad them with a special character $\# \in \Sigma$ that represents empty slots. It may seem that inserting string-typed values requires additional care because the alphabet Σ includes four reserved characters– #, %, ! and $. A pattern with wildcard conditions may hold one or more $'s, for example, W = "sp!(a)$e" for a wildcard condition "%W%" would match to "spice", "hospice", "spore" and many more but not "space", "spare" or any string with "spa". % is only used to denote the beginning and end of a pattern and not actually encoded as a character in our protocol.

ASCII has over 32 characters that are not printed and used solely for controlling interfaces. We can use some of these characters to encode the special characters used in the protocol. First of all, there are two characters used in the protocol that do not need to be encoded. The wildcard character %, as mentioned above, does not need to be encoded as it merely denotes the start and end of a pattern. The exclusion wildcard, !, also does not require encoding as the protocol uses auxiliary information to process exclusions. Secondly, two characters that do require encoding as special characters are the single (non-empty) wildcard character $ and a blank character # that denotes empty slots. These two characters can replace any of the control characters that are not used to represent written information. For example, we can encode $ to the value 2 and # to the value 3. Since we have more than enough control characters, there are no restrictions to using ASCII encoding in practice.

D. Security Model

In this paper, we consider a simple form of the standard definition of security in the static semi-honest model due to Goldreich [38]. Informally, all participants in the semi-honest model are required to follow the instructions prescribed in the protocol. Thus, security in our setting is further apparent since only the case where the server acts semi-honestly is considered. That is, no PPT server gains information about the client's private inputs, other than what can be deduced from the result of the protocol as well as all public textual parts of query statement. We argue the protocol's security in the semi-honest model by comparing an ideal-world model, where a trusted third party (TTP) receives the inputs of the client and outputs the result of the given query, to a real-world model of the protocol without a TTP.

We provide a formal definition for the security of PDQ protocols below.

Definition 1 (Computational Indistinguishability)**.** *Let $S \subseteq \{0,1\}^*$. We say that two ensembles indexed by S, $X := \{X_\sigma\}_{\sigma \in S}$ and $Y := \{Y_\sigma\}_{\sigma \in S}$, are computationally indistinguishable if for any polynomial-time algorithm D, there exists a negligible function $\mathsf{negl}(\cdot)$ such that*

$$\Big| \Pr[D(\sigma, X_\sigma) = 1] - \Pr[D(\sigma, Y_\sigma) = 1] \Big| \leq \mathsf{negl}(|\sigma|).$$

In this case, we denote this by $X \stackrel{c}{\equiv} Y$.

Suppose $\mathcal{Q}$ is a polynomial-time query functionality, and $\pi_\mathcal{Q}$ is the query protocol. Let x and y be the inputs from client and server, respectively. We assume that each input consists of two parts: a public textual part (denoted by x^*) and a private part (denoted by $\bar{x}$). We define the client's view of the protocol as $(x, r^\mathrm{c}, m_1^\mathrm{c}, \ldots, m_k^\mathrm{c})$ and the server's view as $(y, r^\mathrm{s}, m_1^\mathrm{s}, \ldots, m_{k'}^\mathrm{s})$, where r^c and r^s are client's and server's respective internal coin tosses and m_i^c's and m_i^s's are client's and server's respective i-th message during the execution of the protocol. For compatibility with existing notation, we will write the server's view as $\mathsf{view}(x, y)$. For clarity, we would like to remark that client's input includes unencrypted plaintexts (e.g., in SQL, the keywords like `select` and `from` and a list of column names in a target table). Of course, all constants in a search condition and a corresponding query result will be given in encrypted form. Sometimes, in practice, more communication than stated may be needed to complete the protocol. These components make up the client's view.

Definition 2. *Protocol $\pi_{\mathcal{Q}}$ is a secure query protocol for the query functionality $\mathcal{Q}$ in the presence of a semi-honest server if there exists a PPT simulator $\mathcal{S}$ such that*

$$\{\mathcal{S}(y, \mathcal{Q}(x,y))\}_{x,y\in\{0,1\}^*} \stackrel{c}{\equiv} \{\mathsf{view}(x,y)\}_{x,y\in\{0,1\}^*}.$$

We notice that in our setting it is not necessary for a server to provide some private input, and thus y will involve only some public values.

IV. Our Constructions

In this section, we present our PDQ protocols for conjunctive and disjunctive queries with wildcard search conditions on an encrypted database using FHE. We start with designing a pattern matching algorithm on an encrypted string and an encrypted pattern, allowing additional inputs to hide details of the pattern. Then, our PDQ protocols are built on top of this algorithm. For the readers who are more interested in the performance of our protocols, we recommend reading (in order) the details of implementation and optimization in Section V-A and benchmarks and analysis in Section V-B.

Recall that our proposed algorithms and protocols use a ring-LWE based FHE scheme with plaintext space $\mathbb{F}_{2^8}$ and strings are encrypted character-wise. For example, we may employ the BGV scheme while encoding a 8-bit character $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_8) \in \{0,1\}^8$ into $\sum_{i=0}^{7} \alpha_{i+1} t^i \in \mathbb{F}_{2^8}$ where $\mathbb{F}_{2^8}$ is isomorphic to $\mathbb{F}_2[x]/\langle f(x)\rangle$ and $t \in \mathbb{F}_{2^8}$ is a root of $f(x)$.

For encrypted strings $\overline{A} = (\overline{a}_1, \cdots, \overline{a}_\eta)$ and $\overline{B} = (\overline{b}_1, \cdots, \overline{b}_\eta)$ of the same length η, we define ciphertext additions and multiplications on $\overline{A}$ and $\overline{B}$ through component-wise additions and multiplications, respectively, i.e.,

$$\overline{A} + \overline{B} \;=\; (\overline{a}_1 + \overline{b}_1, \cdots, \overline{a}_\eta + \overline{b}_\eta)$$

and

$$\overline{A} \cdot \overline{B} \;=\; (\overline{a}_1 \cdot \overline{b}_1, \cdots, \overline{a}_\eta \cdot \overline{b}_\eta).$$

As stated in Section III-C, we assume that all attribute strings do not include the wildcard character \$ and wildcard patterns do not include #. These two characters, \$ and #, are encoded by replacing some of the unused control characters in the ASCII code.

A. Pattern Matching on Encrypted Data with Auxiliary Inputs

We first design an algorithm that takes two inputs, an encrypted string and an encrypted pattern, and returns an encryption of 1 if the pattern occurs in the string and an encryption of 0 otherwise. The encrypted pattern is composed of two parts, an encrypted string and some encrypted auxiliary bits.

Our Strategy. For a string $A = (a_1, a_2, \ldots, a_\eta) \in (\Sigma\backslash\{\$\})^\eta$ of η characters and a pattern $W = (w_1, w_2, \ldots, w_\tau) \in (\Sigma\backslash\{\#\})^\tau$ of τ characters, a pattern matching test for $\%W\%$ can be described as the following composition of disjunction and conjunction of (non-)equality clauses,

$$\bigvee_{i=0}^{\eta-\tau} \Big(\bigwedge_{j\in S} \underbrace{\big((a_{i+j} = w_j \neq \$) \vee (w_j = \$)\big)}_{\text{(a)}} \bigwedge_{j\notin S} \underbrace{\big((a_{i+j} \neq c_j)\big)}_{\text{(b)}} \Big). \quad (3)$$

Here, S is a subset of $\{1, \ldots, \tau\}$ which consists of positions between 1 and τ such that w_j is not an exclusion wildcard, $!(c_j)$. We note that a key requirement in designing our protocols is that τ should not be revealed while the above relation is evaluated at $\overline{A}$ and $\overline{W}$. In addition to that, we should not reveal the location of the wildcards in the pattern.

