Privacy-Preserving Sharing of Sensitive Information

Emiliano De Cristofaro Yanbin Lu Gene Tsudik

Information and Computer Science
University of California, Irvine
{edecrist,yanbinl,gts}@ics.uci.edu

ABSTRACT

The need for controlled sharing of sensitive information occurs in many realistic everyday scenarios, ranging from critical (e.g., national security) to mundane (e.g., social networks). A typical scenario involves two parties, at least one of which seeks some information from the other. The latter is either willing, or compelled, to share information. This poses two challenges: (1) how to enable this type of sharing such that parties learn no (or minimal) information beyond what they are entitled to, and (2) how to do so efficiently, in real-world practical terms.

In this paper, we discuss the concept of Privacy-preserving Sharing of Sensitive Information (PSSI) and provide an efficient database system implementation. The PSSI system functions as a *privacy shield* to protect parties from disclosing their respective sensitive information. Although seemingly simple, the design and deployment of PSSI prompts a number of new and interesting practical challenges, that are addressed in this paper. We present extensive experimental results that attest to the practicality of attained privacy features.

1. INTRODUCTION

In today's increasingly digital world, there is often a tension between safeguarding privacy and sharing information. Clearly, sensitive data needs to be kept confidential. However, in certain scenarios, data owners are either motivated or forced to share (otherwise sensitive) information. Consider the following examples:

Aviation Safety: The Department of Homeland Security (DHS) needs to check whether any passengers on each flight from or to the United States must be denied boarding or disembarkation, based on several secret lists, including the *Terror Watch List* and the *No Fly List*. Today, airlines surrender their entire passenger manifests to the DHS, alongside a large amount of sensitive information, including credit card numbers [29]. Besides its obvious privacy implications, this *modus operandi* poses liability issues with regard to (mostly) innocent passengers' data and concerns about possible data loss.[1] Ideally, the DHS would obtain information pertaining *only* to passengers on one of its watch-lists, without disclosing any information to the airlines.

Healthcare: A health insurance company needs to retrieve information about its client from other parties, e.g., other insurance carriers or hospitals. Clearly, the latter cannot provide any information on other patients, while the former cannot disclose the identity of the target client.

Law Enforcement: An investigative agency (e.g., the FBI) needs to obtain electronic information about a suspect, e.g., from the local police, the military, the DMV, the IRS, or the suspect's employer. In many cases, it is dangerous or forbidden to disclose the subject of the investigation. On the other hand, the other party cannot disclose its entire data-set and trust the FBI to only extract desired information. Furthermore, FBI requests might need to be pre-*authorized* by some appropriate authority, such as, a federal judge. This way, the FBI can only obtain information corresponding to *authorized* requests.

Other examples of sensitive information sharing include recent results in *collaborative* botnet detection [23] – applications where parties share their logs to identify common anomalies, without revealing anything else about them.

Motivated by the examples above, this paper presents the design and implementation of a system for ***Privacy-preserving Sharing of Sensitive Information (PSSI)***. PSSI functions as a *privacy shield* to protect parties from disclosing more than the required minimum of their respective sensitive information. We model PSSI as a simple database-querying application, composed by two parties: a *server*, holding a database, and a *client*, issuing disjunctive equality queries. Recall the Aviation Safety scenario: airline companies maintain databases with passenger information, while the DHS poses queries corresponding to its secret lists (e.g., the Terror Watch List). Our goal is to prevent the airlines from learning the content of the queries, while letting the DHS obtain only records matching those queries.

Intended Contributions. First, we explore the concept of Privacy-preserving Sharing of Sensitive Information (PSSI). We then turn to Private Set Intersection (PSI) techniques and show that they represent the most appropriate building block to achieve both efficiency and provably-secure guarantees. Next, we identify and address a number of challenges in adapting set intersection techniques to realistic database settings. For instance, these techniques usually assume that set items are unique, whereas, most realistic database settings include duplicate records. Furthermore, their strong privacy requirements impose the server to "touch" every single item and to send its entire encrypted database. This may result in large bandwidth overhead and prompt the problem of long-term data safety and associated liability. (Indeed, an encryption scheme considered strong today might gradually weaken in the long term). We propose a novel architecture that addresses these challenges and allows large-scale privacy-preserving database querying. We demonstrate, through experimental analysis, that our solution achieves negligible overhead compared to (non privacy-preserving) MySQL DBMS. Finally, we publish the source code of all our implementations.[2]

[1] See [6] for a litany of recent incidents where large amounts sensitive data were lost or mishandled by government agencies.

[2] Source code is available at: `http://sprout.ics.uci.edu/projects/iarpa-app/code.php`

Organization. In Section 2, we define PSSI along with its privacy requirements, and present cryptographic building blocks. Then, in Section 3, we discuss some challenges stemming from a naïve adaptation of Private Set Intersection (PSI) techniques to PSSI. Section 4 presents our approach using a novel database encryption mechanism. Section 5 identifies two additional challenges in terms of large-scale database and presents a new architecture to address them. Section 6 concludes the paper. In Appendix, we overview related work and present the details and the performance evaluation of all underlying cryptographic protocols.

2. PRELIMINARIES

We first discuss the concept of Privacy-preserving Sharing of Sensitive Information (PSSI), formalize its privacy requirements, and review our cryptographic building blocks.

2.1 Notation

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

Table 1: Notation

We introduce our notation in Table 1. Also, we use $Enc_k(\cdot)$ and $Dec_k(\cdot)$ to denote symmetric key encryption and decryption (under key k), respectively. Public key encryption and decryption – under keys pk and sk – are denoted as $E_{pk}(\cdot)$ and $E_{sk}(\cdot)^{-1}$, respectively. Next, $\sigma = \mathsf{Sign}_{sk}(M)$ denotes the digital signature computed over message M using secret key sk. Operation $\mathsf{Vrfy}_{pk}(\sigma, M)$ returns either 1 or 0, indicating whether σ is a valid signature on M. $\mathbb{Z}_N^*$ refers to an "RSA" group, where N is the RSA modulus. We use $\mathbb{Z}_p^*$ to denote a cyclic group with a subgroup of order q, where p and q are large primes, s.t. $q|p-1$. We use $H(\cdot), H_1(\cdot), H_2(\cdot), H_3(\cdot)$ to denote different hash functions. In practice, we can implement $H(m), H_1(m), H_2(m), H_3(m)$ using SHA-1 [13] as: SHA-1$(0||m)$, SHA-1$(1||m)$, SHA-1$(2||m)$, SHA-1$(3||m)$, respectively.

2.2 PSSI Syntax

The problem of Privacy-preserving Sharing of Sensitive Information (PSSI) is best described as a simple database querying application. In it, a server is holding a database, DB, containing w records with m attributes $(attr_1, \cdots, attr_m)$. That is, $DB = \{(R_j)\}_{j=1}^w$, where each record $R_j = \{val_{j,l}\}_{l=1}^m$, and $val_{j,l}$ is the value of R_j for the attribute $attr_l$. A client poses simple SQL queries, such as:

SELECT * FROM DB

$$\text{WHERE } (attr_1^* = val_1^* \text{ OR} \cdots \text{ OR } attr_v^* = val_v^*) \quad (1)$$

As a result of the query, the client gets all records in DB satisfying the where clause, and *nothing else*. Whereas, the server *learns nothing* about any $\{attr_i^*, val_i^*\}_{1 \le i \le v}$. 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.

Authorized Queries.. In an alternative version with *authorized queries*, we require the client to obtain matching records only if the corresponding $(attr_i^*, val_i^*)$ is pre-authorized by an appropriate Certification Authority (**CA**), i.e., the client holds pertinent authorizations.

2.3 PSSI Privacy Requirements

We now define PSSI privacy requirements. If needed, we distinguish between requirements for standard or authorized queries.

- *Server Privacy.* The client learns no information about any record in server's database that does not satisfy the where clause(s).
- *Server Privacy (Authorized Queries).* The client learns no information about any record in server's database that does not satisfy the where clause(s) or that corresponds to a query not authorized by the CA.
- *Client Privacy.* The server learns nothing about any parameters of client's queries.
- *Client Unlinkability.* The server cannot determine (with probability non-neglibly 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 afterward.

Note that Forward Security and Unlinkability requirements are crucial in many practical scenarios. For instance, recall the Law Enforcement scenario from Section 1. Suppose that the FBI queries an employee database without having authorization for a given suspect, e.g., Alice: Server Privacy 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 recover her file from the (recorded) protocol transcript. This would violate privacy if authorizations are not retroactive.

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 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.

2.4 Building Blocks – Private Set Intersection

Private Set Intersection (PSI) [15] 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. We overview several PSI variants.

PSI with Data Transfer (PSI-DT) [12] 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 learning nothing except v. This variant is appealing in many database scenarios, where the server holds a set of records, rather than a simple set of items.

