Efficient Techniques for Privacy-Preserving Sharing of Sensitive Information

Emiliano De Cristofaro[1,*], Yanbin Lu[2,*], Gene Tsudik[3]

[1] PARC, [2] Google, Inc., [3] UC Irvine

***Abstract*—The need for controlled (privacy-preserving) sharing of sensitive information occurs in many different and realistic everyday scenarios, ranging from national security to social networking. A typical setting involves two parties: one seeks information from the other without revealing the interest while the latter is either willing, or compelled, to share only the requested information. This poses two challenges: (1) how to enable this type of sharing such that parties learn no information beyond what they are entitled to, and (2) how to do so efficiently, in real-world practical terms. This paper explores the notion of Privacy-Preserving Sharing of Sensitive Information (PPSSI), and provides two concrete and efficient instantiations, modeled in the context of simple database querying. Proposed techniques function as a *privacy shield* to protect parties from disclosing more than the required minimum of their respective sensitive information. PPSSI deployment prompts several challenges, that are addressed in this paper. Extensive experimental results attest to the practicality of attained privacy features and show that they incur quite low overhead (about 10% slower than standard MySQL).**

I. Introduction

In today's increasingly digital world, there is often a tension between safeguarding privacy and sharing information. On the one hand, sensitive data needs to be kept confidential; on the other hand, data owners are often motivated or forced to share sensitive information. Consider the following examples:

- *Aviation Safety:* The Department of Homeland Security (DHS) checks whether any passengers on each flight from/to the United States must be denied boarding or disembarkation, based on several secret lists, including the *Terror Watch List* (TWL) [23]. Today, airlines surrender their passenger manifests to the DHS, along with a large amount of sensitive information, including credit card numbers [44]. Besides its obvious privacy implications, this modus operandi poses liability issues with regard to mostly innocent passengers' data and concerns about possible data loss. (See [10] for a litany of recent incidents where large amounts sensitive data were lost or mishandled by government agencies.) Ideally, the DHS would obtain information pertaining *only* to passengers on one of its watch-lists, without disclosing any information to the airlines.
- *Law Enforcement:* An investigative agency (e.g., the FBI) needs to obtain electronic information about a suspect from other agencies, e.g., the local police, the military, the DMV, the IRS, or the suspect's employer. In many cases, it is dangerous (or simply forbidden) for the FBI to disclose the subjects of its investigation. Whereas, the other party cannot disclose its entire dataset and trust the FBI to only extract desired information. Furthermore, FBI requests might need to be pre-*authorized* by some appropriate authority (e.g., a federal judge). This way, the FBI can only obtain information related to authorized requests.
- *Healthcare:* A health insurance company needs to retrieve information about its client from other entities, such as other insurance carriers or hospitals. The latter cannot provide any information on other patients and the former cannot disclose the identity of the target client.

Other examples of sensitive information sharing include collaborative botnet detection [38] (i.e., service providers share their logs for the sole purpose of identifying common anomalies), interest sharing from smartphones [16], or preventing cheating in online gaming [7].

Motivated by above examples, this paper develops the architecture for **Privacy-Preserving Sharing of Sensitive Information (PPSSI)**, and proposes two efficient and secure instantiations that function as a *privacy shield* to protect parties from disclosing more than the required minimum of sensitive information. We model PPSSI in the context of simple database-querying applications with two parties: a *server*, in possession of a database, and a *client*, performing disjunctive equality queries. In terms of one of the examples above, the airline company (the server) has a database with passenger information, while the DHS (the client) poses queries corresponding to the TWL.

Intended Contributions. In this paper, we explore the notion of Privacy-Preserving Sharing of Sensitive Information (PPSSI). Our main building blocks are efficient Private Set Intersection (PSI) techniques. During the design of PPSSI, we address several challenges stemming from adapting PSI to realistic database settings. Our extensive experimental evaluation demonstrates that our techniques incur very low overhead compared to standard (non privacy-preserving)

[*] Work done while at UC Irvine.

MySQL. All source code is publicly available.[1]

Organization. In next section, we introduce PPSSI syntax, along with its privacy requirements, and review PSI definitions. After reviewing related work in Section III, in Section IV, we discuss the insecurity of a strawman approach obtained with a naïve adaptation of PSI techniques to PPSSI. Then, Section V introduces a secure PPSSI approach using a novel database encryption mechanism. Next, in Section VI, we consider another approach geared for very large databases. Section VII presents our experimental analysis, and Section VIII concludes the paper by discussing future work.

II. Preliminaries

This section introduces Privacy-Preserving Sharing of Sensitive Information (PPSSI), formalizes its privacy requirements, and overviews Private Set Intersection (PSI) – our main building block.

A. PPSSI Syntax & Notation

We model PPSSI in the context of simple database querying. In it, a server maintains a database, DB, containing w records with m attributes $(attr_1, \cdots, attr_m)$. We denote $DB = \{(R_j)\}_{j=1}^{w}$. Each record $R_j = \{val_{j,l}\}_{l=1}^{m}$, where $val_{j,l}$ is R_j's value for attribute $attr_l$. A client poses simple disjunctive SQL queries, such as:

$$\begin{aligned}&\text{SELECT * FROM DB}\\&\text{WHERE } (attr_1^* = val_1^* \text{ OR } \cdots \text{ OR } attr_v^* = val_v^*)\end{aligned} \quad (1)$$

As a result of the query, the client gets all records in DB satisfying *where* clause, and nothing else. Whereas, the server learns nothing about any $\{attr_i^*, val_i^*\}_{1\leq i\leq v}$. We assume that the database schema (format) is known to the client. Furthermore, without loss of generality, we assume that the client only queries searchable attributes.

In an alternative version supporting *authorized queries*, we require the client to receive query authorizations from a mutually trusted offline *Certification Authority* (CA) prior to interacting with the server. That is, the client outputs matching records only if the client holds pertinent authorizations for $(attr_i^*, val_i^*)$.

Our notation is reflected in Table I. In addition, we use $Enc_k(\cdot)$ and $Dec_k(\cdot)$ to denote, respectively, symmetric key encryption and decryption (under key k). Public key encryption and decryption, under keys pk and sk, are denoted as $E_{pk}(\cdot)$ and $E_{sk}(\cdot)^{-1}$, respectively. $\sigma = \mathsf{Sign}_{sk}(M)$ denotes a digital signature computed over message M using secret key sk. Operation $\mathsf{Vrfy}_{pk}(\sigma, M)$ returns 1 or 0 indicating whether σ is a valid signature on M. $\mathbb{Z}_N^*$ refers to a composite-order RSA group, where N is the RSA modulus. We use d to denote RSA private key and e to denote corresponding public key. We use $\mathbb{Z}_p^*$ to denote a cyclic group with a subgroup of order q, where p and q are large primes, and $q|p-1$. Let G_0, G_1 be two multiplicative cyclic groups of prime order p. We use $\hat{e} : G_0 \times G_0 \rightarrow G_1$ to denote a bilinear map. $ZKPK$ is used to denote zero-knowledge proof of knowledge. We use $H(\cdot), H_1(\cdot), H_2(\cdot), H_3(\cdot)$ to denote different hash functions, modeled as random oracles. In practice, we implement $H(m)$, $H_1(m)$, $H_2(m)$, $H_3(m)$ as SHA-1$(0||m)$, SHA-1$(1||m)$, SHA-1$(2||m)$, SHA-1$(3||m)$.

B. Informal Privacy Requirements

We now define PPSSI privacy requirements for both standard and authorized queries.[2] We consider both Honest-but-Curious (HbC) adversaries and malicious adversaries. An HbC adversary faithfully follows all protocol's specifications (but might attempt to infer additional information during or after protocol execution). Whereas, malicious adversaries may arbitrarily deviate from the protocol.

Privacy requirements are as follows:

- *Server Privacy.* The client learns no information about any record in server's database that does not satisfy the *where* $(attr_i^* = val_i^*)$ clause(s).
- *Server Privacy (Authorized Queries).* Same as "Server Privacy" above, but, in addition, the client learns no information about any record satisfying the *where* $(attr_i^* = val_i^*)$ clause, unless the $(attr_i^*, val_i^*)$ query is authorized by the CA.
- *Client Privacy.* The server learns nothing about any client query parameters, i.e., all $attr_i^*$ and val_i^* (nor about its authorizations, for authorized queries), except the number of queried attributes.
- *Client Unlinkability.* The server cannot determine (with probability non-negligibly exceeding $1/2$) whether any two client queries are related.
- *Server Unlinkability.* For any two queries, the client cannot determine whether any record in the server's database has changed, except for the records that are learned (by the client) as a result of both queries.
- *Forward Security (Authorized Queries).* The client cannot violate Server Privacy with regard to prior interactions, using authorizations obtained later.

Note that Forward Security and Unlinkability requirements are crucial in many practical scenarios. Referring to one example in Section I, suppose that the FBI queries an employee database without having authorization for a given suspect, e.g., Alice. Server Privacy (Authorized Queries) ensures that the FBI does not obtain any information about Alice. However, unless Forward Security is guaranteed, if the FBI later obtains authorization for Alice, it could inappropriately recover her file from the (recorded) protocol transcript. On the other hand, Unlinkability keeps one party from noticing changes in other party's input. In particular, unless Server Unlinkability is guaranteed, the client can always detect whether the server updates its database between two interactions. Unlinkability also minimizes the

[1] Source code is available at http://sprout.ics.uci.edu/projects/iarpa-app/index.php?page=code.php.

[2] To ease clarity, our definitions hereby are only informal – formal security arguments (as well as proofs), are presented later in the paper, along with protocol constructions, following traditional ideal model/real world arguments.

risk of privacy leaks. Without Client Unlinkability, if the server learns that the client's queries are the same in two interactions and one of these query contents are leaked, the other query would be immediately exposed.

Finally, note that, on a conservative stance, we have assumed that the database contains no publicly-known records, however, public records can be queried using standard techniques, orthogonally to our privacy-preserving techniques presented in the rest of the paper.

C. Private Set Intersection (PSI)

Private Set Intersection (PSI) [26] constitutes our main building block. It allows two parties – a server and a client – to interact on their respective input sets, such that the client only learns the intersection of the two sets, while the server learns nothing beyond client's set size.

PSI with Data Transfer (PSI-DT): It involves a server, on input a set of w items, each with associated data record, $\mathcal{S} = \{(s_1, data_1), \cdots, (s_w, data_w)\}$, and a client, on input of a set of v items, $\mathcal{C} = \{c_1, \cdots, c_v\}$. It results in the client outputting $\{(s_j, data_j) \in \mathcal{S} \mid \exists c_i \in \mathcal{C} \; s.t. \; c_i = s_j\}$ and the server – nothing except v. This variant is useful whenever the server holds a set of records, rather than a simple set of elements.

Authorized PSI-DT (APSI-DT): It ensures that client input is *authorized* by a mutually trusted offline CA. Unless it holds pertinent authorizations, the client does not learn whether its input is in the intersection. At the same time, the server does not learn whether client's input is authorized, i.e., verification of client authorizations is performed obliviously. More specifically, APSI-DT involves a server, on input of a set of w items: $\mathcal{S} = \{(s_1, data_1), \cdots, (s_w, data_w)\}$, and a client, on input of a set of v items with associated authorizations (typically, in the form of digital signatures), $\mathcal{C} = \{(c_1, \sigma_i) \cdots, (c_v, \sigma_v)\}$. It results in client outputting $\{(s_j, data_j) \in \mathcal{S} \mid \exists (c_i, \sigma_i) \in \mathcal{C} \; s.t. \; c_i = s_j \wedge \mathsf{Vrfy}_{pk}(\sigma_i, c_i) = 1\}$ (where pk is CA's public key).