First, we show how to hide the length of the pattern by tweaking part (a) of Equation (3). We modify the set S into $S' = S \cup \{\tau+1, \ldots, \eta\}$, set $w_j = \$$ for $\tau+1 \le j \le \eta$ and extend part (a) of Equation (3) into

$$\bigwedge_{j\in S'} \big(((a_j^{(i)} = w_j \neq \$) \vee (w_j = \$)\big) \quad (4)$$

where $A^{(i)} := (a_1^{(i)}, \ldots, a_\eta^{(i)}) = (a_{i+1}, \ldots, a_\eta, \overbrace{\#, \ldots, \#}^{i})$ for $0 \le i \le \eta - 1$. From Equation (4), we observe that the two clauses in part (a), $((a_j^{(i)} = w_j) \wedge (w_j \neq \$))$ and $\big(w_j = \$\big)$, are disjoint as long as $a_j^{(i)} \neq \$$. As mentioned earlier, the protocol specifies that there are no \$ present in any string-typed attribute value A; thus, the two clauses are indeed disjoint. Thus, we simply need to perform an equality test to determine if $(a_j^{(i)} = w_j)$, and its result corresponds to the first clause $\big((a_j^{(i)} = w_j) \wedge (w_j \neq \$)\big)$. Then, the sum of the results of the two clauses in part (a) yields their disjunction. To avoid additional computation, we introduce a set of constants $\{e_j\}_{j=1}^{\eta}$, where $e_j = 1$ if $w_j = \$$ and $e_j = 0$ otherwise. Then, Equation (4) can be evaluated by computing

$$\prod_{j\in S'} (\mathtt{EQ}(\overline{a}_j^{(i)}, \overline{w}_j) + \overline{e}_j).$$

Next, we should handle another part that has exclusion wildcards in the pattern. For exclusion wildcards $!(c_j)$, we check if the attribute character $a_j^{(i)}$ is not equal to c_j as encrypted; this can be computed as $\overline{1} + \mathtt{EQ}(\overline{a}_j^{(i)}, \overline{c}_j)$ if an exclusion wildcard is in position j. To homogenize the operations that are done with and without wildcards and exclusions, we set $w_j = c_j$ and $e_j = 1$ if position j of the pattern is the exclusion wildcard $!(c_j)$. Then, we can compute part (a) and (b) in Equation (3) by homomorphically evaluating

$$\overline{z}_i = \prod_{j=1}^{\eta} \Big(\mathtt{EQ}(\overline{a}_j^{(i)}, \overline{w}_j) + \overline{e}_j\Big) \quad (5)$$

without knowledge of τ and W. It is easily verifiable that z_i is 1 if a substring $(a_{i+1}, \ldots, a_{i+\tau})$ of A contains pattern W and 0 otherwise.

Consequently, we can write Equation (3) as $\bigvee_{i=0}^{\eta-\tau} z_i$. To evaluate this without revealing τ, we introduce another set of constants d_i's determined by considering the index i. Specifically, we set $d_i = 1$ if $0 \le i \le \eta - \tau$, and $d_i = 0$ if $\eta - \tau + 1 \le i \le \eta - 1$ for a similar purpose to the e_j's. We then compute the equation $\bigvee_{i=0}^{\eta-\tau} z_i$ by homomorphically evaluating

$$\overline{t} = \overline{1} + \prod_{i=0}^{\eta-1} (\overline{1} + \overline{z}_i \cdot \overline{d}_i), \quad (6)$$

which is $\overline{1}$ if A contains the pattern W and $\overline{0}$ otherwise.

Algorithm Description. Now, we provide a description of our algorithm for performing pattern matching on encrypted data. Let $A = (a_1, \ldots, a_\eta) \in \Sigma^\eta$ and $W = (w_1, \ldots, w_\tau) \in \Sigma^\tau$ be an attribute value of length η and a pattern of length τ, respectively. We expand W to a string of η characters by padding $(\eta - \tau)$ \$'s to the end of W, i.e., $W = (w_1, \ldots, w_\eta) = (w_1, \ldots, w_\tau, \$, \ldots, \$)$, where $w_j = c_j$ if the wildcard $!(c_j)$ is at position j and w_j remains unchanged otherwise. On top of that, we determine two lists of constants e_j, d_i's as follows: For $1 \le j \le \eta$ and $0 \le i \le \eta - 1$,

$$e_j = \begin{cases} 1 & \text{if } w_j \text{ is corresponded to a wildcard, \$ or } !(c_j) \\ 0 & \text{otherwise} \end{cases}$$

and

$$d_i = \begin{cases} 1 & \text{if } 0 \le i \le \eta - \tau \\ 0 & \text{if } \eta - \tau + 1 \le i \le \eta - 1. \end{cases}$$

Then, set $E = (e_1, \ldots, e_\eta)$ and $D = (d_0, \ldots, d_{\eta-1})$.

As a consequence, we specify our wildcard pattern matching algorithm, denoted by $\mathsf{WM}(\overline{A}, \overline{W}, \overline{E}, \overline{D})$, in Algorithm 1.

Algorithm 1 Wildcard Pattern Matching $\mathsf{WM}(\overline{A}, \overline{W}, \overline{E}, \overline{D})$

Input: $\overline{A} = (\overline{a}_1, \ldots, \overline{a}_\eta)$, $\overline{W} = (\overline{w}_1, \ldots, \overline{w}_\eta)$, $\overline{E} = (\overline{e}_1, \ldots, \overline{e}_\eta)$ and $\overline{D} = (\overline{d}_0, \ldots, \overline{d}_{\eta-1})$

Output: $\overline{t}$

1: **for** $0 \le i \le \eta - 1$ **do**
2: $\quad \overline{A}^{(i)} := (\overline{a}_1^{(i)}, \ldots, \overline{a}_\eta^{(i)}) \leftarrow (\overline{a}_{i+1}, \ldots, \overline{a}_\eta, \overline{\#}, \ldots, \overline{\#})$
3: $\quad \overline{X}_i = \overline{A}^{(i)} - \overline{W}$
4: $\quad$ **for** $1 \le j \le \eta$ **do**
5: $\quad\quad \overline{y}_{i,j} = \mathtt{EQ}(\overline{x}_{i,j}, \overline{0})$ /* $\overline{X}_i = (\overline{x}_{i,1}, \overline{x}_{i,2}, \ldots, \overline{x}_{i,\eta})$ */
6: $\quad$ **end for**
7: $\quad \overline{Y}_i \leftarrow (\overline{y}_{i,1}, \overline{y}_{i,2}, \cdots, \overline{y}_{i,\eta})$
8: $\quad \overline{S}_i \leftarrow \overline{Y}_i + \overline{E}$
9: $\quad \overline{z}_i = \prod_{j=1}^{\eta} \overline{s}_{i,j}$ /* $\overline{S}_i = (\overline{s}_{i,1}, \overline{s}_{i,2}, \cdots, \overline{s}_{i,\eta})$ */
10: **end for**
11: $\overline{t} = \overline{1} + \prod_{i=0}^{\eta-1} (\overline{1} + \overline{z}_i \cdot \overline{d}_i)$
12: **return** $\overline{t}$

Analysis. Now, we argue that our pattern matching algorithm works correctly and reveals nothing information about client's private input. First, the following theorem ensures that our proposed pattern matching algorithm is correct.