Authorized PSI-DT (APSI-DT) [12] ensures that client input is *authorized* by an appropriate certification authority (**CA**). Unless it

holds relevant 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 various (A)PSI-DT protocols based on whether they support *pre-distribution*:

- ***(A)PSI-DT with pre-distribution.*** In this variant, 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, just once for all possible clients.
- ***(A)PSI-DT without pre-distribution.*** In this variant, the server cannot pre-process and pre-distribute its input.

Note that several state-of-the-art PSI-DT techniques are reviewed in Appendix B. Observe that pre-distribution precludes Server Unlinkability, since server input is fixed over multiple protocol executions. For the same reason, in the context of authorized protocols, Forward Security cannot be guaranteed with pre-distribution. Therefore, protocols with pre-distribution are preferred in scenarios where server input is mostly static and bandwidth overhead is critical. Whereas, they should be avoided when server database changes frequently, and/or either Server Unlinkability or Forward Security properties are necessary.

3. A FIRST ATTEMPT

It appears possible to meet the privacy requirements (stated in Section 2.3) by using the Private Set Intersection (PSI) techniques outlined in Section 2.4. In this section, we discuss a *strawman* approach to deploy a system for Privacy-preserving Sharing of Sensitive Information (PSSI) using PSI-DT protocols (resp., APSI-DT for authorized queries). However, we show that the strawman solution is not secure, as we discuss below.

The Strawman Solution. 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 correspond to the *where* clause in Equation 1. Optionally, if authorized queries are enforced, $\mathcal{C}$ is accompanied by signatures σ_i over $H(attr_i^*, val_i^*)$, following the APSI-DT syntax. Client and server engage in an (A)PSI-DT interaction and, at the end, the client obtains all records matching its (authorized) query.
The strawman solution has, however, a number of issues, that we discuss below.

Challenge 1: Multi-Sets. By our definitions, PSI-DT and APSI-DT do not support multi-sets, i.e., set items are assumed to be unique. However, most realistic database settings include duplicate records. Although some PSI constructs (e.g., [22]) support multi-sets, their performance is not promising as they involve quadratic overhead (in the size of the sets), and they do not support *data transfer*. However, in all efficient (linear-complexity) (A)PSI-DT constructs, the server sends the client so-called *tags*, i.e., one-way function computations over set items ($(attr_l, val_{j,l})$ in this case), that the client can re-compute (and match) only if the corresponding items are in the intersection. Therefore, tags computed over duplicate $(attr_l, val_{j,l})$ would be identical. Since the entire encrypted database, along with the tags, is transferred to the client, the latter learns all patterns and distribution frequencies. This is problematic, since 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, tag distribution for this attribute would essentially reveal the plaintext, similar to deterministic encryptions.

Challenge 2: Data Pointers. To enable querying by any attribute, each R_j must be copied (and separately encrypted) m times, once for each attribute, and this would incur high storage/bandwidth overhead. This issue can be addressed by encrypting 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 also prompts an additional privacy issue: In order to use the key – instead of the actual record – as "data" in the (A)PSI-DT protocol, a pointer to the encrypted record (on disk or in memory) would have to be stored alongside each tag. This would let the client determine which tags correspond to different attributes of the same record. This (potential) privacy leak is aggravated by the previous issue (multi-sets), since, given two encrypted records, the client can establish their similarity based on the number of tags they have in common. For example, a malicious client could test how many records share exactly two attributes.

4. PRIVACY-PRESERVING SHARING OF SENSITIVE INFORMATION

We now present a secure instantiation of a system for Privacy-preserving Sharing of Sensitive Information (PSSI) based on (A)PSI-DT techniques. We address the two challenges discussed in Section 3 by proposing a novel database encryption mechanism. Note that we select *(A)PSI-DT without pre-distribution* in order to guarantee *Unlinkability* and *Forward Security*.

Any given implementation of (A)PSI-DT without pre-distribution could be used. However, for the sake of clarity, in Figure 1, we show one specific solution using the DT10-1 construction [12]. DT10-1 is the most efficient protocol according to our experimental results. DT10-1 works as follows. Public inputs are p, q, g. First, the client sends to the server $X = [(\prod_{i=1}^{v} hc_i)) \cdot g^{R_c}]$ where $R_c \xleftarrow{r} \mathbb{Z}_q$ and $hc_i = H(attr_i^*, val_i^*)$. Also, for each $1 \le i \le v$, the client sends $y_i = [(\prod_{l \ne i} hc_l) \cdot g^{R_{c:i}}]$, where $R_{c:i} \xleftarrow{r} \mathbb{Z}_q$. The server picks a random R_s in $\mathbb{Z}_q$ and replies with $Z = g^{R_s}$ and $y_i' = y_i^{R_s}$ (for every y_i it receives). Then, it evaluates $Token(hs_{j,l})$ function as $(X/hs_{j,l})^{R_s}$, where $hs_{j,l} = H(attr_j, val_{j,l})$, and invokes $EDB \leftarrow EncryptDatabase$ (Algorithm 1). Also, it sends to the client the encrypted database, $\mathbf{EDB}$, along with Z and $\{y_i'\}_{1 \le i \le v}$. The client, for each of its elements, computes $K_{c:i} = y_i' \cdot Z^{R_c} \cdot Z^{-R_{c:i}}$ and invokes $Lookup$ (Algorithm 2) to recover matching records.

Compared to the original protocol, we modify the "encryption" technique: in step 3(b), instead of straightforwardly using a symmetric-key encryption scheme, the server invokes the $EncryptDatabase$ procedure. Also note that, in step 5(b), the client invokes the $Lookup$ procedure, outlined in Algorithm 2: note that, only if $\exists i, j, l$, such that $hc_i = hs_{j,l}$, then $K_{c:i} = Token(hs_{j,l}) = ((\prod_{l \ne i} hc_l \cdot g^{R_c})^{R_s}$. Thus, step 10 of Algorithm 1 generates a tag matched in step 2 of Algorithm 2.

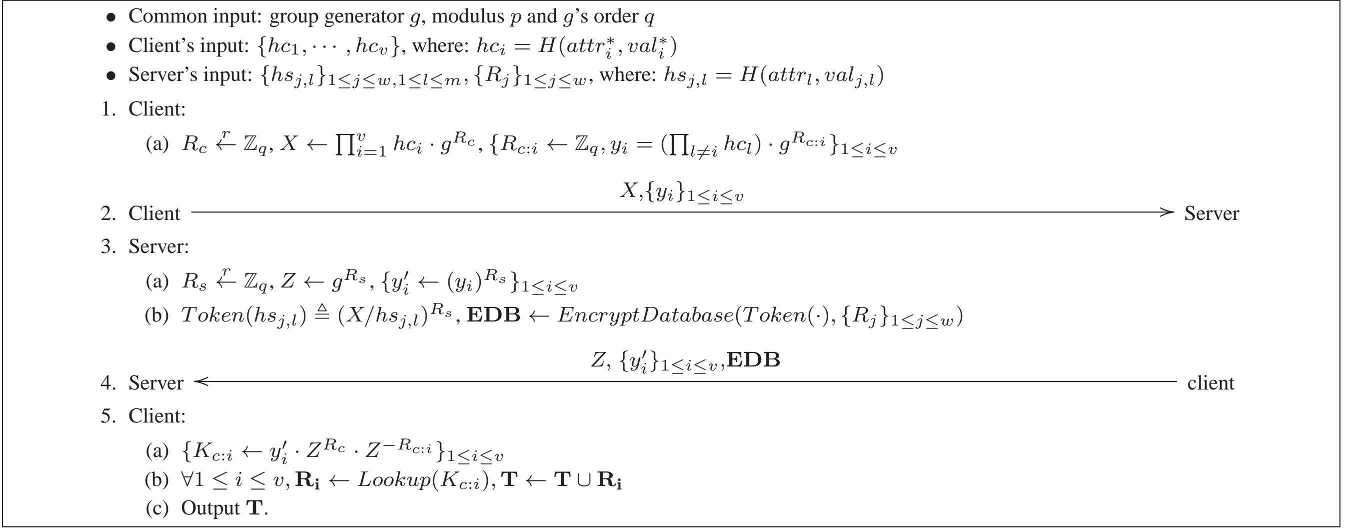


Figure 1: PSI-based PSSI (DT10-1)

Algorithm 1: EncryptDatabase

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

Similarly, one can adapt any APSI-DT without pre-distribution to our PSSI solution for authorized queries. According to our experimental results, the most efficient protocol is DT10-APSI [12]. Since DT10-APSI and DT10-1 are quite similar, the resulting adaptation is almost unaltered and is deferred to the extended version of the paper due to space limitation.

In the rest of the section, we examine the details of the $EncryptDatabase$ and $Lookup$ procedures. Next, we show how our solution addresses the aforementioned challenges, and we present our performance evaluation.