We also distinguish between (A)PSI-DT protocols based on whether or not they support *pre-distribution*:

(A)PSI-DT with pre-distribution: The server can "pre-process" its input set independently from client input. This way, the server can *pre-distribute* its (processed) input before protocol execution. Both pre-processing and pre-distribution can be done offline, once for all possible clients.

(A)PSI-DT without pre-distribution: The server cannot pre-process and pre-distribute its input.

Note that pre-distribution precludes Server Unlinkability, since server input is assumed to be fixed. Similarly, in the context of authorized protocols with pre-distribution, Forward Security cannot be guaranteed.

III. Related Work

A number of cryptographic primitives provide privacy properties resembling those listed in Section II-B. We overview them below.

Secure Two-Party Computation (2PC). 2PC allows two parties, on input x and y, respectively, to privately compute the output of a public function f over (x, y). Both parties learn nothing beyond what can be inferred from the output of the computation. Although one could implement PPSSI with generic 2PC, it is usually far more efficient to have dedicated protocols, as 2PC incurs high computational overhead and involves several communication rounds.

Oblivious Transfer (OT). OT [42] involves a sender holding n secret messages and a receiver willing to retrieve the i-th among sender's messages. It ensures that the sender does not learn which message is retrieved, and the receiver learns no other message. While the OT functionality somehow resembles PPSSI requirements, note that, in PPSSI, receiver's inputs are query keywords, whereas, in OT, they are indices.

Oblivious Keyword Search [40]. This primitive is akin to a special case of PSI-DT, where Client input is a singleton and Server input is a multi-set. We discuss how to handle multi-sets using PSI-DT in Section V-D.

Private Information Retrieval (PIR). PIR [12] allows a client to retrieve an item from a server database, (1) without revealing which item it is retrieving, and (2) incurring a communication overhead strictly lower than $O(n)$, where n is the database size. Observe that, in PIR, privacy of server's database is not protected – the client may receive additional bits of information, besides the records requested. Symmetric PIR (SPIR) [28] additionally offers server privacy, thus achieving OT with communication overhead lower than $O(n)$. However, similar to OT, a client of a symmetric PIR needs to input the index of the desired item in server's database – an unrealistic assumption for PPSSI. An extension to keyword-based retrieval is known as Keyword-PIR (KPIR) [11]. However, KPIR still does not consider server privacy and it involves multiple rounds of PIR executions.

Searchable Encryption (SE). Symmetric Searchable Encryption (SSE) [45], [14], [15] allows a client to store, on an untrusted server, messages encrypted using a symmetric-key cipher under its own secret key. Later, the client can search for specific keywords by giving the server a trapdoor that does not reveal keywords or plaintexts. Boneh et al. [6] later extended SSE to the public-key setting, i.e., anyone can use client's public key to encrypt and route messages through an untrusted server (e.g., a mail server). The client can then generate search tokens, based on its private key, to let the server identify messages including specific keywords. We conclude that Searchable Encryption targets related yet different scenarios compared to PPSSI.

Privacy-Preserving Database Query (PPDQ). PPDQ techniques can be distinguished into two kinds. The first one is similar to SSE: the client encrypts its data, outsources encrypted data to an untrusted service provider (while not maintaining copies), and queries the service provider at will. In addition to simple equality predicates supported by SSE, solutions like [29], [31], [5] support general SQL operations. Again, this setting is often different from PPSSI,

$attr_l$	lth attribute in the database schema	$ctr_{j,l}$	number of times where $val_{j',l} = val_{j,l}, \forall j' <= j$
R_j	jth record in the database	$tag_{j,l}$	tag for $attr_l, val_{j,l}$
$val_{j,l}$	value in R_j corresponding to $attr_l$	$k'_{j,l}$	key used to encrypt k_j
k_j	key used to encrypt R_j	$k''_{j,l}$	key used to encrypt index j
er_j	encryption of R_j	$ek_{j,l}$	encryption of key k_j
$tk_{j,l}$	token evaluated over $attr_l, val_{j,l}$	$eind_{j,l}$	encryption of index j

TABLE I: Notation.

as that data, although stored by the server, belongs to the client; thus, there is no privacy restriction against the client. Moreover, these solutions do not provide provably-secure guarantees, but are based on statistical (probabilistic) methods.

The second kind of PPDQ is closely related to private predicate matching. Olumofin and Goldberg [41] propose a transition from block-based PIR to SQL-enabled PIR. As opposed to PPSSI, however, server's database is assumed to be public, thus, its privacy is not protected. Then, Kantarcioĝlu and Clifton [33] consider a scenario where client matches classification rules against server's database. However, they assume the client's rule set to be fixed in advance and known to the server. Additional work, such as [43], [13], requires several independent, mutually-trusted, and non-colluding parties. Murugesan et al. [37] also allow "fuzzy" matching, yet their solution requires a number of (expensive) cryptographic operations (i.e., public-key homomorphic operations) quadratic in the size of parties' inputs, while we aim at constructing scalable solutions with linear complexity.

IV. A Strawman Approach

Looking at definitions in Section II-C, it seems that PPSSI can be realized by simply instantiating PSI-DT protocols (or APSI-DT for authorized queries). We outline this *strawman* approach below and show that it is not secure.

For each record, consider the hash of every attribute-value pair $(attr_l, val_{j,l})$ as a set element, and R_j as its associated data. Server "set" then becomes:

$$\mathcal{S} = \{(H(attr_l, val_{j,l}), R_j)\}_{1 \le l \le m, 1 \le j \le w}$$

Client "set" is: $\mathcal{C} = \{H(attr^*_i, val^*_i)\}_{1 \le i \le v}$, i.e., elements corresponding to the *where* clause in Equation 1. Optionally, for authorized queries, $\mathcal{C}$ is accompanied by signatures σ_i over $H(attr^*_i, val^*_i)$, following the APSI-DT syntax. Parties engage in an (A)PSI-DT interaction; at the end of it, the client obtains all records matching its query.

The strawman approach faces two security issues:

Challenge 1: Multi-Sets. While most databases include duplicate values (e.g., "gender=male"), PSI-DT and APSI-DT definitions assume that sets do not include duplicates.[3] If server set contains duplicated values, the corresponding messages (pseudorandom function values computed over the duplicated values) to the client would be identical and the client would learn all patterns and distribution frequencies. This raises a serious concern, as actual values can be often inferred from their frequencies. For example, consider a large database where one attribute reflects "employee blood type": since blood type frequencies are well-known for general population, distributions for this attribute would essentially reveal the plaintext, similar to deterministic encryptions.

Challenge 2: Data Pointers. To enable querying by any attribute, each record – R_j – must be separately encrypted m times, i.e., once for each attribute. As this would result in high storage/bandwidth overhead, one could encrypt each R_j with a unique symmetric key k_j and then using k_j (instead of R_j) as data associated with $H(attr_l, val_{j,l})$. Although this would reduce the overhead, it would trigger another issue: in order to use the key – rather than the actual record – as the associated "data" in the (A)PSI-DT protocol, we would need to store a pointer to the encrypted record alongside each $H(attr_l, val_{j,l})$. This would allow the client to identify all $H(attr_l, val_{j,l})$ corresponding to a given encrypted record by simply identifying all $H(attr_l, val_{j,l})$ with associated data pointers equal to the given records. Such a (potential) privacy leak would be aggravated if combined with the previous "attack" on multi-sets: given two encrypted records, the client could establish their similarity based on the number of equal attributes.

Remark. We stress that the above issues do not only apply to the naïve adaptation of Private Set Intersection techniques to the specific PPSSI setting but also to privacy-preserving data mining [22], information sharing across databases [1].

V. The First PPSSI Approach

We now present our PPSSI construction that is both secure and reasonably practical. Like the strawman approach, it relies on (A)PSI-DT. However, it addresses aforementioned challenges by introducing a novel database-encryption technique. In order to guarantee both *Server Unlinkability* and *Forward Security*, we use (A)PSI-DT *without* pre-distribution.

Our approach is illustrated in Figure 1. In step 1, the client and the server engage in the *oblivious* computation of `Token` function: at the end of it, the client obtains $tk_i = \texttt{Token}(c_i)$, where $c_i = H(attr^*_i, val^*_i)$. Note that the server learns nothing about c_i or tk_i. `Token` function is computed using an (A)PSI-DT protocol, thus, different (A)PSI-DTs instantiate it differently. We introduce it to provide a level abstraction independent from the underlying (A)PSI-DT instantiation.

[3] Note that some PSI constructs (e.g., [35]) support multi-sets, however, their performance is not promising as they incur quadratic computational overhead (in the size of the sets), as opposed to more recent (A)PSI-DT protocols with linear complexity (e.g., [32], [20], [17]). Also, they support neither *data transfer* nor *authorization*.

- Client's input: $\{c_i, \sigma_i\}_{1\leq i\leq v}$, where: $c_i = H(attr_i^*, val_i^*)$. σ_i is only used for APSI-DT protocols.
- Server's input: $\{s_{j,l}\}_{1\leq j\leq w, 1\leq l\leq m}, \{R_j\}_{1\leq j\leq w}$, where: $s_{j,l} = H(attr_l, val_{j,l})$

1) Client $\xLeftrightarrow{\text{Obliviously computes:}\{tk_i \leftarrow \texttt{Token}(c_i)\}_{\forall i}}$ Server
2) Server: $\mathbf{EDB} \leftarrow \texttt{EncryptDatabase}(\texttt{Token}(\cdot), \{R_j\}_{1\leq j\leq w})$
3) Server $\xrightarrow{\mathbf{EDB}}$ Client
4) Client: $\forall_{1\leq i\leq v}\mathbf{R_i} \leftarrow \texttt{Lookup}(tk_i, \mathbf{EDB})$, Output $\mathbf{R_1} \cup \cdots \cup \mathbf{R_v}$.

Fig. 1: Outline of our first PPSSI approach.

Scheme name	`Token` definition	PSI category
DT10-1 (Figure 3 of [20])	$\texttt{Token}(c) = ([(\prod_{i=1}^{v} c_i) \cdot g^{R_c}]/c)^{R_s} \bmod p$	PSI-DT without pre-distribution
DT10-APSI (Figure 2 of [20])	$\texttt{Token}(c) = ([(\prod_{i=1}^{v} \sigma_i)^2 \cdot g^{R_c}]^e/c^2)^{R_s} \bmod N$	APSI-DT without pre-distribution

TABLE II: `Token` definition for (A)PSI-DT without pre-distribution (c_i, σ_i is defined in Figure 1 and $c \in \{c_i\}_{1\leq i\leq v}$)

In step 2, the server runs `EncryptDatabase` procedure – described in Algorithm 1 and discussed in Section V-A – and creates the encrypted database, $\mathbf{EDB}$ that is transferred to the client in step 3. Finally, in step 4, the client runs `Lookup` procedure – illustrated in Algorithm 2 and discussed in Section V-B – using tk_i tokens over $\mathbf{EDB}$; at the end of it, the client obtains the set of records satisfying its query.

Our protocol can be used with any (A)PSI-DT, however, we use the variants without pre-distribution, since they provide Server Unlinkability and Forward Security. Following a thorough experimental analysis (shown in Appendix of [19]), we select the PSI-DT protocol from [20] (denoted as **DT10-1**) and its APSI-DT counterpart from [20] (denoted as **DT10-APSI**) for authorized queries. These protocols were proven secure against HbC adversaries [20]. However, it was later shown that, with very similar overhead, they can be extended to achieve security against malicious adversaries [18].