Theorem 1. *(Correctness of* WM*) Let A and W be defined as above. Assuming that E and D are correctly set for a given pattern W, the pattern matching algorithm* WM *correctly computes an encryption of the matching result between an encrypted string $\overline{A}$ and an encrypted pattern $\overline{W}$.*

Proof. At Step 3, for $0 \le i \le \eta - \tau$,

$$x_{i,j} = a_j^{(i)} - w_j = \begin{cases} a_{i+j} - w_j & \text{if } 1 \le j \le \tau, \\ a_{i+j} - \$ & \text{if } \tau + 1 \le j \le \eta - i, \\ \# - \$ & \text{if } \eta - i + 1 \le j \le \eta. \end{cases}$$

Here we do not need to consider the cases that $i > \eta - \tau$ since they are removed from consideration via constants d_i's later. At Step 5, for $0 \le i \le \eta - \tau$,

$$y_{i,j} = \mathtt{EQ}_{\mathcal{P}}(x_{i,j}, 0) = \begin{cases} 1 & \text{if } 1 \le j \le \tau \text{ and } a_{i+j} = w_j, \\ 0 & \text{if } 1 \le j \le \tau \text{ and } a_{i+j} \ne w_j, \\ 0 & \text{if } \tau + 1 \le j \le \eta, \end{cases}$$

where $\mathtt{EQ}_{\mathcal{P}}$ is an algorithm that takes two elements in the plaintext space $\mathcal{P}$ of the underlying FHE scheme as inputs and returns 1 if two inputs are the same, and 0 otherwise. Hence, for $0 \le i \le \eta - \tau$ and $1 \le j \le \eta$,

$$\begin{aligned} s_{i,j} &= y_{i,j} + e_j \\ &= \begin{cases} 0 & \text{if } 1 \le j \le \tau,\ a_{i+j} \ne w_j \text{ and } w_j \ne \$, !(c_j), \\ 0 & \text{if } 1 \le j \le \tau,\ a_{i+j} = c_j \text{ and } w_j = !(c_j), \\ 1 & \text{otherwise,} \end{cases} \end{aligned}$$

since the plaintext space is a field of characteristic 2, and $e_j = 1$ if w_j is a wildcard character $\big(\$ \text{ or } !(c_j)\big)$ and 0 otherwise.

If A contains pattern W, then there exists an index $0 \le i \le \eta - \tau$ such that either $(a_{i+j} = w_j \ne \$)$, $(w_j = \$)$ or $(a_{i+j} \ne c_j$ and $w_j = !(c_j))$ for all $1 \le j \le \tau$. Thus, for this index i, $z_i = \prod_{j=1}^{\eta} s_{i,j} = 1$ and $1 + z_i \cdot d_i = 0$ since $d_i = 1$ for $0 \le i \le \eta - \tau$. The result is that $t = 1 + \prod_{i=0}^{\eta-1}(1 + z_i \cdot d_i) = 1 + 0 = 1$.

Otherwise (i.e., A does not contain pattern W), for all i, there exists at least one index j such that either $(a_{i+j} \ne w_j$ and $w_j \ne \$, !(c_j))$ or $(a_j = c_j$ and $w_j = !(c_j))$. Therefore, $s_{i,j} = 0$ and $z_i = \prod_{j=1}^{\eta} s_{i,j} = 0$ for all i and such j which means $t = 1 + \prod_{i=0}^{\eta-1}(1 + z_i \cdot d_i) = 1 + 1 = 0$. □

It is easy to show that server cannot learn information about pattern string and its length while running the algorithm. Of course, the maximum length of pattern string will be known to the server. Our algorithm WM takes as inputs $(\overline{A}, \overline{W}, \overline{E}, \overline{D})$ which have been encrypted by a semantically secure FHE scheme. All of the following processes are carried out over FHE ciphertexts but the server should know the number of times that the loop statements in the algorithm execute (i.e., η). Hence, the execution of WM reveals no information about the string A and the pattern W except for their maximum length η.

Next, we will check the efficiency of our algorithm and then describe how our algorithm can be used in designing an algorithm for exact matching.

Efficiency. Since an FHE scheme of plaintext space $\mathbb{F}_{2^8}$ is used, the $\mathtt{EQ}$ algorithm at Step 5 requires $3 (= \log_2 8)$ levels. For Steps 9 and 11, multiplying η terms uses $\lceil \log_2 \eta \rceil$ levels each. At Step 11, an additional level is needed when multiplying $\overline{z}_i$ and $\overline{d}_i$. Thus, the proposed algorithm requires $4 + 2\lceil \log_2 \eta \rceil$ levels in total.

Exact Matching. The exact matching test is a special case of wildcard matching tests. More precisely, for the exact matching test against a keyword $W = (w_1, \ldots, w_\eta)$, we set $e_i = 0$ for all $1 \le i \le \eta$, and set $d_0 = 1$ and $d_i = 0$ for all $1 \le i \le \eta - 1$. Then, we can obtain a matching result by running $\mathsf{WM}(\overline{A}, \overline{W}, \overline{E}, \overline{D})$.

1. The client prepares a list of encrypted strings
$$\begin{cases} \overline{W}_j, \overline{E}_j, \overline{D}_j & \text{for } j \in J_1, \\ \overline{W}_j^* & \text{for } j \in J_2, \end{cases}$$
where $\overline{E}_j$ and $\overline{D}_j$ are encryptions of mask bits for the pattern W_j, as in Section IV-A, and sends them to the server.
2. Using $\left(\overline{W}_j, \overline{E}_j, \overline{D}_j\right)_{j \in J_1}$ and $\left(\overline{W}_j^*\right)_{j \in J_2}$, the server
   (a) computes
$$\overline{\beta}_{ij} = \begin{cases} \mathsf{WM}(\overline{A}_{ij}, \overline{W}_j, \overline{E}_j, \overline{D}_j) & \text{if } j \in J_1, \\ \mathsf{EM}(\overline{A}_{ij}, \overline{W}_j^*) & \text{if } j \in J_2, \end{cases}$$
   (b) computes $\overline{\gamma}_i = \overline{A}_{ik} \cdot \prod_{j \in J_1} \prod_{j \in J_2} \overline{\beta}_{ij}$ for $1 \le i \le n$, and
   (c) sends $\{\overline{\gamma}_1, \cdots, \overline{\gamma}_n\}$ to the server.
3. The client obtains $\{\gamma_1, \gamma_2, \cdots, \gamma_n\}$ by decrypting $\{\overline{\gamma}_1, \overline{\gamma}_2, \cdots, \overline{\gamma}_n\}$ under the client's secret key.

Fig. 1. Our PDQ protocol for conjunctive case with exact matching and wildcard matching conditions

Furthermore, if restricted to the exact matching test, WM can be simplified significantly. It takes an encrypted string $\overline{A} = (\overline{a}_1, \ldots, \overline{a}_\eta)$ and an encrypted keyword $\overline{W} = (\overline{w}_1, \ldots, \overline{w}_\eta)$ as inputs and produces $\overline{t} = \prod_{i=1}^{\eta} \overline{y}_i$ where $\overline{y}_i = \mathtt{EQ}(\overline{a}_i, \overline{w}_i)$ for $1 \le i \le \eta$. We denote this simplified algorithm by $\mathsf{EM}(\overline{A}, \overline{W})$.