4.1 Database Encryption with Counters

The procedure used for database encryption is presented in Algorithm 1. It is composed of two phases: (1) *record-level* and (2) *lookup-table* encryptions. The Record-level encryption is relatively trivial; it is shown in lines 1–6. First, the server shuffles record locations in the database. Then, it pads each R_j up to the maximum size of all records, picks a random symmetric key k_j, and encrypts R_j as $er_j = Enc_{k_j}(R_j)$.

The Lookup-table (LTable) encryption, shown in lines 8–15, refers to the encryption of attribute name and value pairs. It enables efficient lookup and record decryption.

In step 8, the server hashes a attribute-value pair and uses the hash result as the input of a ***Token*** function in Step 9. In step 10, we use the concatenation of the output from the ***Token*** procedure and a counter, $ctr_{j,l}$, in order to compute the tag $tag_{j,l}$, later used as lookup tag during client query. We use $ctr_{j,l}$ to denote the index of duplicate value for the l-th attribute. In other words, $ctr_{j,l}$ is the counter of times where $val_{j',l} = val_{j,l}, \forall j' <= j$. For example, the third occurrence of value "Smith" for attribute "Last Name" will have $ctr_{j,l} = 3$. The counter guarantees that duplicate $(attr, val)$ pairs will 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. $k''_{j,l}$ 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 distinguish whether or not two tags belong 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).

4.2 Lookup with counters

We describe the *Lookup* procedure in Algorithm 2. It is used to search (in $\mathbf{EDB}$) for all the records that match client's search tokens (computed in step 5(a) of Figure 1). In step 1 of Algorithm 2, the client sets a counter to 1. Next, it searches $\mathbf{LTable}$ for a tag $tag_{j,l}$ such that $tag_{j,l} = H_1(tk||counter)$. If there is a match, the client attempts to recover the record associated to the $tag_{j,l}$. To do so, the client needs to locate the associated record, thus, 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, R_j is recovered upon decryption, i.e., $R_j = Dec_k(er_{j'})$.

Client Query Size	# Records Returned	PSSI				PSSI Authorized Queries				MySQL
		Client (ms)	Server (ms)	Trans. (ms)	**Total**	Client (ms)	Server (ms)	Trans. (ms)	**Total**	**Total**
1	10	4.1	336.2	1802	**2142.3**	8.2	339.8	1802	**2150.0**	**54.7**
10	100	37.6	337.9	1802	**2177.5**	59.4	360.7	1802	**2222.1**	**547.3**
100	1000	371.4	354.4	1802	**2527.8**	571.5	572.1	1802	**2945.6**	**5473.5**

Table 2: Performance Evaluation of our PSSI system for standard and authorized queries.

Algorithm 2: Lookup

input : A 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 $\quad k'' \leftarrow H_3(tk||ctr)$;
4 $\quad j' \leftarrow Dec_{k''}(eind_{j,l})$;
5 $\quad k' \leftarrow H_2(tk||ctr)$;
6 $\quad k \leftarrow Dec_{k'}(ek_{j,l})$;
7 $\quad R_j \leftarrow Dec_k(er_{j'})$;
8 $\quad \mathbf{R} \leftarrow \mathbf{R} \cup R_j$;
9 $\quad ctr \leftarrow ctr + 1$;
10 **end**

4.3 Challenges Revisited

We now review the challenges discussed in Section 3 and discuss how our solution successfully addresses them.

Multi-sets: The counter used 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 LTable, prevents the server from exposing the relationship between an entry $\mathbf{LTable}_{j,l}$ and its associated record R_j.

4.4 Performance Evaluation

We now discuss the performance of our first PSSI solution based on PSI-DT (APSI-DT for authorized queries), using our novel database encryption method. In our analysis, we select DT10-1 and DT10-APSI as the underlying PSI-DT and APSI-DT constructs, respectively. Both protocols are presented in details in Appendix B.

Experimental Setup. We execute the server on an Intel Harpertown server with two Xeon E5420 CPUs (2.5 GHz) and $8GB$ RAM. The client runs on a laptop with Intel Core 2Duo CPU (2.2 GHz) and $4GB$ RAM. The client is connected to the server via Gigabit ethernet. The test database has two attributes – one "searchable" and another added to pad each record to uniform size – and contains $1,000$ records of size $100KB$ each. Each distinct searchable value corresponds to exact 10 distinct records. We select RC4 as the underlying symmetric encryption scheme. We compare our solution to a non privacy-preserving baseline, i.e., using standard MySQL with indexing enabled on the searchable attribute, using the same database and queries, and running on the same machines. Note that results are the average of 10 independent tests.
In Table 2, we summarize our experimental results. Our implementation combines UDP with Selective Repeat ARQ to provide reliable transmission. The first two columns report, respectively, the size of client's query (i.e., the number of predicates in the disjunctive query) and the number of records returned. We measure query execution time incurred by our PSSI solutions (for both standard and authorized queries), resulting from client/server overhead and the time needed to transfer the entire encrypted database. Finally, the *MySQL* column reports total execution time of the non privacy-preserving baseline (measured from the time the query is issued to the time the last response is returned).

Analysis of PSSI Performance. Analyzing the results, we observe that if queries return a small fraction of the database, our privacy-preserving solution incurs, unsurprisingly, a higher computation overhead than MySQL. Indeed, in order to guarantee Client Privacy, the server needs to *touch* every record, thus incurring an overhead linear in the number of database records. In other words, the computational overhead incurred by the cryptographic operations is negligible compared to the time needed to transfer the encrypted database. Indeed, as queries return larger number of records (close to the entire database), the performance slow-down of PSSI solutions tends to be minimized, and it may even be faster than MySql. For instance, if all $1,000$ records are returned, both our PSSI solutions take less than $3s$ to complete, while the MySQL baseline takes more than $5s$. One possible explanation is that MySQL uses TCP for reliable transmission, and this is quite slower than our reliable UDP solution.

Comparing to Private Information Retrieval (PIR). One possible criticism to our solution may be that the communication overhead is *linear* in the size of the database size, whereas, prior work on *Private Information Retrieval* (PIR) [8] incurred *logarithmic* communication overhead to support private database retrieval. Therefore, we investigate whether PIR techniques can be used alongside our solution to reduce bandwidth. First, observe that in PIR queriers need to know the *index* of desired record (indeed, server's database is assumed to be public), whereas we let the client issue SQL queries. Nonetheless, in our solution, the client can discover the index of matching records: if only $\mathbf{LTable}$ is sent by the server, the client recovers the index of matching records in step 4 of Algorithm 2. Thus, it can employ PIR to retrieve the desired record. However, our performance evaluation shows that the *computational* overhead incurred by PIR is so high that we are better off transferring the entire (encrypted) database.

PIR Benchmark. We benchmark Gentry-Ramzan PIR [16], which, to the best of our knowledge, achieves the lowest communication complexity, i.e., $O(\log n)$ for a database with n records (whereas, it incurs $O(n)$ computation overhead). Figure 2 compares the computation time needed by the PIR implementation (i.e., even without including time for data transmission) to the *total* execution time incurred by our PSSI solutions for standard (resp., authorized) queries, based on DT10-1 (resp., DT10-APSI), to execute online operations and to transmit the encrypted database over a gigabit link. For sake of completeness, we evaluate performance using an increasing number of returned records. Our results show that, independently from the database size, even communication-efficient PIR-s yield less efficient solutions compared to transferring the entire "encrypted" database and computing cryptographic operations related to the underlying Private Set Intersection protocols (i.e., DT10-1 and DT10-APSI). Note that this result mirrors the analysis in [30]. In the next section, we discuss a different solution to reduce the bandwidth overhead incurred for large-scale databases.

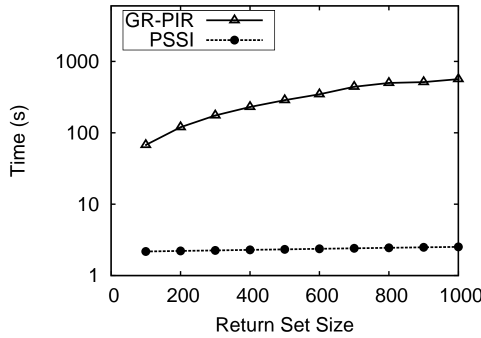


Figure 2: Comparison between computation time of GR-PIR and total query execution time of our PSSI system.

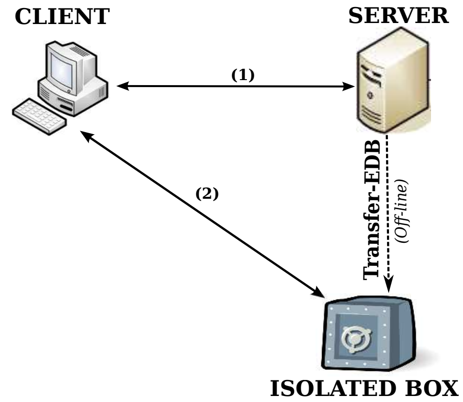


Figure 3: New system architecture with the introduction of the Isolated Box.