For the sake of completeness, we define `Token` function for the selected (A)PSI-DT constructions in Table II. Note that both `Token` definitions involve random values R_c and R_s contributed by the client and the server respectively. `Token` function can be directly evaluated by the server over its own inputs (as in step 9 of Algorithm 1) only after step 1 of Figure 1 where necessary information regarding R_c was sent as part of the oblivious computation protocol by the client to the server. These random values are selected at the beginning of and kept fixed throughout the PPSSI protocol execution. They are chosen independently, for each invocation, in order to guarantee *Server Unlinkability* and *Forward Security*.

We present the complete details of `Token`'s oblivious computation in Figure 2 and Figure 3. Both instantiations incur linear computation overhead with respect to client and server set size.

Compared to the strawman approach, we modified the "encryption" technique: rather than (directly) using a symmetric-key encryption scheme, the `EncryptDatabase` procedure is invoked.

- Public input: p, q
- Client's private input: $\{c_i\}_{\forall i}$
- All operations are modulo p

1) Client: $PCH \leftarrow \prod_{i=1}^{v} c_i,\ R_c \overset{r}{\leftarrow} \mathbb{Z}_q^*,\ X \leftarrow PCH \cdot g^{R_c}$,
   $\forall i, PCH_i \leftarrow \frac{PCH}{c_i}, R_{c,i} \overset{r}{\leftarrow} \mathbb{Z}_q^*, y_i \leftarrow PCH_i \cdot g^{R_{c,i}}$
2) Client $\xrightarrow{X,\{y_i\}_{\forall i}}$ Server
3) Server: $R_s \overset{r}{\leftarrow} \mathbb{Z}_q^*, Z \leftarrow g^{R_s}, \forall i, z_i \leftarrow y_i^{R_s}$
4) Server $\xrightarrow{Z,\{z_i\}_{\forall i}}$ Client
5) Client: $\forall i, \texttt{Token}(c_i) \leftarrow z_i \cdot Z^{R_c} \cdot Z^{-R_{c,i}}$

Fig. 2: Oblivious computation of $\texttt{Token}(\cdot)$ using DT10-1.

- Public input: e, N
- Client's private input: $\{c_i\}_{\forall i}$
- CA's private input: d
- All operations are modulo N

1) CA: $\forall i,\ \sigma_i \leftarrow (c_i)^d$
2) CA $\xrightarrow{\{\sigma_i\}_{\forall i}}$ Client
3) Client: $PCH \leftarrow \prod_{i=1}^{v} c_i,\ PCH^* \leftarrow \prod_{i=1}^{v} \sigma_i,\ R_c \overset{r}{\leftarrow} \mathbb{Z}_{n/4}$,
   $\forall i, PCH_i^* \leftarrow PCH^*/\sigma_i,\ y_i \leftarrow (PCH_i^*)^2 \cdot g^{R_{c,i}}$
   $X \leftarrow (PCH^*)^2 \cdot g^{R_c}$
4) Client $\xrightarrow{X,\{y_i\}_{\forall i}}$ Server
5) Server: $R_s \overset{r}{\leftarrow} \mathbb{Z}_{n/4}, Z \leftarrow g^{e\cdot R_s}, \forall i, z_i \leftarrow y_i^{e\cdot R_s}$
6) Server $\xrightarrow{Z,\{z_i\}_{\forall i}}$ Client
7) Client: $\forall i, \texttt{Token}(c_i) \leftarrow z_i \cdot Z^{R_c} \cdot Z^{-R_{c,i}}$

Fig. 3: Oblivious computation of $\texttt{Token}(\cdot)$ using DT10-APSI.

A. Database Encryption with counters

We illustrate `EncryptDatabase` procedure in Algorithm 1. It takes as input the definition of the `Token` function, and server's record set. It consists of two "phases": (1) *Record-level* and (2) *Lookup-Table* encryptions.

Record-level encryption is relatively trivial (lines 1–6): first, the server shuffles record locations; then, it pads each R_j up to a fixed maximum record size, picks a random symmetric key k_j, and encrypts R_j as $er_j = Enc_{k_j}(R_j)$.

Lookup-Table (LTable) encryption (lines 8–15) pertains to attribute name and value pairs. It enables efficient lookup and record decryption. In step 8, the server hashes an attribute-value pair and uses the result as input to `Token` function in step 9. In step 10, we use the concatenation of `Token` output and a counter, $ctr_{j,l}$, in order to compute the tag $tag_{j,l}$, later used as a lookup tag during client query. We use $ctr_{j,l}$ to denote the index of duplicate value for

Algorithm 1: `EncryptDatabase` Procedure.

```
input  : Function Token(·) and record set $\{R_j\}_{1 \le j \le w}$
output: Encrypted Database $\mathbf{EDB}$
 1: Shuffle $\{R_j\}_{1 \le j \le w}$
 2: $maxlen \leftarrow$ max length among all $R_j$
 3: for $1 \le j \le w$ do
 4:     Pad $R_j$ to $maxlen$;
 5:     $k_j \xleftarrow{r} \{0,1\}^{128}$;
 6:     $er_j \leftarrow Enc_{k_j}(R_j)$;
 7:     for $1 \le l \le m$ do
 8:         $hs_{j,l} \leftarrow H(attr_l, val_{j,l})$;
 9:         $tk_{j,l} \leftarrow \mathtt{Token}(hs_{j,l})$;
10:         $tag_{j,l} \leftarrow H_1(tk_{j,l}||ctr_{j,l})$;
11:         $k'_{j,l} \leftarrow H_2(tk_{j,l}||ctr_{j,l})$;
12:         $k''_{j,l} \leftarrow H_3(tk_{j,l}||ctr_{j,l})$;
13:         $ek_{j,l} \leftarrow Enc_{k'_{j,l}}(k_j)$;
14:         $eind_{j,l} \leftarrow Enc_{k''_{j,l}}(j)$;
15:         $\mathbf{LTable}_{j,l} \leftarrow (tag_{j,l}, ek_{j,l}, eind_{j,l})$;
16:     end for
17: end for
18: Shuffle $\mathbf{LTable}$ with respect to $j$ and $l$;
19: $\mathbf{EDB} \leftarrow \{\mathbf{LTable}, \{er_j\}_{1 \le j \le w}\}$;
```

Algorithm 2: `Lookup` Procedure.

```
input  : Search token $tk$ and encrypted database
         $\mathbf{EDB} = \{\mathbf{LTable}, \{er_j\}_{1 \le j \le w}\}$
output: Matching record set $\mathbf{R}$
 1: $ctr \leftarrow 1$;
 2: while $\exists tag_{j,l} \in \mathbf{LTable}\ s.t.\ tag_{j,l} = H_1(tk||ctr)$ do
 3:     $k'' \leftarrow H_3(tk||ctr)$;
 4:     $j' \leftarrow Dec_{k''}(eind_{j,l})$;
 5:     $k' \leftarrow H_2(tk||ctr)$;
 6:     $k \leftarrow Dec_{k'}(ek_{j,l})$;
 7:     $R_j \leftarrow Dec_k(er_{j'})$;
 8:     $\mathbf{R} \leftarrow \mathbf{R} \cup R_j$;
 9:     $ctr \leftarrow ctr + 1$;
10: end while
```

the l-th attribute. In other words, $ctr_{j,l}$ is the counter of occurrences of $val_{j',l} = val_{j,l}, \forall j' <= j$. For example, the third occurrence of value "Smith" for attribute "Last Name" will have the counter equal to 3. The counter guarantees that duplicate $(attr, val)$ pairs correspond to different tags, thus addressing Challenge 1. Next, the server computes $k'_{j,l} = H_2(tk_{j,l}||ctr_{j,l})$ and $k''_{j,l} = H_3(tk_{j,l}||ctr_{j,l})$. Note that $k'_{j,l}$ is used for encrypting symmetric key k_j. Whereas, $k''_{j,l}$ is used for encrypting the index of R_j. In step 13, the server encrypts k_j as $ek_{j,l} = Enc_{k'_{j,l}}(k_j)$. Then, the server encrypts $eind_{j,l} = Enc_{k''_{j,l}}(j)$. The encryption of index (data pointer) guarantees that the client cannot link two tags belonging to the same record, thus addressing Challenge 2. In step 15, the server inserts each $tag_{j,l}$, $ek_{j,l}$ and $eind_{j,l}$ into LTable, which is $\{tag_{j,l}, ek_{j,l}, eind_{j,l}\}_{1 \le j \le w, 1 \le l \le m}$. Next, the server shuffles LTable (step 18). The resulting encrypted database, $\mathbf{EDB}$, is composed of LTable and $\{er_j\}_{j=1}^{w}$ (step 19).

B. Lookup with counters

We now discuss `Lookup` procedure shown in Algorithm 2. It is used by the client to obtain the query result, i.e., to search $\mathbf{EDB}$ for all records that match client's search tokens.

In step 1, the client initializes a counter to 1. Next, it searches $\mathbf{LTable}$ for tag $tag_{j,l} = H_1(tk||counter)$. If there is a match, the client attempts to recover the record associated with $tag_{j,l}$. To do so, the client needs to locate the associated record: it computes $k'' = H_3(tk||ctr)$ and recovers $j' = Dec_{k''}(eind_{j,l})$. Note that $er_{j'}$ now corresponds to the associated record. To decrypt $er_{j'}$, the client first recovers the key k used to encrypt $er_{j'}$, by computing $k' = H_2(tk||ctr)$ and obtaining $k = Dec_{k'}(ek_{j,l})$. Finally, the client recovers R_j by decryption, i.e., $R_j = Dec_k(er_{j'})$.

There are several ways for the client to store $\mathbf{LTable}$. Hash table storage is most efficient as it only requires constant lookup time. We can also use binary search tree, which takes sublinear lookup time, but it requires ordering $\mathbf{LTable}$ first.

C. Example of Correctness

Assume that server's database includes the attribute "gender" with two occurrences of value "male". In Algorithm 1, the same tk (step 9) will be generated for the two occurrences of ("gender", "male"). However, for the first occurrence, $tag = H_1(tk||1), k' = H_2(tk||1), k'' = H_3(tk||1)$ while, for the second occurrence, $tag = H_1(tk||2), k' = H_2(tk||2), k'' = H_3(tk||2)$.

Suppose that the client searches for records matching "gender = male", it first derives tk (step 1 of Figure 1). Next, it matches $H_1(tk||1)$ in $\mathbf{LTable}$, derives keys $k' = H_2(tk||1), k'' = H_3(tk||1)$, and recovers the index in step 4 and the record in step 7 of Algorithm 2. It also looks for $H_1(tk||2)$ and performs the same operations as before, except that $k' = H_2(tk||2), k'' = H_3(tk||2)$. Finally, the client looks for $H_1(tk||3)$: since it finds no match, it terminates.

D. Challenges Revisited

We claim that our approach addresses Challenge 1 and 2, discussed in Section IV. The intuition is as follows:

Multi-sets: The use of counters during database encryption makes each $tag_{j,l}$ (resp. $ek_{j,l}$, $eind_{j,l}$) distinct in $\mathbf{LTable}$, thus hiding plaintext patterns.

Data Pointers: Storing $eind_{j,l}$ (rather than j) in $\mathbf{LTable}$, prevents the server from exposing the relationship between an entry $\mathbf{LTable}_{j,l}$ and its associated record R_j.