Patterns $W\%$ and $\%W$. Besides the pattern "$\%W\%$", there are two other patterns "$W\%$" and "$\%W$" that contain the wildcard character %. We can also support them by slightly modifying the form of input pattern and constants. For the first case, we can regard it as a matching between attributes and an η-character pattern $W' = (w_1, \ldots, w_\tau, \$, \ldots, \$)$ with allowing that W' has wildcard characters, where $W = (w_1, \ldots, w_\tau)$. Hence, we can support "$W\%$" by simply setting $d_0 = 1$ and $d_i = 0$ for $1 \le i \le \eta - 1$ in our WM algorithm with W'.

In the second case, we require that the pattern occurs strictly at the end of the attribute string. For this case, we can support all patterns W, except for the pattern that ends with an exclusion $!(c_\tau)$, i.e., $w_\tau = !(c_\tau)$ for $W = (w_1, \ldots, w_\tau)$. To this end, we first extend the pattern $W = (w_1, \ldots, w_\tau)$ to an η-character pattern $W' = (w_1, \ldots, w_\tau, \#, \ldots, \#)$ by allowing an extended pattern to have $\#$, differently from other cases "$\%W\%$" and "$W\%$". Furthermore, we set w_τ to $!(\#)$ in the case that $w_\tau = \$$. We note that it does not change the result if A ends with W, because a_i's for $1 \le i \le \eta'$ are not $\#$ when A's actual length is $\eta' \le \eta$. However, we need this modification for correctness when A does not end with W. The rest is the same as the original WM algorithm.

Now, let us check the correctness of the above modification. Let A be an attribute string of actual length $\eta' \le \eta$, i.e., $A = (a_1, \ldots, a_\eta) = (a_1, \ldots, a_{\eta'}, \#, \ldots, \#)$. Define $A^{(i)} := (a_1^{(i)}, \ldots, a_\eta^{(i)}) = (a_{i+1}, \ldots, a_\eta, \underbrace{\#, \ldots, \#}_{i})$. If A ends with W, then for $A^{(\eta'-\tau)} = (a_{\eta'-\tau+1}, a_{\eta'-\tau+2}, \ldots, a_{\eta'}, \#, \ldots, \#)$, we have either $(a_j^{(\eta'-\tau)} = w_j \ne \$)$, $(w_j = \$)$ or $(a_j^{(\eta'-\tau)} \ne c_j$ and $w_j = !(c_j))$ for all $1 \le j \le \tau$. Thus, $z_{\eta'-\tau} = 1$ in Equation (5) and $t = 1$ in Equation (6).

On the other hand, if A does not end with W, we can consider the following cases: For $A^{(i)} := (a_1^{(i)}, \ldots, a_\eta^{(i)}) = (a_{i+1}, \ldots, a_{\eta'}, \#, \ldots, \#)$,

- if $0 \le i < \eta' - \tau$, $a_{\tau+1}^{(i)} = a_{\tau+1+i} \ne \#$, but $w_{\tau+1} = \#$.
- If $i = \eta' - \tau$, there exists at least one index j such that either $(a_j^{(i)} \ne w_j$ and $w_j \ne \$, !(c_j))$ or $(a_j^{(i)} = c_j$ and $w_j = !(c_j))$ since we assume that A does not end with W.
- If $i > \eta' - \tau$, $a_\tau^{(i)} = \#$, but $w_\tau \ne \#$.

Thus, for all $0 \le i \le \eta - 1$, there exist at least one index j such that $(a_j^{(i)} \ne w_j$ and $w_j \ne \$, !(c_j))$ or $\left(a_j^{(i)} = c_j \text{ and } w_j = !(c_j)\right)$. So, $z_i = 0$ in Equation (5) for all $0 \le i \le \eta - 1$ and $t = 0$ in Equation (6).

Note that we replace $w_\tau = \$$ by $w_\tau = !(\#)$ for the correctness when $i > \eta' - \tau$. If $w_\tau = !(c_\tau)$, our modification should support an exclusion of $\#$ and c_τ, but it can support an exclusion of single character only. This is the reason why our algorithm cannot support pattern "$\%W$" if W ends with an exclusion $!(c_\tau)$.

B. Compound Queries with Wildcard Search Condition

We describe PDQ protocols for conjunctive/disjunctive queries with exact and wildcard matching conditions. Recall that $\langle \mathsf{R}(\mathsf{A}_1, \ldots, \mathsf{A}_\rho), \{\overline{A}_1, \ldots, \overline{A}_n\}\rangle$ is an encrypted table of n elements where each A_i has ρ attribute values, i.e., $A_i = (A_{i1}, \ldots, A_{i\rho})$. Throughout this subsection, we assume for simplicity that all attributes are string-typed variables and that each string A_{ij} $(1 \le i \le n, 1 \le j \le \rho)$ is encrypted as before.

Conjunctive Case. We first consider conjunctive queries with exact matching and wildcard matching conditions. More specifically, we wish to handle the following conjunctive query that returns all A_{ik}'s satisfying

$$\Big(\bigwedge_{j \in J_1} (A_{ij} = \%W_j\%) \Big) \bigwedge \Big(\bigwedge_{j \in J_2} (A_{ij} = W_j^*) \Big) \qquad (Q1)$$

for a fixed $1 \le k \le \rho$, where J_1 is the set of indices j's such that the j-th column corresponds to wildcard matching $(A_{ij} = \%W_j\%)$ in query $(Q1)$ and J_2 is the set of indices j's such that the j-th column corresponds to exact matching $(A_{ij} = W_j^*)$ in

query ($Q1$). Note that we can also support patterns "$W\%$" and "$\%W$" by adapting the algorithms explained in Section IV-A.

To better illustrate our PDQ protocol, we present it for this conjunctive query in Figure 1.

Analysis. The correctness of our conjunctive query protocol immediately follows from the correctness of the WM and EM algorithms. The security of the protocol is also straightforward, since our PDQ protocol uses a semantically secure FHE scheme as its underlying encryption scheme. The following theorem formally states the security of the construction.

Theorem 2. *Assume that the FHE scheme used in the protocol is semantically secure. Then, in the PDQ protocol described above, a PPT semi-honest server learns no more information than could be revealed in the ideal model by using the same private inputs with a trusted third party.*

Proof. Recall that the employed FHE scheme is semantically secure; that is, no PPT server can distinguish between $\overline{m}_0$ and $\overline{m}_1$ for arbitrary plaintexts m_0, m_1 of the same size. Therefore, except the public textual parts of a given query statement, nothing is revealed to the server before decryption by the client as their inputs are encrypted by a semantically secure FHE scheme. Note that the maximum length η of string-typed attributes is known to the server during the transmission of a description of the relation schema R. Thus, the server learns only the public textual part of the query statement. Of course, besides the query statement given to the server, the client also needs to send auxiliary inputs which are encrypted as well, so no information about W is revealed to the server. For example, when a query statement (Q) is given by

$$\texttt{select } \mathsf{A}_\ell \texttt{ from R where } \mathsf{A}_j \texttt{ like } \text{“}\%W\%\text{”}; \qquad (Q)$$

the server learns only the `select` and `from` clauses and A_j in the `where` clause, but no information about the pattern W and its length τ is given.