5. LARGE-SCALE PSSI

Our PSSI solution based on Private Set Intersection, presented in Section 4, combines efficiency and provably-secure guarantees, and incurs limited overhead compared to standard (non-privacy preserving) MySQL. However, two additional issues may challenge the effectiveness of our solution in the context of large-scale database applications. We discuss them below.

Challenge 3: Bandwidth. If the database is relatively large and/or communication takes place over an expensive/slow channel, the bandwidth overhead may become a critical issue. Indeed, our solution requires the entire encrypted database to be transferred to the client. Furthermore, in Section 4.4, we have investigated and excluded – due to very high computational overhead – the use of Private Information Retrieval techniques to minimize the communication overhead.

Challenge 4: Liability. The transfer of the (entire) 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. Consequently, it is not too far-fetched to imagine that the client might be able to decrypt the database, 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. Furthermore, in a number of scenarios, 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 that effectively addresses all above challenges and still incurs very limited overhead compared to non-privacy preserving database querying systems. Specifically, we show a technique for adapting (A)PSI-DT protocols with pre-distribution to support database encryption and query lookup, and to guarantee Unlinkability and Forward Security at the same time. Next, we discuss challenges and attained privacy features. Finally, we compare the performance of our new solution to that of baseline MySQL.

5.1 New Architecture with Isolated Box

Overview. In Figure 3, we present a novel system architecture, resulting from the introduction of the *Isolated Box* (IB). During the setup phase, the server transfers the encrypted database and the lookup table (LTable) to the IB. In order to pose an SQL query, the client, in step 1, interacts with the server to obtain a set of *tokens*. (Note that this does not reveal query contents). From each token, in step 2, the client derives a set of tags and sends them to the IB, which returns all matching records. (Again, this does not disclose the query target). Observe that our IB-powered system relies on (A)PSI-DT protocols *with* pre-distribution, but guarantees Unlinkability and Forward Security.

Trust Assumptions. We only assume that the IB does not collude with either the server or the client. (However, we discuss the consequences of collusions in Section 5.6). Also, we assume the existence of a private and authentic channel between the client and the server, as well as between the client and the IB, e.g., using SSL/TLS. In practice, the IB can be implemented on a cloud server or a piece of secure hardware installed on server's premises, as long as the server does not learn *what* the IB reads from its storage (i.e., which records) and transfers to the client.

Note that, before the client can interact with the server, the latter needs to transfer a copy of encrypted database to the IB, *offline*. We present the details of the encryption procedure in Section 5.2 and the query procedure in Section 5.3.

5.2 Database Encryption

Our IB-powered solution uses the same encryption procedure presented in Algorithm 1, but uses a different $Token(\cdot)$ function. Indeed, while, in Section 4, we use (A)PSI-DT *without* pre-distribution (i.e., the server cannot run $Token(\cdot)$ before interacting with the client), we now use (A)PSI-DT *with* pre-distribution. Thus, the server can evaluate $Token(\cdot)$ over its inputs *offline*, and then publish the results of the $Token(\cdot)$ evaluations, together with the encrypted database.

Finally, observe that, instead of transferring the encrypted database to the client, the server transfers it to the IB (offline).

5.3 Query lookup

The new *Lookup* procedure is described in Algorithm 3. For ease of exposition, we assume that the client only queries one $(attr, val)$ pair and wants to retrieve the first t matching records. This corresponds to a simple SQL query, e.g., "SELECT * FROM DB WHERE $attr^* = val^*$ LIMIT t". The extension to disjunctive queries (querying multiple $(attr, val)$ pairs) is relatively trivial[3]. We will discuss how to cope with the case where t is omitted from

[3]Disjunctive queries are implemented by treating each equality condition in an "OR" clause as a separate query and removing duplicate responses.

Algorithm 3: Lookup in IB

Step 1: Client anonymously evaluates
$tk^* = \textbf{Token}(attr^*, val^*)$;
Step 2: Client sends to the IB
$\{tag_i^* = H_1(tk^*\|\|i), k_i'' = H_3(tk^*\|\|i)\}_{1\le i\le t}$;
Step 3: IB computes:
for $1 \le i \le t$ **do**
If$(\exists tag_{j,l} \in \textbf{LTable}_{j,l}$ s.t. $tag_{j,l} = tag_i^*)$
$ek_i^* \leftarrow ek_{j,l}$
$j' = Dec_{k_i''}(eind_{j,l})$
$er_i^* \leftarrow er_{j'}$
end
end
Step 4: IB transfers $\{ek_i^*, er_i^*\}_{1\le i\le t}$ to the client.
Step 5: Client computes:
for $1 \le i \le t$ **do**
$k_i' = H_2(tk^*\|\|i)$
$k_i = Dec_{k_i'}(ek_i^*)$
$R_i = Dec_{k_i}(er_i^*)$
end

client's query in Section 5.4.

In step 1 of Algorithm 3, the client runs any (A)PSI-DT protocol with pre-distribution over a singleton set with $\{(attr^*, val^*)\}$ as its input, and obliviously evaluates $tk^* = Token(attr^*, val^*)$ with the server. In step 2, the client sets a counter i from 1 to t, and computes a set of tags $\{tag_i^* = H_1(tk^*\|\|i)\}_{1\le i\le t}$ and a set of index decrypting keys $\{k_i'' = H_3(tk^*\|\|i)\}_{1\le i\le t}$. Next, the client sends $\{tag_i^*, k_i''\}_{1\le i\le t}$ to the IB. In step 3, for each $i \in [1, t]$, the IB searches for tag_i^* in LTable. If there is no result, the IB adds $\perp$ in the response set. If a tuple $(tag_{j,l}, ek_{j,l}, eind_{j,l})$ is found (where $tag_{j,l} = tag_i^*$), the IB decrypts $eind_{j,l}$ and recovers index j' by running $Dec_{k''}(ek_{j,l})$. The IB then adds $er_{j'}$ and $ek_{j,l}$ (which are equal to er_i^* and ek_i^*, respectively) to the response set. In step 4, the IB returns the response set $\{ek_i^*, er_i^*\}_{1\le i\le t}$ to the client. In step 5, the client computes a set of decrypting keys $\{k_i' = H_2(tk^*\|\|i)\}_{1\le i\le t}$. For each $i \in [1, t]$, it obtains the decryption key $k_i = Dec_{k'}(ek_i^*)$, and decrypts er_i^* by $R_i = Dec_{k_i}(er_i^*)$.

5.4 Optimizations

If t is too large (i.e., there are less than t matching records) or it is simply omitted from the query, computing all the tag_i^* and k_i'' at once in step 3 might be time-consuming. Note that the client can retrieve records one by one from the IB by gradually incrementing the counter i in each round. To address this, we let the client compute only one tag_i^* and k_i'' each time and pipe-line the computation of tag_{i+1}^* and k_{i+1}'' with the retrieval of ek_i^* and er_i^* (step 4–5). The query terminates when either t responses or $\perp$ is received. This way, the overhead incurred in step 3 is related to the computation of only one tag and one key. Also, the client does not need to estimate how many tags and keys to compute in step 3.

Observe that we can further optimize the computation of $ek_{j,l}$ and $eind_{j,l}$ (steps 13–14 in Algorithm 1). Since we use a counter as part of the input to compute $k_{j,l}'$ (respectively, $k_{j,l}''$), each $k_{j,l}'$ (respectively, $k_{j,l}''$) is different from 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 Alg. 3, $Dec_{k_i''}(eind_{j,l})$ becomes $k_i'' \oplus eind_{j,l}$ and $Dec_{k_i'}(ek_i^*)$ changes to $k_i' \oplus ek_i^*$.

5.5 New Challenges Revisited

Since we reuse the encryption procedure discussed in Section 4, Challenge 1 and 2 are straightforwardly addressed. Therefore, we only discuss Challenge 3 and 4.

Bandwidth: Once the server transfers its database (offline) to the IB, the latter sends only records matching the query back to the client, thus minimizing bandwidth consumption.

Liability: Since the IB holds the encrypted database, the client only obtains the result of its queries.

Finally, note that the introduction of the IB allows guaranteeing Unlinkability and Forward Security, despite we employ (A)PSI-DT techniques *with* pre-distribution.

5.6 Discussion

Privacy Revisited. The introduction of the IB and the use of counter mode in database encryption provides additional privacy properties. We use the term *transaction* to denote a complete query procedure (from the time a SQL query is issued, until the last response from the IB is received). *Retrieval* denotes the receipt of a single response record during a transaction. We observe that, if the client performs only one query transaction, as in Algorithm 3, the IB can link all tag_i^* values in step 3 to the same $(attr, val)$ pair. This may pose a similar risk to that discussed in Challenge 1. 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 delays 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.