E. Security Analysis of First PPSSI Approach

1) Cryptographic primitive security definition: Below, we review the standard security definitions for some cryptographic primitives.

First, we define the security for a hash function $H_s(\cdot)$ based on the following experiment. The experiment: $\mathsf{HashColl}_{\mathcal{A}}$:

1) A key s is generated by a hash key generator.
2) The adversary $\mathcal{A}$ is given s and outputs x, x'.
3) The output of the experiment is defined to be 1 if and only if $x \neq x'$ and $H_s(x) = H_s(x')$.

Definition 1: We say that a hash function is (t, ϵ) collision resistant if for any adversary $\mathcal{A}$ bounded by time t, there exists a negligible parameter ϵ such that

$$\mathsf{Pr}[\mathsf{HashColl}_{\mathcal{A}} = 1] \le \epsilon$$

Then, we define the security for a semantic encryption scheme. The experiment: $\mathsf{PrivEnc}_{\mathcal{A}}$:

1) $\mathcal{A}$ outputs a pair of messages m_0, m_1.
2) A random bit b is chosen. Give $c \leftarrow Enc(m_b)$ to $\mathcal{A}$.
3) $\mathcal{A}$ may keep querying $Enc(\cdot)$.
4) $\mathcal{A}$ outputs b'.
5) The output of the experiment is defined to be 1 if and only if $b = b'$.

Definition 2: We say that an encryption scheme is (t, ϵ) secure if for any adversary $\mathcal{A}$ bounded by time t, there exists a negligible parameter ϵ such that

$$\mathsf{Pr}[\mathsf{PrivEnc}_{\mathcal{A}} = 1] \le \frac{1}{2} + \epsilon$$

Last, we define the security for an unpredictable function. The experiment: $\mathsf{PrivToken}_{\mathcal{A}}$:

1) $\mathcal{A}$ is allowed to query Token$(\cdot)$ for polynomial number of times.
2) $\mathcal{A}$ outputs a pair (x, y) where x is not queried before.
3) The output of the experiment is defined to be 1 if and only if Token$(x) = y$.

Definition 3: We say that a Token function is (t, ϵ) unpredictable if for any adversary $\mathcal{A}$ bounded by time t, there exists a negligible parameter ϵ such that

$$\mathsf{Pr}[\mathsf{PrivToken}_{\mathcal{A}} = 1] \le \epsilon$$

2) Security against Honest-but-Curious/Malicious Client: We use q_i to denote the ith query of the form $(attr, val)$ issued by the client and use Q_i to denote all records matching query q_i.

We define security against Honest-but-Curious/Malicious client by comparing its view under real model with that under ideal model. In the ideal model, there is a trusted third party (TTP) serving as an honest server who, in response to the query q_i, only replies Q_i.

We first consider Honest-but-Curious adversary and then analyze malicious adversary at the end of this section. We define a simulator SIM that attempts to simulate to a real-model client based on output from ideal-model TTP as follows:

Simulator SIM:

SIM is given input $\{q_1, \ldots, q_n\}$

1) SIM picks all the secret and public parameters.
2) SIM interacts with $\mathcal{A}$ as a real-model server during oblivious computation of Token (step 1 of Figure 1).
3) SIM sends $\{q_1, \ldots, q_n\}$ to the TTP and receives $\{Q_1, \ldots, Q_n\}$.
4) SIM runs some function on $\{Q_1, \ldots, Q_n\}$ and outputs the result to the client.

Note that we require SIM to take $\{q_1, \ldots, q_n\}$ at once and therefore does not handle adaptive query.

We then define an experiment for any adversary $\mathcal{A}$:

The experiment $\mathsf{SPriv}_{\mathsf{C},\mathcal{A}}$:

1) The adversary $\mathcal{A}$ outputs to the challenger a list of queries $\{q_1, \ldots, q_n\}$.
2) The challenger chooses a random bit $b \xleftarrow{r} \{0, 1\}$ and does one of the following:
   a) If $b = 0$, then the challenger interacts with $\mathcal{A}$ as a real-model server.
   b) If $b = 1$, then the challenger interacts with $\mathcal{A}$ as SIM$(\{q_1, \ldots, q_n\})$.
3) The adversary $\mathcal{A}$ outputs a bit b'.
4) The output of the experiment is defined to be 1 if $b' = b$, and 0 otherwise.

Definition 4: The first PPSSI approach is secure against honest-but-curious client if, for all probabilistic polynomial-time adversaries $\mathcal{A}$, there exists a probabilistic polynomial-time simulator SIM such that

$$\mathsf{Pr}[\mathsf{SPriv}_{\mathsf{C},\mathcal{A}} = 1] \le \frac{1}{2} + \epsilon$$

This definition ensures that the client in the real model does not get more or different information than the ideal implementation.

Theorem 1: If the hash function $H(\cdot)$, $H_1(\cdot)$, $H_2(\cdot)$, $H_3(\cdot)$ are (t_0, ϵ_0), (t_1, ϵ_1), (t_2, ϵ_2), (t_3, ϵ_3) collision resistant, Enc is a $(t_{enc}, \epsilon_{enc})$ semantic secure encryption, and Token is a (t_T, ϵ_T) unpredictable function, then first PPSSI approach is (t, ϵ)-secure against any probabilistic polynomial-time honest-but-curious client where $t \le min(t_0, t_1, t_2, t_3, t_{enc}, t_T) - w \cdot m \cdot t_{\text{Token}} - w \cdot t_{enc}$ and $\epsilon = \epsilon_0 + \epsilon_1 + \epsilon_2 + \epsilon_3 + 2\epsilon_T + \epsilon_{enc}$.

Proof: Our goal is to construct a simulator SIM such that $\mathcal{A}$ cannot tell the difference between the view when interacting with SIM and the view when interacting with real-model server. Our SIM is constructed as follows:

1) SIM picks all the secret and public parameters on behalf of a real-model server and publish all public parameters.
2) SIM interacts with $\mathcal{A}$ as a real-model server during oblivious computation of Token (step 1 of Figure 1).
3) SIM queries TTP for $\{q_1, \ldots, q_n\}$ and gets back $\{Q_1, \ldots, Q_n\}$.
4) Let Q denote $\cup_i Q_i$. SIM generates $w - |Q|$ random records of the same length as any other message in Q. Let DB' denote the concatenation of Q and these random records. Note that $|DB'| = w$.

5) Use Algorithm 1 to encrypt DB' and returns encrypted database $\mathbf{EDB}'$ to the client.
6) SIM answers client's request for $H(\cdot), H_1(\cdot), H_2(\cdot), H_3(\cdot)$ as a random oracle.

We first analyze $\mathcal{A}$'s view between tags in $\mathbf{EDB}$ and tags in $\mathbf{EDB}'$. Note that a tag in $\mathbf{LTable}$ is computed as $H_1(\texttt{Token}(H(attr, val))||ctr)$. The only time that $\mathcal{A}$ observes difference is when (1) there exists $q_j, (attr, val)$ such that $H(q_j) = H(attr, val)$ while $q_j \neq (attr, val)$; (2) there exists two different pairs – $(attr', val')$, $(attr''$, $val'')$ – such that $H_1(\texttt{Token}(H(attr', val'))||ctr') = H_1(\texttt{Token}(H(attr'', val''))||ctr'')$; (3) $\mathcal{A}$ forges $\text{Token}(H(attr, val))$ for certain $(attr, val)$. This means finding a collision in $H(\cdot)$ or $H_1(\cdot)$, or breaking Token, which happens with probability at most $\epsilon' = \epsilon_0 + \epsilon_1 + \epsilon_T$ if $t + w \cdot m \cdot t_{\text{Token}} + t_{enc}$ is bounded by $min(t_0, t_1, t_T)$.

Next we analyze $\mathcal{A}$'s view between $(\{ek_{j,l}$, $eind_{j,l}\}_{1 \le l \le m}, er_j)_{1 \le j \le w}$ in $\mathbf{EDB}$ and those in $\mathbf{EDB}'$. For all $ek, eind, er$ whose corresponding tags do not match $\{q_1, \ldots, q_n\}$, $\mathcal{A}$ cannot tell the difference between them in $\mathbf{EDB}$ and in $\mathbf{EDB}'$ unless (1) $\mathcal{A}$ breaks symmetric encryption algorithm; (2) finds collision in $H_2(\cdot)$ or $H_3(\cdot)$; (3) $\mathcal{A}$ can forge $\text{Token}(H(attr, val))$ for certain $(attr, val)$. All these happen with probability at most $\epsilon'' = \epsilon_{enc} + \epsilon_2 + \epsilon_3 + \epsilon_T$ if $t + w \cdot m \cdot t_{\text{Token}} + w \cdot t_{enc}$ is bounded by $min(t_{enc}, t_2, t_3, t_T)$.

The claim follows when $t + w \cdot m \cdot t_{\text{Token}} + w \cdot t_{enc} \le min(t_0, t_1, t_2, t_3, t_{enc}, t_T)$. ■

In order to consider malicious adversary, we need to change the simulator definition and the experiment. In SIM, there is no input of $\{q_1, \ldots, q_n\}$ and, in $\mathsf{SPriv}_{\mathsf{C},\mathcal{A}}$, there is no step 1. Note, for the first PPSSI approach, it is secure against malicious adversary only if [18] is used for oblivious computation of Token.

Theorem 2: If oblivious computation of Token protocol is secure against malicious client, the hash function $H(\cdot)$, $H_1(\cdot)$, $H_2(\cdot)$, $H_3(\cdot)$ are collision resistant and Enc is a semantic secure encryption, then first PPSSI approach is secure against any probabilistic polynomial-time malicious client.

Proof: SIM construction is the same as that in the proof for theorem 1 except that, in step 2, SIM extracts all $\{q_1, \ldots, q_n\}$ from the ZKPK sent by $\mathcal{A}$, which requires rewinding of $\mathcal{A}$. Then the proof follows that for Theorem 1. ■

3) Security against Honest-but-Curious/Malicious Server: Given that the server gets no output from the protocol, the definition of client's privacy requires simply that the server cannot distinguish between cases in which the client has different inputs.

We define an experiment for any adversary $\mathcal{A}$:

The experiment $\mathsf{SPriv}_{\mathsf{S},\mathcal{A}}$:

1) The adversary $\mathcal{A}$ chooses its own database DB and outputs to the challenger two list of queries – $(q_1^0, \ldots, q_n^0)$ and $(q_1^1, \ldots, q_n^1)$.
2) The challenger chooses a random bit $b \xleftarrow{r} \{0, 1\}$ and does one of the following:
   a) If $b = 0$, then the challenger interacts with $\mathcal{A}$ as an honest client using queries $(q_1^0, \ldots, q_n^0)$.
   b) If $b = 1$, then the challenger interacts with $\mathcal{A}$ as an honest client using queries $(q_1^1, \ldots, q_n^1)$.
3) The adversary $\mathcal{A}$ outputs a bit b'.
4) The output of the experiment is defined to be 1 if $b' = b$, and 0 otherwise.

Definition 5: The first PPSSI approach is secure against honest-but-curious/malicious server if, for all probabilistic polynomial-time adversaries $\mathcal{A}$,

$$\Pr[\mathsf{SPriv}_{\mathsf{S},\mathcal{A}} = 1] \le \frac{1}{2} + \epsilon$$

Theorem 3: If oblivious computation of Token function is secure against any probabilistic polynomial-time honest-but-curious or malicious server, the first PPSSI approach is secure against any probabilistic polynomial-time honest-but-curious or malicious server.