Furthermore, all attribute values A_{ij}'s in the i-th tuple for a matched attribute A_j against the encrypted pattern $\overline{W}$ were encrypted by the semantically secure FHE scheme and stored at the server. Hence, the server cannot learn any information about matched strings $\overline{A}_{ij}$'s. As studied in the analysis of our matching algorithm in Section IV-A, the algorithm WM reveals no information about its inputs to the server. Moreover, applying the same result to our exact-matching algorithm EM implies that the algorithm also reveals no information on its inputs to the server. Therefore, while computing $\overline{\beta}_{ij}$ from the client's inputs and encrypted database, the server learns nothing about the pattern W. Likewise, the server computes $\{\overline{\gamma}_i\}_{i=1}^n$ without knowledge of the pattern W. Overall, no information about the pattern W and any matched attribute value A_{ij} of the honest client are revealed to the semi-honest server other than that given by the encrypted result set $\{\overline{\gamma}_i\}_{i=1}^n$ of the protocol. □

Efficiency. Step 2 (a) in our protocol requires a multiplicative depth $4 + 2\lceil \log_2 \eta \rceil$ for running WM and EM algorithms. Step 2 (b) requires $(1 + |J_1| + |J_2|)$ multiplications, and so it requires a multiplicative depth $\lceil \log_2 (1 + |J_1| + |J_2|) \rceil$. Therefore, the total depth for our conjunctive query protocol is $4 + 2\lceil \log_2 \eta \rceil + \lceil \log_2 (1 + |J_1| + |J_2|) \rceil$.

The communication cost on the client side is $(3|J_1|+|J_2|)\eta$ FHE ciphertexts and that on the server side is n FHE ciphertexts of strings of η characters, i.e., $n\eta$ FHE ciphertexts.

Remark 1. *We may reduce the communication cost on the client side if the server himself obtains the encrypted constants $\overline{E}_j = (\overline{e}_{jk})_{k=1}^{\eta}$ from the encrypted pattern $\overline{W}_j = (\overline{w}_{jk})_{k=1}^{\eta}$ by performing $\overline{e}_{jk} \leftarrow \mathtt{EQ}(\overline{w}_{jk}, \overline{\$})$ for $j \in J_1$ and $1 \le k \le \tau$, since $e_{jk} = 1$ if $w_{jk} = \$$ and 0 otherwise. With this method, the client does not need to transmit η ciphertexts per each wildcard matching condition and hence the communication cost on the client side is reduced to $(2|J_1|+|J_2|)\eta$ FHE ciphertexts. See Section V-A for the running time overhead required for this modification.*

Disjunctive Case. We can easily obtain a PDQ protocol for disjunctive case by modifying computations of $\overline{\beta}_{ij}$ and $\overline{\gamma}_i$. Let us consider the following disjunctive query that returns all A_{ik}'s satisfying

$$\Big(\bigvee_{j \in J_1} (A_{ij} = \%W_j\%) \Big) \bigvee \Big(\bigvee_{j \in J_2} (A_{ij} = W_j^*) \Big) \qquad (Q2)$$

for a fixed $1 \le k \le \rho$, where, as denoted earlier, J_1 is the set of indices j's such that the j-th column corresponds to wildcard matching $(A_{ij} = \%W_j\%)$ in the query $(Q2)$ and J_2 is the set of indices j's such that the j-th column corresponds to exact matching $(A_{ij} = W_j^*)$ in the query $(Q2)$. We obtain a PDQ protocol for disjunctive case by replacing computations of $\overline{\beta}_{ij}$ and $\overline{\gamma}_i$ in our PDQ protocol for conjunctive case with

$$\overline{\beta}_{ij} = \begin{cases} \overline{1} - \mathsf{WM}(\overline{A}_{ij}, \overline{W}_j, \overline{E}_j, \overline{D}_j) & \text{if } j \in J_1, \\ \overline{1} - \mathsf{EM}(\overline{A}_{ij}, \overline{W}_j^*) & \text{if } j \in J_2, \end{cases}$$

and

$$\overline{\gamma}_i = \overline{A}_{ik} \cdot (\overline{1} + \prod_{j \in J_1} \prod_{j \in J_2} \overline{\beta}_{ij}),$$

respectively, for all $1 \le i \le n$. We note that computational and communicational costs for the disjunctive case are exactly the same as those for the conjunctive case.

V. Implementation

In this section, we report some performance evaluation results with a prototype implementation of our solutions. We first introduce several techniques used to improve performance in Section V-A and discuss some experimental results in Section V-B.

A. Implementation Details

SIMD Techniques. To minimize the number of ciphertext operations, we use SIMD techniques, first proposed by Smart and Vercauteren in [36]. SIMD enables packing multiple characters into one ciphertext and using one operation to compute on all packed characters simultaneously. With enough many slots, multiple strings can be packed into one ciphertext; as many strings as possible are packed in and then padded with

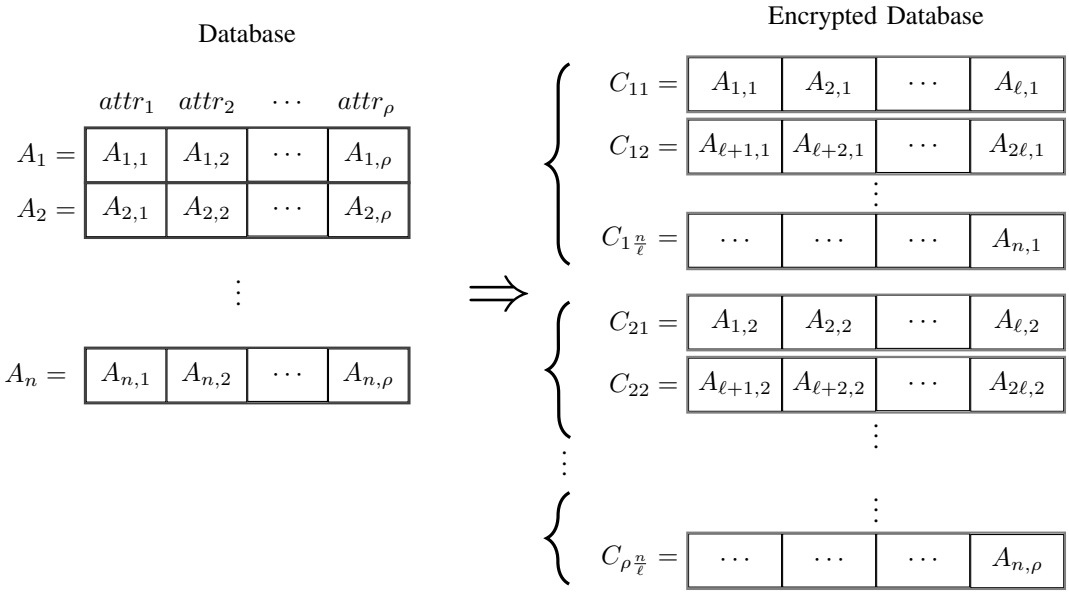

Fig. 2. Diagram of the Packing Method

zeroes. Permuting the locations of encrypted characters is done by performing a key-switching operation on the ciphertext. This does not require multiplicative depth but may take more time than operations such as ciphertext addition or constant multiplication.

The key to the SIMD technique for existing ring-LWE based FHE schemes is to notice that the m-th cyclotomic polynomial modulus used, $\Phi_m(x)$, factors into $r = \eta\ell$ irreducible factors, $\{f_i(x)\}_{i=1}^r$, each of degree δ for some δ that divides $\phi(m)$, modulo 2 $\big($i.e. $\Phi_m(x) \equiv \prod_{i=1}^r f_i(x) \bmod 2\big)$. We can then encode a vector of r polynomials, with binary coefficients and degree less than δ each, into a larger polynomial modulo $\Phi_m(x)$ and 2, with the Chinese Remainder Theorem for polynomial rings. Note that the algebra of each slot in the vector corresponds to the field $\mathbb{F}_{2^\delta}$.