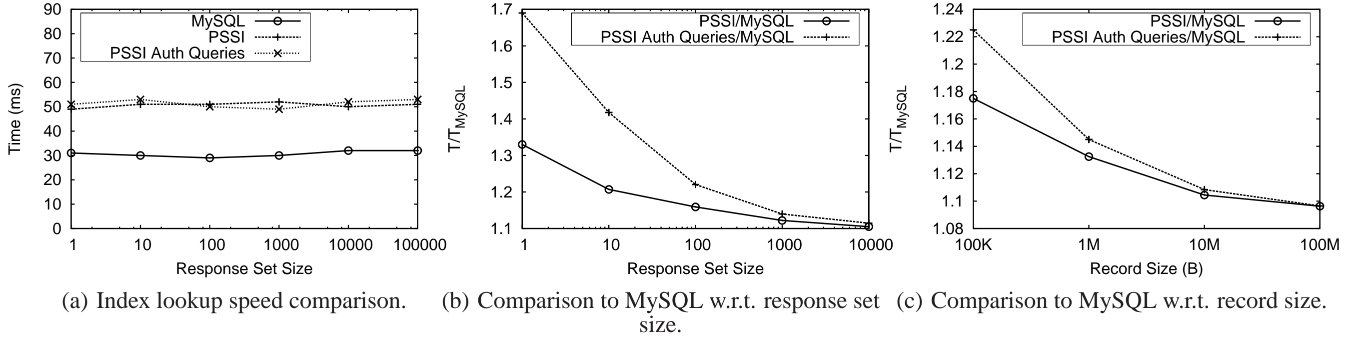
We also note that the introduction of the IB does not violate Client or Server Privacy. Client Privacy is preserved because the client (obliviously) computes a token, which is not learned by the server. The IB does not learn client's interests, since client's input to the IB (tag) is statistically indistinguishable from a random value. 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.

Limitations. We acknowledge that our PSSI system 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. 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 whole encrypted database, hence, liability becomes a problem. Last, Server Unlinkability is actually guaranteed 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 we currently support only equality and disjunctive SQL queries. Whereas, supporting 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.

5.7 Comparison to MySQL

We now discuss the performance of our second PSSI solution, geared to address the new challenges discussed in the context of



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

Figure 4: Performance comparison between our PSSI implementations and non-privacy preserving MySql baseline

large-scale database applications.

Recall that we use PSI-DT and APSI-DT protocols with pre-distribution. In our analysis, we select DT10-2 and IBE-APSI as the underlying PSI-DT and APSI-DT constructs with pre-distribution, respectively. Both protocols are presented in details in Appendix B.

We use the same system setup as Section 4.4 with the IB and Server running on the same machine. The test database has now 45 searchable attributes and 1 unsearchable attribute used to pad each record to uniform size. There are, in total, $100,000$ records. All records have the same size, which we vary during experiments. Again, we compare our solution to a non privacy-preserving baseline, i.e., using standard MySQL with indexing enabled on the searchable attributes, using the same database and queries, and running on the same machines. Note that results are average of 10 independent tests.

First, we compare *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 4(a) shows index lookup time for DT10-2, IBE-APSI, and MySQL with respect to the response set size. Both DT10-2 and IBE-APSI incur almost the same overhead and are only 1.5 times more expensive than MySQL. We also measure index lookup time with respect to general record size. Since the results are similar to the previous experiment, we omit them here due to space limitation.

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 4(b) shows the time for the client to complete a query for a specific response set size divided by the time taken by MySQL. Results gradually converge to 1.1 for increasing response set sizes. This is because of the extra delay incurred by cryptographic operations being amortized by subsequent data lookups and decryptions.

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 4(c) shows the ratio between DT10-2 and MySQL, IBE-APSI and MySQL, respectively. Again, results gradually converge to 1.1 with increasing of record size which occurs because the overhead of symmetric record decryption becomes dominant with growing record size.

In summary, both index lookup time and total query time of our implementation are strictly less than double their respective counterparts in MySQL.

6. CONCLUSIONS & FUTURE WORK

In this paper, our main goal was to drag some useful and efficient privacy-preserving tools out of their cryptographic "closet" and use them to construct practical systems for Privacy-preserving Sharing of Sensitive Information (PSSI). Indeed, we have shown that, leveraging efficient Private Set Intersection techniques, we can construct privacy-enhanced database systems that are efficient enough to be used in real-world large-scale applications. In doing so, we encountered (and addressed) a number of interesting issues that led to specific architectural choices. As confirmed by experimental evaluation, our PSSI implementations incur reasonable additional cost over a base-line (non privacy-preserving) MySQL implementation.

Nonetheless, much remains to be done: First, we do not currently support conjunctive (AND) queries across multiple attributes. This is not difficult to do naïvely, however, the overhead is likely to be quite high (exponential in the number of attributes). We also plan to explore ways to support "fuzzy" querying, i.e., where client's input represents non-normalized data.

7. REFERENCES

[1] M. Belenkiy, J. Camenisch, M. Chase, M. Kohlweiss, A. Lysyanskaya, and H. Shacham. Randomizable proofs and delegatable anonymous credentials. In *Crypto*, 2009.

[2] E. Bertino, J. Byun, and N. Li. Privacy-preserving database systems. *Foundations of Security Analysis and Design*, 2005.

[3] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Eurocrypt*, 2004.

[4] D. Boneh and M. K. Franklin. Identity-based encryption from the weil pairing. *SIAM Journal of Computing*, 32(3), 2003.

[5] J. Camenisch and G. M. Zaverucha. Private intersection of certified sets. In *Financial Cryptography and Data Security*, 2009.

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

[7] B. Chor, N. Gilboa, and M. Naor. Private information retrieval by keywords. *Manuscript*, 1998.

[8] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *Journal of the ACM*, 45(6), 1998.

[9] S. Chow, J. Lee, and L. Subramanian. Two-party computation model for privacy-preserving queries over distributed databases. In *NDSS*, 2009.

[10] J. Daeman and V. Rijmen. AES proposal: Rijndael. 1999.

[11] E. De Cristofaro, S. Jarecki, J. Kim, and G. Tsudik. Privacy-preserving policy-based information transfer. In *PETS*, 2009.

[12] E. De Cristofaro and G. Tsudik. Practical private set intersection protocols with linear complexity. In *Financial Cryptography and Data Security*, 2010.

[13] D. Eastlake and P. Jones. US Secure Hash Algorithm 1, 2002.

[14] M. Freedman, Y. Ishai, B. Pinkas, and O. Reingold. Keyword search and oblivious pseudorandom functions. In *TCC*, 2005.

[15] M. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. In *Eurocrypt*, 2004.

[16] C. Gentry and Z. Ramzan. Single-database private information retrieval with constant communication rate. In *ICALP*, 2005.

[17] Y. Gertner, Y. Ishai, E. Kushilevitz, and T. Malkin. Protecting data privacy in private information retrieval schemes. In *STOC*, 1998.
[18] C. Hazay and Y. Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. In *TCC*, 2008.
[19] B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. In *VLDB*, 2004.
[20] S. Jarecki and X. Liu. Efficient Oblivious Pseudorandom Function with Applications to Adaptive OT and Secure Computation of Set Intersection. In *TCC*, 2009.
[21] S. Jarecki and X. Liu. Fast secure computation of set intersection. In *SCN*, 2010.
[22] L. Kissner and D. Song. Privacy-preserving set operations. In *CRYPTO*, 2005.
[23] S. Nagaraja, P. Mittal, C. Hong, M. Caesar, and N. Borisov. BotGrep: Finding Bots with Structured Graph Analysis. In *Usenix Security*, 2010.
[24] W. Ogata and K. Kurosawa. Oblivious keyword search. *Journal of Complexity*, 20(2-3), 2004.
[25] F. Olumofin and I. Goldberg. Privacy-preserving queries over relational databases. In *PETS*, 2010.
[26] M. Rabin. How to exchange secrets by oblivious transfer. TR-81, Harvard Aiken Computation Lab, 1981.
[27] M. Raykova, B. Vo, S. Bellovin, and T. Malkin. Secure anonymous database search. In *CCSW*, 2009.
[28] R. Rivest. The RC4 Encryption Algorithm. 1992.
[29] Sherri Davidoff. What Does DHS Know About You? `http://philosecurity.org/2009/09/07/what-does-dhs-know-about-you`.
[30] R. Sion. On the computational practicality of private information retrieval. In *NDSS*, 2007.
[31] D. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy*, 2000.
[32] B. Waters, D. Balfanz, G. Durfee, and D. Smetters. Building an encrypted and searchable audit log. In *NDSS*, 2004.
[33] A. Yao. Protocols for secure computations. In *FOCS*, 1982.

APPENDIX

A. RELATED WORK

Several cryptographic primitives provide privacy properties comparable to those listed in Section 2.3. Below, we discuss related primitives and motivate our design choices.