Proof: In the first PPSSI approach, the only messages $\mathcal{A}$ gets from the client is during oblivious Token computation. If oblivious computation of Token function is secure against any probabilistic polynomial-time honest-but-curious or malicious server, the messages $\mathcal{A}$ receives from the client should be hidden by randomness. Therefore the theorem follows. ■

Remark:. Our protocols do not consider *selective failure* (see [9]), which we leave as part of future work.

VI. The Second PPSSI Approach for Very Large Databases

The first PPSSI approach in Section V, combines efficiency with provably-secure guarantees. However, in the context of *very large* databases, it faces two additional issues:

Challenge 3: Bandwidth. If server's database is very large and/or communication takes place over a slow channel, the bandwidth overhead incurred by the transfer of the encrypted database may become prohibitive.

Challenge 4: Liability. The transfer of the encrypted database to the client also prompts the problem of long-term data safety and associated liability. An encryption scheme considered strong today might gradually weaken in the long term. While we ensure that the client cannot decrypt records outside its query, it is not too far-fetched to imagine that the client might decrypt the entire database in reasonably near future, e.g., 10 or 20 years later. However, data sensitivity might not dissipate over time. For example, suppose that a low-level DoD employee is only allowed to access unclassified data. By gaining access to the encrypted database containing top secret data and patiently waiting for the encryption scheme to "age", the employee might obtain still-classified sensitive information. Further,

- Public input: e, N
- Client's private input: $\{c_i\}_{\forall i}$
- Server's private input: $d = e^{-1} \bmod \phi(N)$ where $\phi(N)$ denotes the order of $\mathbb{Z}_N^*$

1) Client: $\forall i, r_i \overset{r}{\leftarrow} \mathbb{Z}_N^*, y_i \leftarrow (r_i)^e \cdot c_i \bmod N$
2) Client $\xrightarrow{\{y_i\}_{\forall i}}$ Server
3) Server: $\forall i, z_i \leftarrow (y_i)^d \bmod N$
4) Server $\xrightarrow{\{z_i\}_{\forall i}}$ Client
5) Client: $\forall i, \texttt{Token}(c_i) \leftarrow z_i \cdot r_i^{-1} \bmod N$

Fig. 6: Oblivious computation of $\texttt{Token}(\cdot)$ using DT10-2.

- Public input: p, q
- Client's private input: $\{c_i\}_{\forall i}$
- Server's private input: $k \in \mathbb{Z}_q^*$

1) Client: $\forall i, \alpha_i \overset{r}{\leftarrow} \mathbb{Z}_q^*$,
   $y_i \leftarrow ((c_i)^{(p-1)/q})^{\alpha_i} \bmod p$
2) Client $\xrightarrow{\{y_i\}_{\forall i}}$ Server
3) Server: $\forall i, z_i \leftarrow y_i^k \bmod \mathbb{Z}_p^*$
   $\pi \leftarrow ZKPK\{k | \{z_i = y_i^k\}_{\forall i}\}$
4) Server $\xrightarrow{\{z_i\}_{\forall i}, \pi}$ Client
5) Client: Aborts if π doesn't verify.
   $\forall i, \texttt{Token}(c_i) \leftarrow z_i^{1/\alpha_i} \bmod p$

Fig. 7: Oblivious computation of $\texttt{Token}(\cdot)$ using JL10.

- Public input: $P, Q = P^s$
- Client's private input: $\{c_i\}_{\forall i}$
- CA's private input: s.
- Server's private input: z

1) CA: $\forall i,\ \sigma_i \leftarrow (c_i)^s$
2) CA $\xrightarrow{\{\sigma_i\}_{\forall i}}$ Client
3) Server: $R \leftarrow P^z$ (Offline)
4) Server $\xrightarrow{R}$ Client
5) Client: $\forall i, \texttt{Token}(c_i) \leftarrow \hat{e}(R, \sigma_i)$

Fig. 8: Oblivious computation of $\texttt{Token}(\cdot)$ using IBE-APSI.

in several settings, parties (e.g., banks) may be prevented, by regulation, from releasing copies of their databases (even if encrypted).

In the rest of this section, we introduce a novel architecture to address the challenges for very large databases. Our new approach incurs very limited overhead (in terms of both computation and communication), even when compared to non-privacy preserving querying systems.

A. Introducing the "Isolated Box"

In order to address Challenge 3 and 4, we propose a system architecture shown in Figure 4. It includes a new component: *"Isolated Box"* (IB), a non-colluding, untrusted party connected with both the server and the client.

The new interaction involving IB is shown in Figure 5. During the (offline) setup phase, the server encrypts its database, using `EncryptDatabase` (Algorithm 1), and transfers the encrypted database to the IB. Server's computation of `Token` functionality no longer depends on client's input, thus, the server can evaluate $\texttt{Token}(\cdot)$ without involving the client.

To pose a query, the client first engages with the server in oblivious computation of `Token` (online step 1). Next, for each computed token, it runs the `IBLookup` procedure (Algorithm 3) to retrieve matching records from the IB.

The $\texttt{Token}(\cdot)$ functionality is now instantiated using (A)PSI-DT *with* pre-distribution. Specifically, we select the construction from [20] (denoted as **DT10-2**), [32] (denoted as **JL10**) and [17] (denoted as **IBE-APSI**). Again, our choices are based on these protocols' efficiency and security models. Our experiments – in Appendix of [19] – show that DT10-2, secure in the presence of HbC adversaries, is the most efficient construction, while JL10 combines reasonable efficiency with security against malicious adversary. IBE-APSI is the only APSI-DT with pre-distribution, and it is secure against HbC adversaries. For the sake of completeness, we define `Token` function for the selected (A)PSI-DT constructions in Table III. Note that d, k, z are server's secret parameters. Complete details, for each instantiation of oblivious computation, are presented in Figure 6, 7 and 8.

Trust Assumptions. The Isolated Box is assumed not to collude with either the server or the client. (Although, we discuss the consequences of collusion in Section VI-F.) We remark that the use of non-colluding parties in the context of Secure Computation [46] was first suggested by [24], and then applied in [4], [8], [21], [34], [33], [2].

While our requirement for the presence of IB might seem like a "strong" assumption, we stress that the IB is only trusted not to collude with other parties. It simply stores server's encrypted database and returns ciphertexts matching client's encrypted queries (i.e., *tags*), without learning any information about records and queries. Also note that, in practice, the IB can be either instantiated as a (non-colluding) cloud server or as a piece of secure hardware installed on server's premises: it is only important to ensure that the server does not learn *what* the IB reads from its storage and transfers to the client.

B. Database Encryption

IB's presence does not really affect database encryption, i.e., `Encryptdatabase` procedure presented in Algorithm 1. It only uses a different $\texttt{Token}(\cdot)$ function. While in the first approach (Section V) we rely on (A)PSI-DT *without* pre-distribution (i.e., the server cannot run $\texttt{Token}(\cdot)$ before interacting with the client), we now use (A)PSI-DT *with* pre-distribution. Thus, the server can evaluate $\texttt{Token}(\cdot)$ over its own inputs, *offline*, and then transfer the encrypted database to the IB.

C. Query lookup

`IBLookup` procedure is used by the client to obtain records matching client's query. It is shown in Algorithm 3.

Similar to our first approach, the client runs the lookup procedure after obtaining search tokens (via oblivious computation of `Token` – online step 1 in Figure 5). For each derived token, tk_i, it invokes `IBLookup` to retrieve (from the IB) all records matching tk_i.

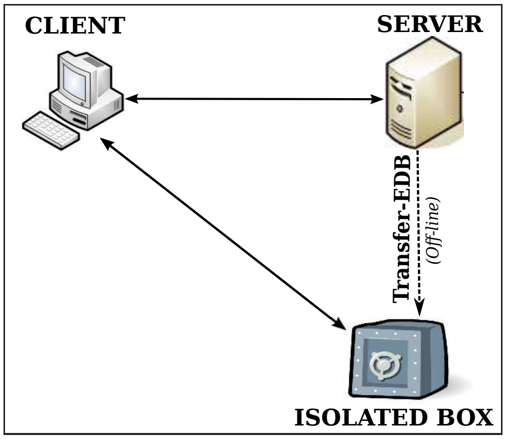


Fig. 4: The introduction of the Isolated Box.

- Client's input: $\{c_i, \sigma_i\}_{1 \le i \le v}$, where: $c_i = H(attr_i^*, val_i^*)$
- Server's input: $\{s_{j,l}\}_{1 \le j \le w, 1 \le l \le m}, \{R_j\}_{1 \le j \le w}$,
  where: $s_{j,l} = H(attr_l, val_{j,l})$

Offline:

```
1) Server: $\mathbf{EDB} \leftarrow \texttt{EncryptDatabase}(\texttt{Token}(\cdot), \{R_j\}_{1 \le j \le w})$
2) Server $\xrightarrow{\mathbf{EDB}}$ IB
```

Online:

```
1) Client $\xLeftrightarrow{\text{Obliviously computes}\{tk_i \leftarrow \texttt{Token}(c_i)\}_{\forall i}}$ Server
2) Client $\xLeftrightarrow{\{\mathbf{R}_i \leftarrow \texttt{IBLookup}(tk_i)\}_{\forall i}}$ IB
3) Client: $\forall R \in \cup_{\forall i} \mathbf{R}_i$, output $R$
```

Fig. 5: Outline of our second PPSSI approach based on IB.

Scheme name	`Token` definition	PSI category
DT10-2 (Figure 4 of [20])	$\texttt{Token}(c) = (c)^d \bmod N$	PSI-DT with pre-distribution
JL10 (Figure 2 of [32])	$\texttt{Token}(c) = ((c)^{(p-1)/q})^k \bmod p$	PSI-DT with pre-distribution
IBE-APSI (Figure 5 of [17])	$\texttt{Token}(c) = \hat{e}(Q, c)^z$	APSI-DT with pre-distribution

TABLE III: `Token` for (A)PSI-DT with pre-distribution ($c_i = H(attr_i^*, val_i^*)$ and $c \in \{c_i\}_{1 \le i \le v}$)

```
Algorithm 3: IBLookup Procedure
Client's input  : $tk_i$
IB's input      : $\mathbf{EDB} = \{\mathbf{LTable}, \{er_j\}_{1 \le j \le w}\}$
Client's output: Matching record set $\mathbf{R}$
  1) Client: $ctr \leftarrow 1$
  2) Client: $tag_i \leftarrow H_1(tk_i||ctr)$, $k_i'' \leftarrow H_3(tk_i|||ctr)\}$
  3) Client $\xrightarrow{tag_i, k_i''}$ IB
  4) IB: If $(\exists tag_{j,l} \in \mathbf{LTable}_{j,l}$ s.t. $tag_{j,l} = tag_i)$
            $j' \leftarrow Dec_{k_i''}(eind_{j,l})$,
            $ret \leftarrow \{ek_{j,l}, er_{j'}\}$
         else
            $ret \leftarrow \perp$
  5) IB $\xrightarrow{ret}$ Client
  6) Client: If $ret = \perp$, abort
         else $k_i' = H_2(tk_i||ctr)$, $k_i = Dec_{k_i'}(ek_{j,l})$
            $R_i = Dec_{k_i}(er_{j'})$, $\mathbf{R} \leftarrow \mathbf{R} \cup R_i$
            $ctr \leftarrow ctr + 1$, Goto step 2.
```

We use the term *transaction* to denote a complete query procedure, for each tk_i (from the time the first query for tk_i is issued, until the last response from the IB is received). *Retrieval* denotes the receipt of a single response record during a transaction. A transaction is composed of several retrievals between the client and the IB. The client retrieves records one by one from the IB, by gradually incrementing the counter ctr. In step 1, the client sets ctr to 1. In step 2, the client derives tag_i and an index decryption key k_i'' from token tk_i. After receiving tag_i and k_i'' in step 3, the IB searches for matching tags in the lookup table in step 4. If there is a match, the IB recovers the index j' by decrypting $eind_{j,l}$ with k_i'', assembles the corresponding record $er_{j'}$ and the ciphertext of its decryption key $ek_{j,l}$ into ret and transmits ret to the client in step 5. Otherwise, $\perp$ is transmitted. If the client receives $\perp$, it aborts. Otherwise, it decrypts $ek_{j,l}$ into k_i with k_i' and recovers record R_i from $er_{j'}$ using k_i. Then, it increments ctr and starts another retrieval by returning to step 2.