In Figure 2, we depict the method we use to fill ciphertexts with entries from the database. Each row of the table, A_i, corresponds to one record in the database, with its attributes being $A_{i,1}$ to $A_{i,\rho}$. When filling ciphertexts, we do not put each record into one ciphertext, but pack ciphertexts by attributes to maximise the advantage of SIMD operations, since the cells for each attribute would undergo the same test. Each box in the figure represents one cell in the database and takes up η characters in the ciphertext as we encrypt data character-wise.

Multi-Threading. As the protocol contains several loops that each operates independently, we use OpenMP [39] to distribute the processing load across multiple threads. We thus utilize most of the available resources to increase implementation performance and greatly reduce the time taken to run the protocol. Besides loops, we also use multiple threads to speed up the computation of $\bar{t}$. Computing $\prod_{i=0}^{\eta-1}(1-\overline{z}_i \cdot \overline{d}_i)$ should be done by arranging each term $(1-\overline{z}_i \cdot \overline{d}_i)$ in the leaves of a binary tree and multiplying them pairwise till one is left. This method consumes the least depth, $\lceil \log_2 \eta \rceil$, and uses $(\eta - 1)$ multiplications which is the same as multiplying the elements sequentially. At each level of the multiplication binary tree, the ciphertexts that have different parents are independent of each other which allows us to use one thread per parent at each level.

From Remark 1, to reduce the communication cost on the server side, the protocols can be changed so that encrypted constants $\{\overline{e}_j\}_{j=1}^{\eta}$ are computed at the server with equality tests between $\{\overline{w}_j\}_{j=1}^{\eta}$ and $\overline{\$}$. We report that the increase in amortized time for this modification is about 0.3-0.5 seconds from our experiments with multi-threading techniques.

Conjunction Optimization. The wildcard matching algorithm WM requires $\lceil \log_2 \eta \rceil$ more multiplicative depth than the exact matching algorithm EM in the system because intermediate values $\overline{z}_i$'s do not need to be computed. Thus, rather than computing $\overline{\gamma}_i$ after WM algorithms are complete, $\prod_{j \in J_2} \overline{\beta}_{ij}$ is done first then multiplied by the results of $\mathsf{WM}(\overline{A}_{ij}, \overline{W}_j, \overline{E}_j, \overline{D}_j)$ to obtain $\overline{\gamma}_i$. This allows us to efficiently use the excess levels and perform this combining operation while waiting for the WM algorithm to complete.

B. Experiment Results

The test platform was a server equipped with Intel® Xeon® E5-2620 @ 2.40 GHz with 24 threads and 64 GB RAM. The implementation is built on top of the GMP [40], NTL [41], HElib [42] and OpenMP [39] libraries to employ a BGV FHE scheme [21] as an underlying FHE scheme and use multi-threading techniques. To evaluate security levels of the chosen BGV FHE scheme parameters, we use the estimator for (ring-)LWE instances by Albrecht et al. [43] factoring the latest dual lattice attack on FHE instances proposed by Albrecht [44].

For experiments, we used the BGV schemes whose slot size for packing and SIMD techniques is $\mathbb{F}_{2^{15}}$, due to the sparsity of available parameters that provide a slot size of $\mathbb{F}_{2^8}$ and adequate security together. Some kinds of SIMD operations, e.g., shift and rotation, do not consume a level in theory, but

TABLE I
PERFORMANCE RESULTS OF WILDCARD MATCHING ALGORITHM WM (UNIT FOR TIME: SEC)

String Length (Bytes)	# of Strings in a Ciphertext	Evaluate $\overline{X}_i$, $\overline{Y}_i$ & $\overline{S}_i$	Evaluate $\overline{z}_i$	Evaluate $\overline{t}$	Total Time	Amortised Time
35	51	40.34	55.46	5.10	100.91	1.98
45	40	51.92	63.05	5.98	120.96	3.02
55	32	66.74	79.05	7.43	153.21	4.78

A leveled BGV scheme of plaintext space $\mathbb{Z}_2[x]/\langle\Phi_m(x)\rangle$ for the m-th cyclotomic polynomial $\Phi_m(x)$ with $m = 32767$, slot size $\mathbb{F}_{2^{15}}$ and 29 levels with 83-bit security is used for this experiment.

TABLE II
PERFORMANCE RESULTS ON THE SERVER SIDE OF OUR PDQ PROTOCOL FOR CONJUNCTION QUERIES WITH 1 WM AND 5 EM CONDITIONS (UNIT FOR TIME: SEC)

String Length (Bytes)	# of Strings in a Ciphertext	Evaluate $\overline{X}_i$, $\overline{Y}_i$ & $\overline{S}_i$	Evaluate $\overline{z}_i$	Evaluate $\overline{\gamma}$	Total Time	Amortised Time
35	51	43.56	63.36	7.49	114.42	2.24
45	40	58.15	83.63	8.58	150.36	3.76
55	32	67.55	92.76	10.03	170.34	5.32

A leveled BGV scheme of plaintext space $\mathbb{Z}_2[x]/\langle\Phi_m(x)\rangle$ for the m-th cyclotomic polynomial $\Phi_m(x)$ with $m = 32767$, slot size $\mathbb{F}_{2^{15}}$ and 31 levels with 81-bit security is used for this experiment.

noise increase after such operations should be carefully considered if more operations are needed in practice. These factors increased the levels required to evaluate our algorithms and protocols correctly; so, we used the BGV scheme with 29 and 31 levels instead of theoretical bounds of $20(= 2\lceil\log_2\eta\rceil + 4)$ and $23(= 2\lceil\log_2\eta\rceil + 4 + \lceil\log_2(1 + |J_1| + |J_2|)\rceil)$ for the experiments of our algorithms and protocols, respectively.

However, despite taking a longer time to run the algorithms and protocols, larger parameters provide higher number of slots which means that more strings can be packed together and lowers the amortised time. To further minimise the amortised time, the parameters of the underlying BGV scheme were selected such that we can pack as many attribute strings as possible into one ciphertext while ensuring sufficient security. On the other hand, the length of the pattern is not considered as it is hidden in the system and the performance is unaffected by it. The $A^{(i)}$'s were pre-computed and do not count towards to the time taken by the algorithm and protocol.

Table I reports the implementation results of our wildcard algorithm for various attribute string sizes. Timing results for some computations are lumped together as some parts consume very little time compared to the rest. It is not explicitly reported, but we note that evaluating $\overline{Y}_i$ takes more than 95% of the time listed in the cells for the third column. The data from Table I demonstrates that computing $\overline{Y}_i$ and $\overline{z}_i$ take the longest time and is mostly because η of them have to be computed and these computations are the most intensive. The general trend of the results is as expected, larger η requires more time to process in both total and amortised time. This is because lower η means fewer ciphertext operations and more attribute strings are packed in a ciphertext which increases the performance significantly for amortised time.

Table II shows our implementation results of the proposed PDQ protocol for conjunction queries with one wildcard search condition and 5 exact search conditions. Similar to Table I, evaluating $\overline{Y}_i$ takes the vast majority of the listed time in the third column and computing $\overline{Y}_i$ and $\overline{z}_i$ take the longest time. The fifth column shows the time taken to convert the outputs of WM and EM algorithms into a final result.

Exact matching operations are interwoven with the wildcard test and so the timing results cannot distinguish between the two tests. $\overline{y}_i$'s and $\overline{z_i}$'s of EM are computed alongside $\overline{y}_{i,j}$'s and $\overline{z}_i$ of WM respectively in multiple threads. Although more operations are done with exact matching tests, multithreading reduces its impact and the added overhead to complete WM and EM evaluations is only about 15-25 seconds. This is largely due to evaluating $\overline{z}_i$ and that the results of the WM and EM algorithms have to be combined. Besides that change, the protocol returns the attribute A_{ik} that satisfies the conditions which involves additional processing after the algorithms which takes 5-8 seconds, based on η, with multithreading. Thus, the system can support other types of queries in additional to wildcard without much overhead.