Secure Two-Party Computation. Two parties, on input x and y, respectively, can use Secure Two-Party Computation (2PC) to privately compute the value of a public function f at point (x, y). Both parties learn $f(x, y)$ and nothing else. A general procedure for 2PC of any function expressed as a Boolean circuit is due to Yao [33]. Although one could implement PSSI with 2PC, this technique would incur impractical computation and communication complexities – at least quadratic, as pointed out in [15, 22].

Oblivious Transfer (OT). Introduced by Rabin [26], OT involves a sender with n secret messages and a receiver with one index i. The receiver wants to retrieve the i-th among sender's messages. The sender does not learn which message is retrieved, and the receiver learns no other message. OT privacy requirements resemble those of PSI-DT. However, in PSI-DT, inputs are items (e.g., keywords), whereas, in OT, the receiver needs to know (and input) an existing index – an unrealistic assumption for the applications we have in mind.

Private Information Retrieval (PIR). PIR [8] allows a client to retrieve an item from a server database without revealing which item it is retrieving, incurring a communication overhead strictly lower than $O(n)$ (if n is the database size). Note that, in PIR, privacy of server's database is not protected – the client may receive items/records beyond those requested. Symmetric PIR (SPIR) [17] 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 know and input the index of the desired item in server's database. An extension to keyword-based retrieval is Keyword-PIR (KPIR) [7]. However, KPIR is still focused on minimizing bandwidth, rather than optimizing computation or protecting server's privacy, and it incurs significantly higher computational overhead, as well as multiple rounds of PIR executions.

Oblivious Keyword Search [24]. This primitive is akin to a special case of PSI-DT, where client input set is a singleton, i.e., the set has only one element. The k-out-of-n oblivious keyword search [24] is very similar to the DT10-2 [12] protocol described in Appendix B.2, while the one in [14] is very similar to the FNP04 [15] protocol described in Appendix B.1. The main difference between PSI-DT and OKS is that, in OKS, server input may contain duplicate keywords. Whereas, we discuss how to deal with both duplicate keywords (Multi-Sets) and duplicate data (Data Pointers) in Section 4. Due to similar functionality and efficiency, we do not consider OKS as a candidate building block.

Searchable Encryption. Searching on encrypted data (SoE) was introduced, in the symmetric key setting, in [31]. This primitive allows a client to store its encrypted data on an untrusted server and later search for a specific keyword by giving the server a search capability that does not reveal the keyword or any plaintext. We cannot use SoE to implement PSSI because the client cannot encrypt or access the entire server's database. Note that the problem of PSSI with authorized queries could be solved using *Public-key Encryption with Keyword Search* (PEKS) [3] (or, similarly, searchable encrypted logs [32]), based on Identity-based Encryption (IBE) [4]. The database server could append a so-called PEKS for each pair of $(attr_l, val_{j,l})$ to the encrypted data record R_j and send the whole encrypted database to the client. Whereas, the client could "test" a predicate $attr^* = val^*$ (and obtain associated data) only if it has a corresponding trapdoor, i.e., an authorization of $(attr^*, val^*)$. Nonetheless, observe that the Test algorithm in PEKS requires the client to test each trapdoor against each encrypted attribute-value pair it receives, thus incurring *quadratic* computational overhead.

Related Privacy-Preserving Database Systems. Privacy-preserving database querying has already been considered by related work. Although some prior results support more complex query types (i.e., not only equality and disjunctive queries), they exhibit certain limitations. For instance, Olumofin and Goldberg [25] construct a novel Keyword-based PIR (KPIR [7]) providing a transition from block-based PIR to SQL-enabled PIR. However, similar to KPIR-s, it incurs high computational complexity (several order of magnitudes higher than our solutions), and lets clients obtaining other bits of database data (beyond the query result). Next, solutions in [19, 2] do not provide rigorous (provably-secure) privacy guarantees. Finally, [27, 9] require several independent trusted parties, unlike our IB-based solution (presented in Section 5), which involves only one semi-trusted party (implemented on a secure hardware). In particular, the work in [9] proposes a two-party query computation model over distributed databases that involves three entities: a randomizer, a computing engine, and a query front-end. Local answers to queries are randomized by each database and aggregate results are de-randomized at the front-end.

B. STATE-OF-THE-ART PSI-S

In the following, we review state-of-the-art PSI protocols and focus on PSI-DT variants. In the rest of the section, we assume client and server set sizes are v and w, respectively.

B.1 PSI-DT without Pre-Distribution

FNP04. Freedman, Nissim, and Pinkas [15] use *oblivious polynomial evaluation* to implement PSI. Their approach can be slightly modified to support PSI-DT. The modified protocol – denoted as FNP04 – works as follows: the client first setups an additively homomorphic encryption scheme, such as Paillier, with key pair (pk_c, sk_c). Client defines a polynomial $f(y) = \prod_{i=1}^{v}(y - c_i) = \sum_{i=0}^{v} a_i y^i$ whose roots are its inputs. It encrypts each coefficient a_i under its public key pk_c and sends encrypted coefficients $\{Enc_{pk_c}(a_i)\}_{i=0}^{k}$ to the server. Since the encryption is homomorphic, the server can evaluate $Enc(f(s_j))$ for each $s_j \in \mathcal{S}$ independenlty from the client. Then, the server returns $\{(Enc(r_j \cdot f(s_j) + s_j), Enc(r'_j \cdot f(s_j) + data_j))\}_{j=0}^{n}$ to the client where r_j and r'_j are fresh random numbers for each input in $\mathcal{S}$. Client, for each returned pair (e_l, e_r), decrypts e_l by computing $c' = Dec_{sk_c}(e_l)$. Then if $c' \in \mathcal{C}$, the client continues to decrypt e_r and gets the associated data. Otherwise, the client only gets some random value and moves onto the next returned pair. In order to speed up the performance, FNP04 can use modified ElGamal encryption instead of Paillier. Specifically, the client uses g^{a_i} instead of a_i as the input to the ElGamal encryption where g is a generator with order q modulo p. And when it decrypts e_l, it recovers $g^{c'}$. Client can still decide whether $c' \in \mathcal{C}$ by comparing $g^{c'}$ to $g^{c_i}, \forall c_i \in \mathcal{C}$. In terms of data, the server can choose a random key g^{k_j} and uses it to symmetrically encrypt $data_j$. Then the server sends $\{(Enc(r_j \cdot f(s_j) + s_j), Enc(r'_j \cdot f(s_j) + k_j), Enc_{g^{k_j}}(data_j))\}_{j=0}^{w}$ to the client. If the client can recover g^{k_j}, it can also decrypt $data_j$. Using balanced bucket allocation to speed up operations, client overhead is dominated by $O(v + w)$ $|q|$-bit mod p exponentiations (in ElGamal). Whereas, server overhead is dominated by $O(w \log \log v)$ $|q|$-bit mod p exponentiations.

KS05. Kissner and Song [22] also use oblivious polynomial evaluation to construct a variety of set operations. However, their solution is designed for mutual intersection over *multi-set* that may contain duplicate elements, and it is unclear how to adapt it to transfer associated data. Also, their technique incurs quadratic ($O(vw)$) computation (but linear communication) overhead. As we use a different method to handle multi-sets (see Section 4) and we only consider one-way PSI, we do not consider KS05 any further.

DT10-1. De Cristofaro and Tsudik present an unlinkable PSI-DT protocol (Fig. 3 in [12]) with linear computation and communication complexities. This protocol, denoted as DT10-1, operates as follows: The setup phase yields primes p (e.g. 1024 bits) and q (e.g. 160 bits), s.t. $q|p-1$, and a generator g with order q modulo p. In the following, we assume computation is done mod p. First, the client sends to the server $X = [(\prod_{i=1}^{v} H(c_i)) \cdot g^{R_c}]$ where R_c is randomly selected from $\mathbb{Z}_q$. Also, for each $1 \le i \le v$, the client sends $y_i = [(\prod_{l \ne i} H(c_l)) \cdot g^{Rc:i}]$, where the $R_{c:i}$'s are random in $\mathbb{Z}_q$. The server picks a random R_s in $\mathbb{Z}_q$ and replies with $Z = g^{R_s}$ and $y'_i = y_i^{R_s}$ (for every y_i it received). Also, for each item s_j ($1 \le j \le w$), it computes $K_{s:j} = (X/H(s_j))^{R_s}$, and sends the *tag* $t_j = H_1(K_{s:j})$ with the associated data record encrypted under $k_j = H_2(K_{s:j})$. The client, for each of its elements, computes $K_{c:i} = y'_i \cdot Z^{R_c} \cdot Z^{-R_{c:i}}$ and the tag $t'_i = H_1(K_{c:i})$. Only if c_i is in the intersection (i.e., there exists an element $s_j = c_i$), the client finds a pair of matching tags (t'_i, t_j). Besides learning the elements intersection, the client can decrypt associated data records by key $H_2(K_{c:i})$. Client overhead amounts to $O(v)$ $|q|$-bit modulo p exponentiations and multiplications and server overhead is $O(v+w)$ $|q|$-bit modulo p exponentiations.