We can use hash table to store $\mathbf{LTable}$ for efficiency. If $\mathbf{LTable}$ is too big to be stored in hash table, we can turn to B-tree. Creating B-tree can be done offline at the server.

D. Optimizations

Since transmission of ret may incur some delay, Algorithm 3 can be sped up by pipe-lining computation of tag_i and k_i'' (step 2) in next retrieval with the transmission of ret (step 5) in current retrieval.

Note that the computation of $ek_{j,l}$ and $eind_{j,l}$ (steps 13–14 in Algorithm 1) can also be optimized. Since we use a counter as input to compute $k_{j,l}'$ (respectively, $k_{j,l}''$), each $k_{j,l}'$ (respectively, $k_{j,l}''$) is different for any j, l. Both $k_{j,l}'$ and $k_{j,l}''$ are 160-bit values (SHA-1), while k_j is 128 bits and j is clearly smaller. Hence, we can use *one-time-pad* encryption (i.e. $ek_{j,l} = k_{j,l}' \oplus k_j$ and $eind_{j,l} = k_{j,l}'' \oplus j$) to speed up computation. In Algorithm 3, $Dec_{k_i''}(eind_{j,l})$ becomes $k_i'' \oplus eind_{j,l}$ and $Dec_{k_i'}(ek_{j,l})$ changes to $k_i' \oplus ek_{j,l}$.

E. Challenges Revisited

Since we use the same encryption procedure discussed in Section V, Challenge 1 and 2 are already addressed. Thus, we only consider Challenge 3 and 4.

Bandwidth: Once the server transfers its database (offline) to the IB, the latter returns to the client only records matching its query. Therefore, bandwidth consumption is minimized.

Liability: Since the IB holds the encrypted database, the client only obtains the result of its queries, thus, ruling out any potential liability issues.

Finally, the introduction of the IB enables Server Unlinkability and Forward Security, despite the fact that we use (A)PSI-DT *with* pre-distribution techniques. Indeed, records not matching a query are never available to the client, thus, it does not learn whether they have changed. Similarly, the client cannot use future authorizations to maliciously obtain information from previous (recorded) interactions.

F. Discussion

Privacy Revisited. The introduction of the IB and the use of counter mode in database encryption provide additional privacy properties. If the client performs only one query transaction, as in Algorithm 3, the IB can link all tag values in step 3 to the same $(attr, val)$ pair. This may pose a similar risk to that discussed in the "multi-set" challenge, with respect to the IB. However, the counter allows the client to retrieve matching records one by one. Therefore, the client can choose to add a random delay between two subsequent retrievals in a single transaction. If the distribution of additional delay is indistinguishable from time gaps between two transactions, the IB cannot tell the difference between two continuous retrievals within one transaction from two distinct transactions. As a result, the IB cannot infer whether two continuously retrieved records share the same $(attr, val)$ pair and the distribution of the attribute value remains hidden.

Also note that the introduction of the IB does not violate Client or Server Privacy. Client Privacy is preserved because the client (obliviously) computes a token, which is not learned by the server. The IB does not learn client's interests, since client's input to the IB (tag) is statistically indistinguishable from a random value, in the random oracle model. Server Privacy is preserved because the client does not gain any extra information by interacting with the IB. Finally, the IB only holds the encrypted database and learns no plaintext.

Removing Online Server. Although it only needs to perform oblivious computation of tokens, we still require the server to be online. Inspired by [30] and [25], we can replace the online server with a tamper-proof smartcard, dedicated to computing `Token` function. The server only needs to program its secret key into the smartcard, which protects the key from being accessed by the client. This way, after handing the smartcard to the client, the server can go offline. The smartcard is assumed to enforce a limit on the number of `Token` invocations.

Limitations. We acknowledge that our second PPSSI approach has some limitations. Over time, as it serves many queries, the IB gradually learns the relationship between tags and encrypted records through pointers associated with each tag. This issue can be mitigated by letting the server periodically re-encrypt the database. IB also learns database access patterns generated by query executions. Nonetheless, without knowing the distribution of query predicates, the access pattern of encrypted data leaks very little information to the IB. Next, if the server and the IB collude, Client Privacy is lost, since the IB learns tag that the client seeks, and the server knows the $(attr, val)$ pair each tag is related to. On the other hand, if the client and the IB collude, the client can access the entire encrypted database, thus, liability (long-term data safety) may be endangered. Last, Server Unlinkability is protected only with respect to the client. Server Unlinkability with respect to the IB is not guaranteed, since the IB learns about all changes in server's database. Finally, note that PPSSI (with all approaches) currently supports only equality and disjunctive queries. Enabling conjunctive queries would require treating all combinations of $(attr, val)$ pairs as server's set elements. Thus, client's input would become exponential in terms of the number of attributes. This remains an interesting challenge left as part of future work.

Dynamic Databases.. So far, the database is assumed to be static. We emphasize that our second approach can also deal with dynamic database with some small modifications. To be specific, we require the server, for each different pair of $(attr, val)$, store the max counter, $ctrmax_{attr,val} \leftarrow max_{attr_l=attr,val_{j,l}=val}(ctr_{j,l})$. This data is proportional to the number of different $(attr, val)$ pairs and can be huge. Instead of saving it on its own, the server can outsource this data as a new encrypted database to the IB. When the server needs to add[4] a new record, it fetches the max counter for each $(attr, val)$ in the new record and then encrypt it according to Algorithm 1(line 8-15) after incrementing corresponding max counters by one. Then the server pushes the new encrypted record and updated $ctrmax_{attr,val}$ to the IB. To delete a record, the server first needs to query and locate the record. Then it updates its content with a special mark 'D', re-encrypts the record and pushes it to the IB. When client retrieves a deleted record, it does not process the record but continues to retrieve the next one.

G. Security Analysis of Second PPSSI Approach

Since we do not consider collusion, the security against Honest-but-Curious/Malicious client and server follows exactly from Theorem 1, 2, 3. So we only discuss security against Honest-but-Curious/Malicious Isolated Box.

Like Section V-E, we use q_i to denote the ith query of the form $(attr, val)$ issued by the client and use Q_i to denote all records matching query q_i.

1) Security against Honest-but-Curious/Malicious Isolated Box (IB): We define security against Honest-but-Curious/Malicious Isolated Box (IB) by comparing its view when interacting with an honest client and an honest server with its view when interacting with a simulator SIM.

Simulator SIM:
SIM is given $|X_U|$ for all $U \subseteq \{0, \ldots, n\}$ where $X_U = \cap_{i \in U} Q_i$.

1) SIM outputs an encrypted database $\mathbf{EDB'}$ to $\mathcal{A}$.
2) SIM interacts with $\mathcal{A}$ as a client, simulating queries $\{q_1, \ldots, q_n\}$ (even though SIM does not know $\{q_1, \ldots, q_n\}$).

[4] We only consider 'append' operation since it would not present different answers than 'insert' operation to our supported database queries.

Note, in the above definition, the only information SIM knows is the cardinality of X_U which is defined as the intersection of a subset of query answers.

We then define an experiment for any adversary $\mathcal{A}$:

The experiment $\mathsf{SPriv}_{\mathsf{IB},\mathcal{A}}$:

1) The adversary $\mathcal{A}$ outputs to the challenger a database DB and a list of queries $\{q_1, \ldots, q_n\}$.
2) The challenger chooses a random bit $b \overset{r}{\leftarrow} \{0, 1\}$ and does one of the following:
   a) If $b = 0$, then the challenger interacts with $\mathcal{A}$ as an honest client and an honest server.
   b) If $b = 1$, then the challenger computes $\{Q_1, \ldots, Q_n\}$ based on DB, derives all intersections X_U for all $U \subseteq \{1, \ldots, n\}$ and interacts with $\mathcal{A}$ as $\mathsf{SIM}(\{|X_U|\}_{\forall U \subseteq \{1,\ldots,n\}})$.
3) The adversary $\mathcal{A}$ outputs a bit b'.
4) The output of the experiment is defined to be 1 if $b' = b$, and 0 otherwise.

Definition 6: The second PPSSI approach is secure against honest-but-curious/malicious IB if, for all probabilistic polynomial-time adversaries $\mathcal{A}$, there exists a probabilistic polynomial-time simulator SIM such that

$$\Pr[\mathsf{SPriv}_{\mathsf{IB},\mathcal{A}} = 1] \le \frac{1}{2} + \epsilon$$

Theorem 4: If the hash function $H(\cdot)$, $H_1(\cdot)$, $H_2(\cdot)$, $H_3(\cdot)$ are (t_0, ϵ_0), (t_1, ϵ_1), (t_2, ϵ_2), (t_3, ϵ_3) collision resistant, Enc is a $(t_{enc}, \epsilon_{enc})$ semantic secure encryption, and Token is a (t_T, ϵ_T) unpredictable function, then the second PPSSI approach is (t, ϵ)-secure against any probabilistic polynomial-time honest-but-curious/malicious IB where $t \le min(t_0, t_1, t_2, t_3, t_{enc}, t_T) - w \cdot m \cdot t_{\text{Token}} - w \cdot t_{enc}$ and $\epsilon = \epsilon_0 + \epsilon_1 + \epsilon_2 + \epsilon_3 + 2\epsilon_T + 2\epsilon_{enc}$.