General trends of this experiment are the same as above since the overhead is similar regardless of string length. The only difference is that the gap between amortized time for different string lengths has increased, due to the overhead of additional exact equality tests. Only in the case when the string length is 45 do we see a larger difference, but this is due to the difference in the number of cycles required to run the protocol compared to the WM algorithm. With only 24 threads available, the computation of WM, which spawns about 45 threads for parallel computation, could be done in two cycles of 24 at a time. For the protocol, the 5 EM conditions were ran with one thread per condition and thus needed 50 threads for parallel computation, taking 3 cycles of $24, 24$ and 2. The results show that it is possible to support a protocol with conjunctions between exact equality and wildcard queries since the overhead incurred is between 10-25% of the performance of the wildcard algorithm depending on the number of threads available.

C. Comparison and Discussions

As discussed in Section II, we have two closely related work relying on homomorphic encryption: Yasuda et al.'s protocol [19] and Saha and Koshiba's protocol [20]. However, since these solutions match encrypted patterns to cleartexts and require heavy processing to support compound queries, a fair comparison is quite difficult. Furthermore, none of them explicitly provide a way to match two encrypted strings.

In a different vein, there have been lots of nice results based on symmetric encryption with search capabilities. While it is quite clear that techniques in this line outperform FHE-based ones, as a trade-off, FHE-based techniques guarantee stronger security. To the best of our knowledge, a benchmark that meets the requirement of our research could not be found. Thus, we publicly release our code at our project site[6] for reproducibility.

VI. Conclusion

In this paper, we designed an algorithm for wildcard pattern matching between a string and a keyword encrypted using leveled fully homomorphic encryption, but using additional encrypted inputs. We guess this overhead seems to be unavoidable for hiding the pattern length. Then, by relying on the proposed matching algorithm on encrypted data, we provided private database query protocols for compound queries with wildcard search conditions on encrypted databases. Finally, we demonstrated proof-of-concept implementation results by applying several techniques for performance improvements, such as single-instruction-multiple-data operations and multi-threading techniques.

Our performance evaluation results demonstrate the efficiency of our protocol, but clearly are far from the practicality required for real-world applications. Thus the direction for future work is to develop a highly efficient system which will be configurable and match many realistic applications while preserving strong privacy.

Acknowledgements

We thank anonymous reviewers for their careful reading of our manuscript and their many insightful comments and suggestions. Myungsun Kim was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2017-R1D1A1B04035209). This work was done while Hyung Tae Lee was with Nanyang Technological University. Hyung Tae Lee, San Ling, Benjamin Hong Meng Tan, and Huaxiong Wang were supported by Research Grant TL-9014101684-01 and the Singapore Ministry of Education under Research Grant MOE2013-T2-1-041. Huaxiong Wang was also supported by NTU under Tier 1 grant RG143/14. Hyung Tae Lee is the corresponding author.

[6]https://www.github.com/benjaminhmtan/wildcardHElib

References

[1] Transaction Processing Performance Council, "TPC Benchmark H, Revision 2.17.1." available at http://www.tpc.org/.

[2] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *IEEE Symposium on Security and Privacy 2000*. IEEE Computer Society, 2000, pp. 44–55.

[3] E. Goh, "Secure indexes," *IACR Cryptology ePrint Archive*, vol. 2003, p. 216, 2003. [Online]. Available: http://eprint.iacr.org/2003/216

[4] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: Improved definitions and efficient constructions," in *ACM Conference on Computer and Communications Security (ACM CCS) 2006*, A. Juels, R. N. Wright, and S. D. C. di Vimercati, Eds. ACM, 2006, pp. 79–88.

[5] D. Boneh and B. Waters, "Conjunctive, subset, and range queries on encrypted data," in *Theory of Cryptography (TCC) 2007*, ser. LNCS, S. P. Vadhan, Ed., vol. 4392. Springer, 2007, pp. 535–554.

[6] S. Sedghi, P. van Liesdonk, S. Nikova, P. H. Hartel, and W. Jonker, "Searching keywords with wildcards on encrypted data," in *Security and Cryptography for Networks (SCN) 2010*, ser. LNCS, J. A. Garay and R. D. Prisco, Eds., vol. 6280. Springer, 2010, pp. 138–153.

[7] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *ACM Conference on Computer and Communications Security (ACM CCS) 2012*, T. Yu, G. Danezis, and V. D. Gligor, Eds. ACM, 2012, pp. 965–976.

[8] S. Kamara and C. Papamanthou, "Parallel and dynamic searchable symmetric encryption," in *Financial Cryptography and Data Security (FC) 2013*, ser. LNCS, A. Sadeghi, Ed., vol. 7859. Springer, 2013, pp. 258–274.

[9] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Highly-scalable searchable symmetric encryption with support for boolean queries," in *Advances in Cryptology - CRYPTO 2013 Part I*, ser. LNCS, R. Canetti and J. A. Garay, Eds., vol. 8042. Springer, 2013, pp. 353–373.

[10] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Dynamic searchable encryption in very-large databases: Data structures and implementation," in *Network and Distributed System Security Symposium (NDSS) 2014*. The Internet Society, 2014.

[11] M. Chase and E. Shen, "Substring-searchable symmetric encryption," *PoPETs*, vol. 2015, no. 2, pp. 263–281, 2015.

[12] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner, "Rich queries on encrypted data: Beyond exact matches," in *European Symposium on Research in Computer Security (ESORICS) 2015 Part II*, ser. LNCS, G. Pernul, P. Y. A. Ryan, and E. R. Weippl, Eds., vol. 9327. Springer, 2015, pp. 123–145.

[13] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Protecting confidentiality with encrypted query processing," in *ACM Symposium on Operating Systems Principles (SOSP) 2011*, T. Wobber and P. Druschel, Eds. ACM, 2011, pp. 85–100.

[14] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich, "Processing analytical queries over encrypted data," *PVLDB*, vol. 6, no. 5, pp. 289–300, 2013.

[15] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *ACM Symposium on Theory of Computing (STOC) 2009*, M. Mitzenmacher, Ed. ACM, 2009, pp. 169–178.

[16] D. Boneh, C. Gentry, S. Halevi, F. Wang, and D. Wu, "Private database queries using somewhat homomorphic encryption," in *Applied Cryptography and Network Security (ACNS) 2013*, ser. LNCS, M. Jacobson, M. Locasto, P. Mohassel, and R. Safavi-Naini, Eds., vol. 7954. Springer, 2013, pp. 102–118.

[17] C. Gentry, S. Halevi, C. Jutla, and M. Raykova, "Private database access with HE-over-ORAM architecture," in *Applied Cryptography and Network Security (ACNS) 2015*, ser. LNCS, T. Malkin, V. Kolesnikov, A. B. Lewko, and M. Polychronakis, Eds., vol. 9092. Springer, 2015, pp. 172–191.

[18] M. Kim, H. T. Lee, S. Ling, and H. Wang, "On the efficiency of FHE-based private queries," 2016, to appear in *IEEE Trans. Dependable and Secure Computing*.

[19] M. Yasuda, T. Shimoyama, J. Kogure, K. Yokoyama, and T. Koshiba, "Privacy-preserving wildcards pattern matching using symmetric somewhat homomorphic encryption," in *ACISP 2014*, ser. LNCS, W. Susilo and Y. Mu, Eds., vol. 8544. Springer, 2014, pp. 338–353.