B.2 PSI-DT with Pre-Distribution

JL09. Jarecki and Liu [20] (following the idea in [18]) give a PSI-DT based on Oblivious PRF (OPRF) [14]. We denote this protocol as JL09 (and present the improved OPRF construction discussed in [1]). Recall that an OPRF is a two-party protocol that securely computes a pseudorandom function $f_k(\cdot)$, on key k contributed by a server and input x contributed by a client, such that the server learns nothing about x, while the client learns $f_k(x)$. The main idea is the following: For every item $s_j \in \mathcal{S}$, the server publishes a set of pair $\{H_1(f_k(s_j)), Enc_{H_2(f_k(s_j))}(data_j)\}$. Then, the client, for every item $c_i \in \mathcal{C}$, obtains $f_k(c_i)$ by OPRF with the server. As a result, the client can use $H_1(f_k(c_i))$ to check if $c_i \in \mathcal{C} \cap \mathcal{S}$ and if so then it uses $H_2(f_k(c_i))$ to recover $data_j$. JL09 incurs $O(w+v)$ server exponentiations, and $O(v)$ client exponentiations. Exponentiations are $|N|$-bit modulo N^2, where N is the RSA modulus.

JL10. Another recent work by Jarecki and Liu [21] (denoted as JL10) leverages an idea similar to JL09 [20] to achieve PSI-DT. Instead of using OPRF, JL10 uses the newly-introduced *Parallel Oblivious Unpredictable Function* (POUF), $f_k(x) = (H(x)^k \bmod p)$, in the Random Oracle Model. In order to obliviously compute $f_k(x)$, the client first picks a random exponent α and sends $y_j = H(c_j)^\alpha$ to the server. The server replies to the client with $z_j = (y_j)^k$. Then the client recovers $f_k(x) = z^{1/\alpha}$. The computational complexity of this protocol amounts to $O(v)$ online exponentiations for both server and client, as the server can pre-process (offline) its $O(w)$ exponentiations. Exponentiations are q-bit modulo p, similar to DT10-1.

DT10-2. In Fig. 4 of [12], De Cristofaro and Tsudik present a PSI-DT based on blind-RSA signatures in the Random Oracle Model (ROM). We denote this protocol as DT10-2. The protocol uses the hash of RSA signatures as a PRF in ROM and achieves the same asymptotic complexities as DT10-2 and JL10, but (1) the server now computes RSA signatures (e.g., 1024-bit exponentiations), and (2) client workload is reduced to only multiplications if the RSA public key, e, is chosen short enough (e.g., $e = 3$).

In summary, we consider JL09, JL10 and DT10-2 in the context of PSI-DT *with* pre-distribution. Note that, although faster than protocols without pre-distribution, these protocols do not achieve Server Unlinkability.

B.3 APSI-DT without Pre-distribution

DT10-APSI. In Fig.2 of [12], De Cristofaro and Tsudik also present an APSI-DT technique mirroring its PSI-DT counterpart, DT10-1. We denote this protocol as DT10-APSI. It operates as follows: the client first obtains authorization from the court for its element c_i, where an authorization corresponds to an RSA-signature: $\sigma_i = H(c_i)^d$. Then, the client sends the server $X = [(\prod_{i=1}^{v} \sigma_i) \cdot g^{R_c}]$ for a random R_c. Then, for each element c_i, it sends $y_i = [(\prod_{l \ne i} \sigma_l) \cdot g^{Rc:i}]$, where the $R_{c:i}$'s are additional random values. The server picks a random value, R_s, and replies with $Z = g^{eR_s}$, $y'_i = y_i^{eR_s}$ (for each received y_i). Also, for each element s_j, she computes $K_{s:j} = (X^e/H(s_j))^{R_s}$, and sends the *tag* $t_j = H_1(K_{s:j})$ and the associated data record encrypted under the key $k_j = H_2(K_{s:j})$. Client, for each of its elements, computes $K_{c:i} =$

$y'_i \cdot Z^{R_c} \cdot Z^{-R_{c:i}}$ and the tag $t'_i = H_1(K_{c:i})$. Client can find a pair of matching tag (t'_i, t_j) only if c_i is in the intersection and σ_i is a valid signature on c_i, Besides learning the elements in the intersection, the client can decrypt associated data records. The computation overhead is $O(v)$ exponentiations for the client, and $O(v + w)$ – for the server. Exponentiations are $|N|$-bit modulo N, where N is the RSA modulus.

CZ09. Camenisch and Zaverucha [5] provide mutual set intersection with authorization on both parties' input. The proposed protocol builds upon oblivious polynomial evaluation and has quadratic computation and communication overhead. Also, it does not provide data transfer.

As a result, we only consider the DT10-APSI protocol in the context of APSI-DT *without* pre-distribution. Note that DT10-APSI provides both Server and Client Unlinkability, as well as Forward Security.

B.4 APSI-DT with Pre-distribution

IBE-APSI. The protocol in Fig. 5 of [11] presents a protocol based on Boneh-Franklin Identity-based Encryption [4], which can be adapted to APSI-DT with pre-distribution. We denote this protocol as IBE-APSI. Note that such a construct is described in the context of a different primitive – Privacy-Preserving Information Transfer (PPIT). However, it can be converted to APSI-DT.
First, the authorization authority (acting as the IBE PKG) generates a prime q, two groups $\mathbb{G}_1, \mathbb{G}_2$ of order q, a bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_1 \rightarrow \mathbb{G}_2$. A random $s \in \mathbb{Z}_q$ is selected as a secret master key. Then, a random generator $P \in \mathbb{G}_1$ is chosen, and Q is set such that $Q = s \cdot P$. (P, Q) are public parameters. Client obtains authorization for an element c_i as an IBE secret key, $\sigma_i = s \cdot H(c_i)$. In the pre-distribution phase, the server first selects a random $z \in \mathbb{G}_1$ and then, for each $(s_j, data_j)$, publishes (t_j, e_j) where $t_j = H_1(e(Q, H(s_j))^z)$ and e_j is the IBE encryption of $data_j$ under identifier s_j. Then, the server gives the client $R = zP$ and the client computes $t'_i = H_1(e(R, \sigma_i))$. For any t'_i, s.t. $t'_i = t_j$, the client can decrypt e_j. The protocol can be speeded up by encrypting e_j under symmetric key $H_2(e(Q, H(s_j))^z)$. The computation overhead for the client amounts to $O(v)$ pairing operations, while there is no online overhead for the server.
Remark that IBE-APSI has two drawbacks compared to APSI-DT: it provides neither Server Unlinkability nor Forward Security.

C. BENCHMARKING (A)PSI-DT CONSTRUCTS

In this section, we benchmark several (A)PSI-DT protocols and compare their performance through experimental results. During the process, we try to identify the most efficient (A)PSI-DT protocols (with or without pre-distribution), and select the building blocks of our PSSI solutions.

Candidate Protocols. We discuss efficient implementation of the following (A)PSI-DT protocols:

	w/o Pre-Distribution	w/ Pre-Distribution
PSI-DT	FNP04 ([15]), DT10-1 (Fig.3 in [12])	JL09 ([20]), JL10 ([21]), DT10-2 (Fig.4 in [12])
APSI-DT	DT10-APSI (Fig.2 in [12])	IBE-APSI (Fig.5 in [11])

Table 3: Candidate PSI-DT and APSI-DT protocols.

Each protocol was implemented in C++ using GMP (ver. 5.01) and PBC (ver. 0.57) libraries. All benchmarks were collected on a Ubuntu 9.10 desktop platform with Intel Xeon E5420 CPU (2.5GHz and 6MB cache) and 8GB RAM.

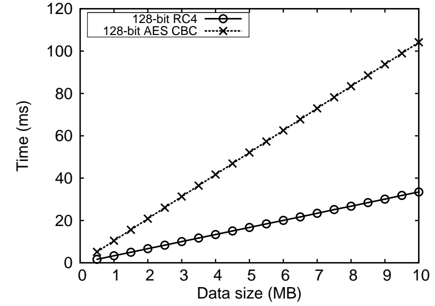


Figure 5: Symmetric key en-/de-cryption performance.

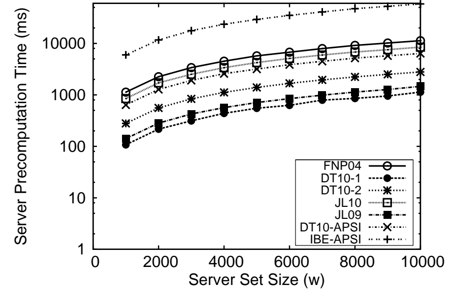


Figure 6: Server pre-computation overhead.