Proof: Our goal is to construct a simulator SIM such that $\mathcal{A}$ cannot tell the difference between the view when interacting with SIM and the view when interacting with an honest client and an honest server. Our SIM is constructed as follows:

1) SIM creates $\mathbf{EDB}'$:
   - Pick w random messages of same length as encrypted messages.
   - Then create $\mathbf{LTable}' = \{(tag'_{j,l}, ek'_{j,l}, eind'_{j,l})\}_{1 \le j \le w, 1 \le l \le m}$ where $tag'_{j,l} \in_R \{0,1\}^{l_h}$, $ek'_{j,l} \in_R \{0,1\}^{l_e}$, $eind'_{j,l} \in_R \{0,1\}^{l_e}$, l_h is the output length of hash function, l_e is the output length of encryption function.
2) For each query q_i, SIM prepares the matching tag set $T_i = \{tag^i_1 \ldots tag^i_{|Q_i|}\}$ such that $| \cap_{i \in U} T_i| = |X_U|$ $\forall U \subseteq \{0, \ldots, n\}$ as follows:
   - $\forall U \subseteq \{0, \ldots, n\}$, compute $|\hat{X}_U|$ where $\hat{X}_U = X_U \backslash \cup_{|U'| > |U|} X_{U'}$, i.e., the fraction of the intersection determined by U without being covered in any U' such that $|U'| > |U|$. Given $|X_U|$, $|\hat{X}_U|$ can be computed as
     $$|\hat{X}_U| = |X_U| - |X'_U|$$
     where $|X'_U| = |X_U \cap (\cup_{|U'|>|U|} X_{U'})| = \sum_{|U'|>|U|} |X_U \cap X_{U'}| - \sum_{|U'_1|>|U|, |U'_2|>|U|, U'_1 \ne U'_2} |(X_U \cap X_{U'_1}) \cap (X_U \cap X_{U'_2})| + \cdots + (-1)^{\binom{n}{n} + \cdots + \binom{n}{|U|+1}} \cdot | \cap_{|U'|>|U|} (X_U \cap X_{U'})|)$. The above formula is attributed to the inclusion-exclusion principle [3]. Note that $|X_{U_1} \cap \cdots \cap X_{U_i}| = |X_{U_1 \cup \cdots \cup U_i}|$ and therefore we can compute $|\hat{X}_U|$ for any U.
   - Randomly pick $\sum_{\forall U} |\hat{X}_U|$ different tags from $\mathbf{LTable}'$ and store them in Y. Note that $\sum_{\forall U} |\hat{X}_U| = | \cup_{j=1}^n Q_j|$. For each U, initialize $\hat{Q}_U$ as follows:
     a) Pick $|\hat{X}_U|$ distinct tags from Y and add them to $\hat{Q}_U$.
     b) Update $Y \leftarrow Y \backslash \hat{Q}_U$.
   - For $i = 1, \ldots, n$, set $T_i = \cup_{i \in U} \hat{Q}_U$. Note that $| \cap_{i \in U} T_i| = |X_U|$ due to the above construction of $\hat{Q}_U$.
3) SIM plays the role of a client as follows: for the λth query, make $|T_\lambda|$ probes where θth probe is the θth element in T_λ.

We first analyze the view of $\mathcal{A}$ between tags in $\mathbf{EDB}$ and those in $\mathbf{EDB}'$. The distribution of tags in $\mathbf{EDB}$ and those in $\mathbf{EDB}'$ is the same unless one of the following happens: (1) there exists $(attr_i, val_i) \ne (attr_j, val_j)$ but $H(attr_i, val_i) = H(attr_j, val_j)$; (2) $H(attr_i, val_i) \ne H(attr_j, val_j)$ but $H_1(\texttt{Token}(H(attr', val'))||ctr') = H_1(\text{Token}(H(attr'', val''))||ctr'')$; (3) $\mathcal{A}$ forges $\text{Token}(H(attr, val))$ for certain $(attr, val)$. This means finding a collision in $H(\cdot)$ or $H_1(\cdot)$, or breaking Token, which happens with probability at most $\epsilon' = \epsilon_0 + \epsilon_1 + \epsilon_T$ if $t + w \cdot m \cdot t_{\text{Token}} + t_{enc}$ is bounded by $min(t_0, t_1, t_T)$.

Next we analyze $\mathcal{A}$'s view between $(\{ek_{j,l}, eind_{j,l}\}_{1 \le l \le m}, er_j)_{1 \le j \le w}$ in $\mathbf{EDB}$ and those in $\mathbf{EDB}'$. The only time that $\mathcal{A}$ observes difference is when (1) $\mathcal{A}$ breaks semantic secure encryption algorithm; (2) $\mathcal{A}$ finds collision in $H_2(\cdot)$ or $H_3(\cdot)$ (which breaks one-time-pad encryption); (3) $\mathcal{A}$ forges $\text{Token}(H(attr, val))$ for certain $(attr, val)$. All these happen with probability at most $\epsilon'' = \epsilon_{enc} + \epsilon_2 + \epsilon_3 + \epsilon_T$ if $t + w \cdot m \cdot t_{\text{Token}} + w \cdot t_{enc}$ is bounded by $min(t_{enc}, t_2, t_3, t_T)$.

Last we show that $\mathcal{A}$ cannot distinguish the way that an honest client's queries are answered using $\mathbf{EDB}$ and the way that SIM's queries are answered using $\mathbf{EDB}'$. For an honest client's query q_i, there are $|Q_i|$ matches in $\mathbf{EDB}$. For the SIM's ith query, it makes $|T_i|$ probes (excluding the last failed probe) and there will be $|T_i|$ matches. Since $|T_i| = |Q_i|$ and $\forall U$, $| \cap_{i \in U} T_i| = |X_U| = | \cap_{i \in U} Q_i|$, $\mathcal{A}$ cannot distinguish er_j from er'_j unless $\mathcal{A}$ breaks semantic secure encryption which happens with probability ϵ_{enc} if $t + w \cdot m \cdot t_{\text{Token}} + w \cdot t_{enc}$ is bounded by t_{enc}.

Putting it all together, the second PPSSI approach is (t, ϵ)-secure if $t + w \cdot m \cdot t_{\text{Token}} + w \cdot t_{enc} \le min(t_0, t_1, t_2, t_3, t_{enc}, t_T)$ and $\epsilon = 2\epsilon_{enc} + \epsilon_0 + \epsilon_1 + \epsilon_2 + \epsilon_3 + 2\epsilon_T$ ■

VII. Performance Evaluation

In this section, we evaluate the performance of our PPSSI approaches. First, we benchmark step-by-step cost of proposed techniques. Next, we compare our first PPSSI approach to PIR. Finally, we build a (limited) database management system to compare our second PPSSI approach to a non privacy-preserving MySQL database.

A. Benchmarking All PPSSI Components

The following benchmark refers to executions on an Intel Harpertown server with Xeon E5420 CPU (2.5 GHz, 12MB L2 Cache) and 8GB RAM inside. We build the benchmarking tool based on OpenSSL library (ver.1.0.0c) [47] and PBC library (ver.0.5.11) [36].

1) PPSSI Operations: We now evaluate the performance of all operations involved in both of our PPSSI approaches. Remark that we use 2048-bit modulus and records of fixed $2KB$ length.

Figure 9 measures the time needed to perform the oblivious computation of `Token` function, for every possible (A)PSI-DT instantiation. Observe that the cost always increases linearly with client's query size. As for protocols without pre-distribution, DT10-APSI is unsurprisingly more expensive than DT10-1. Whereas, DT10-2 and JL10 are, respectively, the most and the least efficient ones of protocols with pre-distribution.

Then, Figure 10 evaluates the performance of the Look-up-Table encryption, performed by the server. This operation includes server's computation of `Token` function over its own input (Note that this is not oblivious computation). Again, running time always increases linearly with the product of the number of records (w) and the number of attributes (m).

In Figure 11, we report the cost of the Record-level encryption. This only depends on the number of records. Compared to the Lookup-table encryption, the Record-level encryption incurs a negligible overhead.

Finally, Figure 12 presents the running time of the Lookup procedures (Algorithm 2 and Algorithm 3 without consideration of communication delay). Unsurprisingly, cost is identical for both algorithms and increases linearly with the number of matching records (v_m). This is because we use a hash table to store all server computed tags in **LTable** and matching one client tag takes only constant time.

We conclude that, as all operations have linear complexity, our approaches scale efficiently for larger databases and query sets. All above experiments are done in small scale because it is easy to pinpoint exact numbers in such scale. One can easily infer results for super large parameters and hence we omit them here.

B. First PPSSI Approach vs PIR

We now aim at comparing the efficiency of proposed first PPSSI approach (Section V) to that of related work – SPIR. Recall that first PPSSI approach provides very similar privacy properties of SPIR. Indeed, both PPSSI and SPIR hide client's access patterns to the server and also protect privacy of server's data (with respect to records not matching the queries). However, one possible criticism against our side is that the communication overhead is *linear* in the size of the database size, whereas, SPIR incurs *sub-linear* communication overhead. Remark, however, that: (1) SPIR does not support keyword search, and (2) SPIR introduces a remarkably higher computation overhead, which ends up "overshadowing" the advantage in the communication complexity. To support the latter claim, we compare the overall performance of our first PPSSI approach with that of Gentry and Ramzan's single-database PIR (GR-PIR) [27], which is, to the best of our knowledge, the most efficient single-database PIR. Specifically, GR-PIR [27], assuming a database with n records, incurs $O(k+d)$ communication complexity (where $k \leq \log n$ and d is the bit-length of each record), and $O(n)$ computation overhead. Also recall that, according to [39], any single-database PIR can be extended to SPIR/OT and we are not aware of any SPIR/OT that is more efficient than GR-PIR.

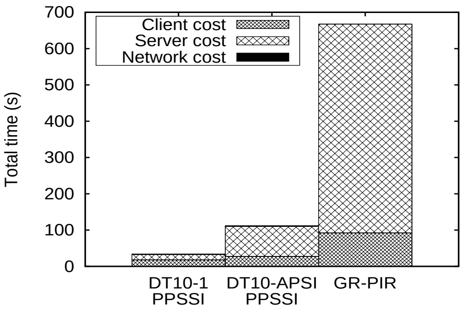


Fig. 13: Performance comparison between the first PPSSI approach (Section V) and GR-PIR [27].

In our comparison, we use a database with $w = 1024$ records and $m = 5$ attributes. Each record has size $2KB$. We assume the client's query size is $v = 1024$ and there will be 10 (1%) records matching the query (v_m). On a conservative stance, we choose a relatively slow connection between the client and the server, i.e., a $10Mbps$ link. Remark that we choose 2048-bit modulus and use RC4 and SHA1 as symmetric encryption and hash function, respectively.

The result of our comparison is showed in Figure 13 and confirms that our approach is significantly more efficient than GR-PIR. We break down the results into client, server and network transmission cost. Note that, for all schemes, network cost (at the top stack in each bar) is negligible compared to client and server cost. Also observe that GR-PIR imposes a significant overhead on both client and server. We do not show results for larger databases, since: (1) both server and client computational costs will always increase linearly for all schemes, and (2) for very large database, we prefer the approach with the Isolated Box (whose overall performance is evaluated next).

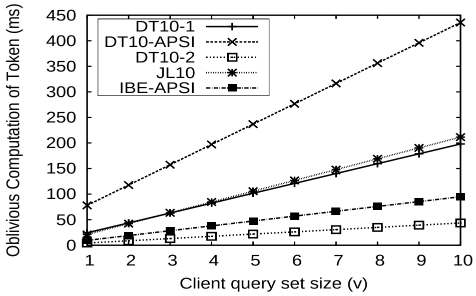


Fig. 9: `Token` Oblivious Computation.

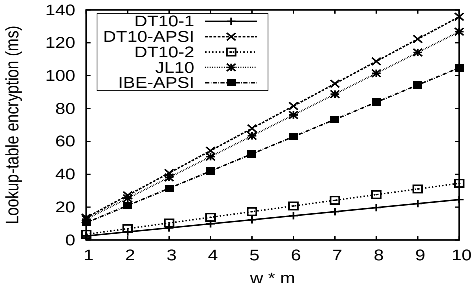


Fig. 10: Lookup-Table Encryption (line 8-15 of Algorithm 1).

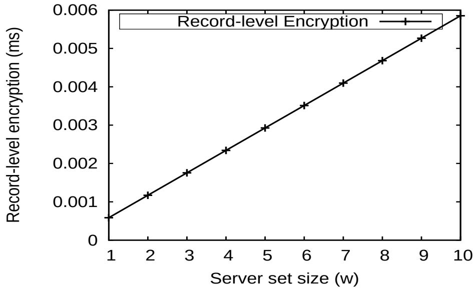


Fig. 11: Record-level Encryption (line 1-6 of Algorithm 1).