[20] T. K. Saha and T. Koshiba, "An enhancement of privacy-preserving wildcards pattern matching," in *Foundations and Practice of Security (FPS) 2016*, ser. LNCS, F. Cuppens, L. Wang, N. Cuppens-Boulahia, N. Tawbi, and J. García-Alfaro, Eds., vol. 10128. Springer, 2016, pp. 145–160.

[21] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," in *Innovations in Theoretical Computer Science (ITCS) 2012*, S. Goldwasser, Ed. ACM, 2012, pp. 309–325.

[22] J. H. Cheon, M. Kim, and M. Kim, "Optimized search-and-compute circuits and their application to query evaluation on encrypted data," *IEEE Trans. Information Forensics and Security*, vol. 11, no. 1, pp. 188–199, 2016.

[23] G. S. Cetin, H. Chen, K. Laine, K. Lauter, P. Rindal, and Y. Xia, "Private queries on encrypted genomic data," Cryptology ePrint Archive, Report 2017/207, 2017, http://eprint.iacr.org/2017/207.

[24] M. Yasuda, T. Shimoyama, J. Kogure, K. Yokoyama, and T. Koshiba, "Secure pattern matching using somewhat homomorphic encryption," in *ACM Cloud Computing Security Workshop (CCSW) 2013*, A. Juels and B. Parno, Eds. ACM, 2013, pp. 65–76.

[25] Y. Zhang, J. Katz, and C. Papamanthou, "All your queries are belong to us: The power of file-injection attacks on searchable encryption," in *USENIX Security 2016*, T. Holz and S. Savage, Eds. USENIX Association, 2016, pp. 707–720.

[26] E. M. McCreight, "A space-economical suffix tree construction algorithm," *J. ACM*, vol. 23, no. 2, pp. 262–272, 1976.

[27] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica*, vol. 14, no. 3, pp. 249–260, 1995.

[28] J. Katz, A. Sahai, and B. Waters, "Predicate encryption supporting disjunctions, polynomial equations, and inner products," in *Advances in Cryptology - EUROCRYPT 2008*, ser. LNCS, N. P. Smart, Ed., vol. 4965. Springer, 2008, pp. 146–162.

[29] C. Hazay and Y. Lindell, "Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries," in *Theory of Cryptography (TCC) 2008*, ser. LNCS, R. Canetti, Ed., vol. 4948. Springer, 2008, pp. 155–175.

[30] K. B. Frikken, "Practical private DNA string searching and matching through efficient oblivious automata evaluation," in *DBSec 2009*, ser. LNCS, E. Gudes and J. Vaidya, Eds., vol. 5645. Springer, 2009, pp. 81–94.

[31] J. Katz and L. Malka, "Secure text processing with applications to private DNA matching," in *ACM Conference on Computer and Communications Security (ACM CCS) 2010*, E. Al-Shaer, A. D. Keromytis, and V. Shmatikov, Eds. ACM, 2010, pp. 485–492.

[32] J. Baron, K. E. Defrawy, K. Minkovich, R. Ostrovsky, and E. Tressler, "5pm: Secure pattern matching," in *Security and Cryptography for Networks (SCN) 2012*, ser. LNCS, I. Visconti and R. D. Prisco, Eds., vol. 7485. Springer, 2012, pp. 222–240.

[33] J. W. Bos, K. E. Lauter, J. Loftus, and M. Naehrig, "Improved security for a ring-based fully homomorphic encryption scheme," in *IMA International Conference on Cryptography and Coding (IMACC) 2013*, ser. LNCS, M. Stam, Ed., vol. 8308. Springer, 2013, pp. 45–64.

[34] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Advances in Cryptology - CRYPTO 2013 Part I*, ser. LNCS, R. Canetti and J. A. Garay, Eds., vol. 8042. Springer, 2013, pp. 75–92.

[35] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the AES circuit," in *Advances in Cryptology - CRYPTO 2012*, ser. LNCS, R. Safavi-Naini and R. Canetti, Eds., vol. 7417. Springer, 2012, pp. 850–867.

[36] N. P. Smart and F. Vercauteren, "Fully homomorphic SIMD operations," *Des. Codes Cryptography*, vol. 71, no. 1, pp. 57–81, 2014.

[37] G. S. Çetin, Y. Doröz, B. Sunar, and E. Savas, "Depth optimized efficient homomorphic sorting," in *Progress in Cryptology - LATINCRYPT 2015*, ser. LNCS, K. E. Lauter and F. Rodríguez-Henríquez, Eds., vol. 9230. Springer, 2015, pp. 61–80.

[38] O. Goldreich, *Foundations of Cryptography: Volume II Basic Applications*. Cambridge University Press, 2004.

[39] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[40] "GMP: The GNU multiple precision arithmetic library ver. 6.1.0," available at http://gmplib.org.

[41] V. Shoup, "NTL: A library for doing number theory version 9.7." available at http://www.shoup.net/ntl/.

[42] S. Halevi and V. Shoup, "HElib: Software library for homomorphic encryption," 2013, http://github.com/shaih/HElib.git.

[43] M. R. Albrecht, R. Player, and S. Scott, "On the concrete hardness of learning with errors," *J. Mathematical Cryptology*, vol. 9, no. 3, pp. 169–203, 2015.

[44] M. R. Albrecht, "On dual lattice attacks against small-secret LWE and parameter choices in HElib and SEAL," Cryptology ePrint Archive, Report 2017/047, 2017, http://eprint.iacr.org/2017/047.

Myungsun Kim received a B.S. degree in computer science and engineering from Sogang University, Seoul, Korea, in 1994 and an M.S. degree in computer science and engineering from the Information and Communications University (ICU), Daejeon, in 2002. He received a Ph.D. degree in mathematics from Seoul National University (SNU), Seoul, in 2012. Currently, he is an assistant professor in the Department of Information Security at The University of Suwon. His research interests include multiparty computation in cryptography.

Hyung Tae Lee received B.S., M.S., and Ph.D. degrees in mathematics from Seoul National University, Korea, in 2006, 2008, and 2013, respectively. He is currently an assistant professor with the Division of Computer Science and Engineering, College of Engineering, Chonbuk National University, Korea. Prior to that, he was a research fellow with Nanyang Technological University, Singapore. His research interests include computational number theory and cryptography.

San Ling received a B.A. degree in mathematics from the University of Cambridge in 1985 and a Ph.D. degree in mathematics from the University of California, Berkeley in 1990. Since April 2005, he has been a professor with the Division of Mathematical Sciences, School of Physical and Mathematical Sciences, Nanyang Technological University, Singapore. Prior to that, he was with the Department of Mathematics at the National University of Singapore. His research fields include arithmetic modular curves and applications of number theory to combinatorial design, coding theory, cryptography and sequences.

Benjamin Hong Meng Tan received a B.Sc degree in mathematics from Nanyang Technological University, Singapore in 2013. He is currently a Ph.D. student at the Division of Mathematical Sciences, School of Physical and Mathematical Sciences, Nanyang Technological University, Singapore. His research interests include cryptography, multiparty computation and secure cloud computing.

Huaxiong Wang obtained a Ph.D. in mathematics from University of Haifa, Israel, in 1996, and a Ph.D. in computer science from University of Wollongong, Australia, in 2001. He is currently an associate professor with the Division of Mathematical Sciences, School of Physical and Mathematical Sciences, Nanyang Technological University, Singapore. His research interests include cryptography, information security, coding theory, combinatorics, and theoretical computer science.