For protocols supporting data transfer, data associated with each server element can be arbitrarily long. Also, performance of some protocols is dominated by each element's data size, rather than set size (e.g., in FNP04). In order to obtain a fair comparison, we need to capture the "intrinsic" cost of each protocol. To this end, we employ the following strategy to eliminate data size effects: First, in all protocols, we encrypt each element's data with a distinct random symmetric key and consider these keys as the new associated data. Assuming that a different key is selected at each interaction, this technique does not violate Server Unlinkability. This way, the computation cost of each protocol is measured based on the same fixed-length key, regardless of data size. In our experiments, we set symmetric key size to 128 bits.

As a result, each protocol execution involves additional overhead of symmetric en-/de-cryption of records. Figure 5 compares the resulting overhead (for variable data sizes), using either RC4 [28] or AES-CBC [10] (with 128-bit keys). Therefore, to estimate the total cost of a protocol, one needs to combine: (1) symmetric encryption overhead, (2) computation cost of each protocol, and (3) data transfer delay for transmitting the encrypted data and PSI values.

We further assume that the client does not perform any pre-computation, while the server performs as much pre-computation on its input as possible. This reflects the reality where client input is (usually) determined in real time, while server input is pre-determined. Figure 6 shows the pre-computation overhead for each protocol.

Next, we evaluate online computation overhead. Figures 7 and 8 present client online computation overhead with respect to client and server input sizes, respectively. Figures 9 and 10 show server online computation overhead with respect to client and server input size, respectively.

Furthermore, Figures 11 and 12 evaluate protocol bandwidth complexity with respect to client and server input sizes. For protocols with pre-distribution, bandwidth consumption (since the transfer

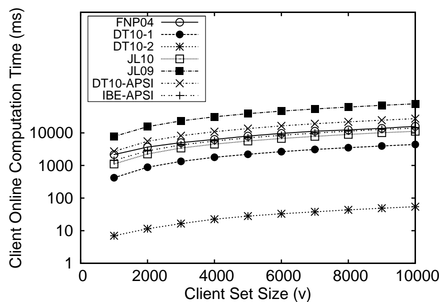


Figure 7: Client online computation w.r.t. client set size.

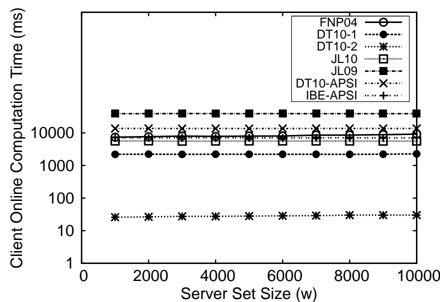


Figure 8: Client online computation w.r.t. server set size.

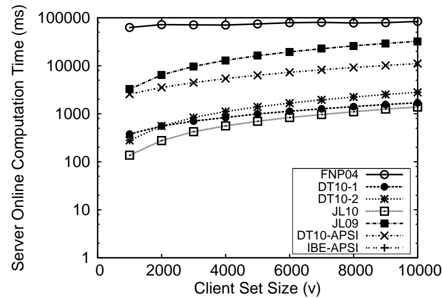


Figure 9: Server online computation w.r.t. client set size.

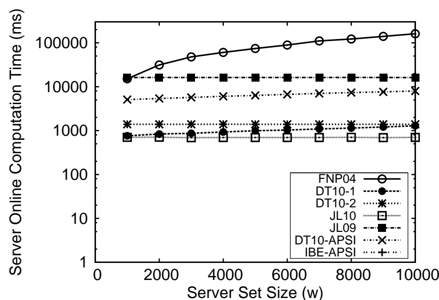


Figure 10: Server online computation w.r.t. server set size.

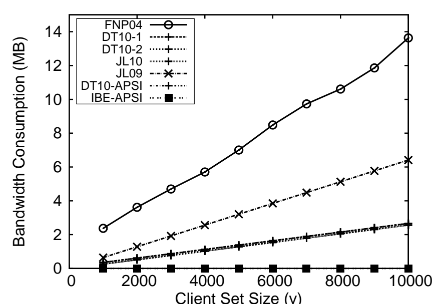


Figure 11: Bandwidth consumption w.r.t. client set size.

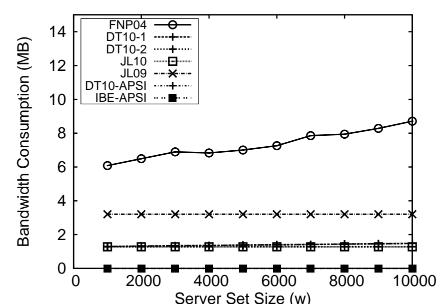


Figure 12: Bandwidth consumption w.r.t. server set size.

of database encryption is performed offline) does not include pre-distribution overhead. Note that, in these figures, we sometimes use the same marker for different protocols to indicate that these protocols share the same value. Client input size v (resp., server input size w) is fixed at $5,000$ in figures where x-axis refers to the server (resp., the client) input size.

Finally, note that, in all experiments, we use a 1024-bit RSA modulus and a 1024-bit cyclic-group modulus with a 160-bit subgroup order. All test results are averaged over 10 independent runs. All protocols are instantiated under the assumption of *Honest-but-Curious* (HbC) adversaries and in the *Random Oracle Model* (ROM).

PSI-DT without pre-distribution. We now focus on the comparison between FNP04 and DT10-1. Figures 7-12 show that that FNP04 is much costlier than DT10-1 in terms of client and server online computation as well as bandwidth consumption. For each client set size, DT10-1 client overhead ranges from $460ms$ to $4,400ms$, while FNP04 server overhead – between $1,300ms$ and $15,000ms$. For each chosen server set size, server overhead in DT10-1 is under $1,300ms$, while, in FNP04, it exceeds $15,000ms$.

PSI-DT with pre-distribution. Next, we compare JL09, JL10 and DT10-2, i.e., PSI-DTs with pre-distribution. Recall that all protocols are instantiated in the HbC model, thus ZKPK's are not included for JL09 and JL10. Figures 7-12 show that DT10-2 incurs client overhead almost two orders of magnitude lower than JL09 and JL10. Indeed, DT10-2 involves two client multiplications for each item, while JL09 performs two heavy homomorphic operations and JL10 – two exponentiations. In JL10, the server online computation overhead results from v 160-bit exponentiations, whereas, in DT10-2, it results from v RSA exponentiations. Since these exponentiations can be speeded up using the Chinese Remainder Theorem, the gap (for server computation overhead) between JL10 and DT10-2 is only double. Summing up server and client computation overhead, DT10-2 results to be the most efficient. In terms of bandwidth consumption, DT10-2 and JL10 are almost the same, while JL09 is slightly more expensive.

APSI-DT without pre-distribution. The only protocol available in this context is DT10-APSI (as discussed in Appendix B.3). Figure 7-10 illustrates that client overhead is determined only by client set size, whereas, server overhead is determined by both client and server set sizes. Note that measurements obtained for APSI-DT naturally mirror those of DT10-1, as the former simply adds authorization of client inputs (by merging signatures into the protocol).

APSI-DT with pre-distribution. The only protocol we evaluate for APSI-DT with data pre-distribution is IBE-APSI (as discussed in Appendix B.4). Figure 7-8 shows that client overhead increases linearly with client set size and does not depend on server set size. Recall that, in IBE-APSI, the server needs to compute pairing operations for each item, independent of client input. Moreover, since these operations can be pre-computed, server-side overhead and bandwidth consumption are negligible, as shown in Figures 9-12.[4] During the pre-computation phase, the server needs to compute w pairing and exponentiations, which makes pre-computation relatively expensive. Thus, note that, If Server Unlinkability is desired, server would need to repeat, for every interaction, the operations otherwise performed only during pre-computation.

One party small set case. Finally, we compare online computation costs and show the trend with small client or server set size. Our goal is to address scenarios where one party only has a single input. Table 4 shows client and server overhead for different protocols

[4] In these figures, y-values for IBE-APSI are all 0 which is out of the scope of the y-axis.

Protocols	Online Computation Overhead (ms)					
	v=1, w=10,000		v=10,000, w=1		v=1, w=1	
	Client	Server	Client	Server	Client	Server
FNP04	1,556.3	19,450.4	12,627.1	65.1	1.2	2.3
DT10-1	0.4	22.7	3,140.8	1,376.6	0.3	0.1
DT10-2	0	0.3	52.6	2,787.7	0	0.3
JL09	7.6	3.3	77,622.6	32,373.4	7.6	3.2
JL10	1.1	0.2	11,270.9	1,415.7	1.1	0.2
IBE-APSI	1.4	0	14,142.3	0	1.4	0
DT10-APSI	1.9	26.8	18,646.5	9,162.3	2.2	2.1

Table 4: Online computation overhead (in ms)

where either party's input is a singleton. We observe that the result mirror those showed by Figures 7-10.

Take Away. Based on our extensive experimental results, we conclude that DT10-1 is best-suited for PSI-DT without pre-distribution, and DT10-2 for PSI-DT with pre-distribution. DT10-APSI is the choice for APSI-DT without pre-distribution, whereas, IBE-APSI – for APSI-DT with pre-distribution.