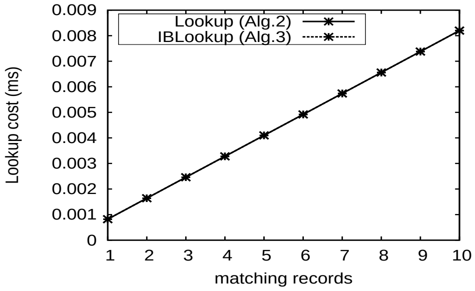


Fig. 12: Lookup (Alg. 2) and IBLookup (Alg. 3).

C. Second PPSSI approach vs MySQL

To the best of our knowledge, there is no available approach to PPSSI that combines efficiency with provably secure guarantees and that relies on a non-colluding, untrusted party, such as the Isolated Box. Therefore, we cannot compare our second PPSSI approach for very large databases (Section VI) to any prior work. Nonetheless, we evaluate its performance by measuring it against standard (non privacy-preserving) MySQL.

On a conservative stance, we use MySQL with indexing enabled on each searchable attribute. We run the IB and the server on the same machine. Client is connected to the server and the IB through a $100Mbps$ link. The testing database has 45 searchable attributes and 1 unsearchable attribute (type "LARGEBLOB") used to pad each record to a uniform size. There are, in total, $100,000$ records. All records have the same size, which we vary during experiments. The IB is preloaded with **LTable** into memory. All results are averaged over 10 runs.

First, we compare the *index lookup time*, defined as the time between SQL query issuance and the receipt of the first response from the IB. We select a set of SQL queries that return 0, 1, 10, 100, 1000, 10000 ($\pm 10\%$) responses, respectively, and fix each record size at 500KB. Figure 14(a) shows index lookup time for our PPSSI approach (with respect to all underlying (A)PSI-DT instantiations), as well as MySQL, with respect to the response set size. All proposed schemes' cost are slightly more expensive than MySQL and are independent of the response size.

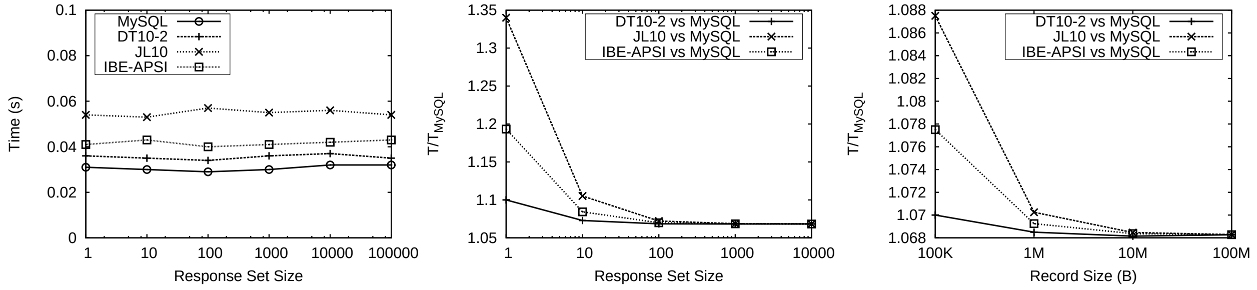
Next, we test the impact of the response set size on the *total query time*, which we define as the time between SQL query issuance and the arrival of the last response from the IB. Figure 14(b) shows the time for the client to complete a query for a specific response set size divided by the time taken by MySQL (again, with respect to all underlying (A)PSI-DT instantiations). Results gradually converge to 1.1 for increasing response set sizes, i.e., our approach is only 10% slower than standard MySQL. This is because the extra delay incurred by cryptographic operations (in the oblivious evaluation of `Token`) is amortized by subsequent data lookups and decryptions. Note that we can also infer the impact of various client query set size by multiplying the client query set size with each single query delay.

Last, we test the impact of record size on the total query time. We fix response set size at 100 and vary each record size between $100KB$ and $100MB$. Figure 14(c) shows the ratio between our PPSSI approach and MySQL, once more with respect to all underlying (A)PSI-DT instantiations. Again, results gradually converge well below 1.1 with increasing record size. This occurs because, with bigger records, the overhead of record decryption becomes the "bottleneck".

VIII. Conclusion

In this paper, we proposed secure and efficient techniques for Privacy-Preserving Sharing of Sensitive Information (PPSSI), which enable a client and a server to exchange information without leaking more than the required minimum of information. Privacy guarantees are formally defined and achieved with provable security.

We implemented two variants of PPSSI: one is geared for small/medium-size data sets, while the other minimizes communication overhead, as well as liability issues, for



(a) Index lookup speed comparison. (b) Comparison to MySQL w.r.t. response size. (c) Comparison to MySQL w.r.t. record size.

[DT10-2, JL10, IBE-APSI labels indicate the instantiation used for the Token function in PPSSI]

Fig. 14: Performance comparison between the second PPSSI approach (Section VI) and MySQL.

very large databases. The latter introduces a non-colluding, untrusted party – the Isolated Box – which can be implemented as a piece of secure hardware.

Finally, we presented extensive experimental results, which confirmed that our PPSSI approaches are efficient enough to be used in real-world applications. Our future work includes supporting versatile query predicates (e.g., conjunctive queries) as well as fuzzy queries over non-normalized data.

IX. Acknowledgment

This research was supported by the US Intelligence Advanced Research Projects Activity (IARPA) under grant number FA8750-09-2-0071. We also would like to thank Xiaomin Liu and Stanislaw Jarecki for their helpful comments.

References

[1] Agrawal, R., Evfimievski, A., Srikant, R.: Information sharing across private databases. In: SIGMOD (2003)

[2] Asonov, D., Freytag, J.C.: Almost optimal private information retrieval. In: Privacy Enhancing Technologies (2003)

[3] Balakrishnan, V.K.: Theory and Problems of Combinatorics. McGraw-Hill (1995)

[4] Beaver, D.: Commodity-based cryptography. In: STOC (1997)

[5] Bertino, E., Byun, J., Li, N.: Privacy-preserving database systems. Foundations of Security Analysis and Design (2005)

[6] Boneh, D., Di Crescenzo, G., Ostrovsky, R., Persiano, G.: Public key encryption with keyword search. In: Eurocrypt (2004)

[7] Bursztein, E., Lagarenne, J., Hamburg, M., Boneh, D.: OpenConflict: Preventing Real Time Map Hacks in Online Games. In: S&P (2011)

[8] Cachin, C.: Efficient private bidding and auctions with an oblivious third party. In: CCS (1999)

[9] Camenisch, J., Neven, G., Shelat, A.: Simulatable adaptive oblivious transfer. In: Eurocrypt'07, pp. 573–590 (2007)

[10] Caslon Analytics: Consumer Data Losses. http://www.caslon.com.au/datalossnote.htm

[11] Chor, B., Gilboa, N., Naor, M.: Private information retrieval by keywords. Manuscript (1998)

[12] Chor, B., Kushilevitz, E., Goldreich, O., Sudan, M.: Private information retrieval. Journal of the ACM **45**(6), 965–981 (1998)

[13] Chow, S., Lee, J., Subramanian, L.: Two-party computation model for privacy-preserving queries over distributed databases. In: NDSS (2009)

[14] Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: CCS (2006)

[15] Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: Improved definitions and efficient constructions. Journal of Computer Security **19**(5) (2011)

[16] De Cristofaro, E., Durussel, A., Aad, I.: Reclaiming Privacy for Smartphone Applications. In: PerCom (2011)

[17] De Cristofaro, E., Jarecki, S., Kim, J., Tsudik, G.: Privacy-preserving policy-based information transfer. In: PETS (2009)

[18] De Cristofaro, E., Kim, J., Tsudik, G.: Linear-Complexity Private Set Intersection Protocols Secure in Malicious Model. In: Asiacrypt (2010)

[19] De Cristofaro, E., Lu, Y., Tsudik, G.: Efficient techniques for privacy-preserving sharing of sensitive information. Cryptology ePrint Archive, Report 2011/113 (2011). http://eprint.iacr.org/

[20] De Cristofaro, E., Tsudik, G.: Practical private set intersection protocols with linear complexity. In: FC (2010)

[21] Du, W., Zhan, Z.: A practical approach to solve secure multi-party computation problems. In: NSPW (2002)

[22] Evfimievski, A., Gehrke, J., Srikant, R.: Limiting privacy breaches in privacy preserving data mining. In: PODS

[23] Federal Bureau of Investigation: Terrorist Screening Center.
http://www.fbi.gov/terrorinfo/counterrorism/tsc.htm

[24] Feige, U., Killian, J., Naor, M.: A minimal model for secure computation (extended abstract). In: STOC (1994)

[25] Fischlin, M., Pinkas, B., Sadeghi, A.R., Schneider, T., Visconti, I.: Secure set intersection with untrusted hardware tokens. In: CT-RSA (2011)

[26] Freedman, M., Nissim, K., Pinkas, B.: Efficient private matching and set intersection. In: Eurocrypt (2004)

[27] Gentry, C., Ramzan, Z.: Single-database private information retrieval with constant communication rate. In: ICALP (2005)

[28] Gertner, Y., Ishai, Y., Kushilevitz, E., Malkin, T.: Protecting data privacy in private information retrieval schemes. In: STOC (1998)

[29] Hacigümüş, H., Iyer, B., Li, C., Mehrotra, S.: Executing SQL over encrypted data in the database-service-provider model. In: SIGMOD (2002)

[30] Hazay, C., Lindell, Y.: Constructions of truly practical secure protocols using standardsmartcards. In: CCS (2008)

[31] Hore, B., Mehrotra, S., Tsudik, G.: A privacy-preserving index for range queries. In: VLDB (2004)

[32] Jarecki, S., Liu, X.: Fast secure computation of set intersection. In: SCN (2010)

[33] Kantarcioĝlu, M., Clifton, C.: Assuring privacy when big brother is watching. In: DMKD (2003)

[34] Kantarcioglu, M., Vaidya, J.: An architecture for privacy-preserving mining of client information. In: CRPIT (2002)

[35] Kissner, L., Song, D.: Privacy-preserving set operations. In: CRYPTO (2005)

[36] Lynn, B.: PBC: The Pairing-Based Cryptography Library. http://crypto.stanford.edu/pbc/

[37] Murugesan, M., Jiang, W., Clifton, C., Si, L., Vaidya, J.: Efficient privacy-preserving similar document detection. VLDB (2010)

[38] Nagaraja, S., Mittal, P., Hong, C., Caesar, M., Borisov, N.: BotGrep: Finding Bots with Structured Graph Analysis. In: Usenix Security (2000)

[39] Naor, M., Pinkas, B.: Oblivious Transfer and Polynomial Evaluation. In: STOC (1999)

[40] Ogata, W., Kurosawa, K.: Oblivious keyword search. Journal of Complexity **20**(2-3), 356–371 (2004)

[41] Olumofin, F., Goldberg, I.: Privacy-preserving queries over relational databases. In: PETS (2010)

[42] Rabin, M.: How to exchange secrets by oblivious transfer. TR-81, Harvard Aiken Computation Lab (1981)

[43] Raykova, M., Vo, B., Bellovin, S., Malkin, T.: Secure anonymous database search. In: CCSW (2009)

[44] Sherri Davidoff: What Does DHS Know About You? http://philosecurity.org/2009/09/07/what-does-dhs-know-about-you

[45] Song, D., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: S&P (2000)

[46] Yao, A.: Protocols for secure computations. In: FOCS (1982)

[47] Young, E., Hudson, T.: OpenSSL: The Open Source toolkit for SSL/TLS. http://www.openssl.org