Partitioned Data Security on Outsourced Sensitive and Non-sensitive Data

Sharad Mehrotra,[1] Shantanu Sharma,[1] Jeffrey D. Ullman,[2] and Anurag Mishra*[1]
[1]University of California, Irvine, USA. [2]Stanford University, USA.
sharad@ics.uci.edu, shantanu.sharma@uci.edu, ullman@gmail.com

ABSTRACT

Despite extensive research on cryptography, secure and efficient query processing over outsourced data remains an open challenge. This paper continues along the emerging trend in secure data processing that recognizes that the entire dataset may not be sensitive, and hence, non-sensitivity of data can be exploited to overcome limitations of existing encryption-based approaches. Taking the cue from recent papers on hybrid clouds, a direction to secure query processing that exploits database techniques is explored. We propose a new secure approach, entitled query binning (QB) that allows non-sensitive parts of the data to be outsourced in cleartext while guaranteeing that no information is leaked by the joint processing of non-sensitive data (in clear-text) and sensitive data (in encrypted form). Further, we show how QB can be extended to perform join and range queries. Interestingly, in addition to improve performance, we show that QB actually strengthens the security of the underlying cryptographic technique by preventing size, frequency-count, and workload-skew attacks.

1. INTRODUCTION

The last two decades have witnessed the development of secure and privacy-preserving encryption-based [6, 13, 30, 36, 64, 21, 16, 32, 60, 38, 52, 9, 19, 39] or secret-sharing-based [62, 31, 14, 44, 23, 69, 28, 49] techniques to realize the database as a service model [36]. Despite significant progress, a cryptographic approach that is both *secure* (*i.e.*, no leakage of sensitive data to the adversary) and *efficient* (in terms of time) simultaneously has proved to be very challenging. Broadly, work on cryptography to support secure outsourcing has taken the following directions:

1. *Techniques that support strong security guarantees*. The leading example of which is fully homomorphic encryption [30], which when mixed with oblivious-RAM (ORAM) [32], offers possibly amongst the most secure mechanisms. However, such mechanisms incur high overhead in terms of computation time.
2. *Techniques that do not depend on the data encryption* but provide strong security, especially, information-theoretic security, by distributing a value in the form of the secret-shares to non-colluding clouds. Shamir's secret-sharing [62], distributed point functions [31], function secret-sharing [14], and accumulating-automata [23] are a few examples of such techniques. Such methods often limit the type of operations one can perform while imposing high overhead in terms of communication.
3. *Techniques that try to support a wide range of operations* including index-based retrieval or joins, such as CryptDB [59], Arx [58], searchable encryption [64], and cryptographic indexes [27, 63, 39]. Such techniques often trade security for performance; for instance, techniques that depend on deterministic and order-preserving encryptions [6], traversal of the index by the cloud, or leakage of the searching token do not offer strong security. Papers [53, 41] show that order-preserving and deterministic encryption techniques when used together, on a dataset in which the entropy of the values is not high enough can leak the entire data in clear-text to an attacker through frequency analysis on the encrypted data.
4. *Techniques/systems that exploit secure hardware* (Intel Software Guard Extensions (SGX) [18]), *e.g.*, M2R [22], VC3 [61], and Opaque [77]. Such techniques also leak information during a query execution [54] and are inefficient (in terms of time) due to a limited available memory of the secure hardware.

Techniques	Time	Preventing attacks		
		Size	Workload-Skew	Access-patterns
Deterministic encryption	$1.43x$			
Non-deterministic encryption [58]	$2.1x$			
DSSE [39]	$3281x$			✓
SGX [18]	$6724x$			✓
Full-retrieval	$11135x$	✓	✓	✓
Homomorphic Encryption with ORAM	$> 11135x$			✓

Table 1: Comparing different cryptographic techniques in terms of time (for selection queries over TPC-H data) and attacks. §2 provides details of attacks. x is the time to search a predicate in cleartext. ✓indicates a technique is not vulnerable to a given attack.

Table 1 summarizes different techniques based on efficiency and security. Note that none of the above-mentioned techniques are completely secure against each attack mentioned in the table, except the full retrieval of the database from the public cloud to the trusted private side.

Given the state of the research, this paper explores a radically different approach to secure outsourcing that scales cryptographic mechanisms using database techniques while providing strong security guarantees. Our work is motivated by recent works on the hybrid cloud that has exploited the fact that for a large class of application contexts, data can be partitioned into sensitive and non-sensitive components. Such a classification, which is a common practice in industries for secure computing [1, 2] and done via appropriately using existing techniques [29, 25], was exploited to

*Anurag Mishra contributed only in implementing the proposed algorithm.

This material is based on research sponsored by DARPA under agreement number FA8750-16-2-0021. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

build hybrid cloud solutions [43, 76, 75, 56, 55]. These solutions outsource only non-sensitive data and enjoy both the benefits of the public cloud as well as strong security guarantees (without revealing sensitive data to an adversary). While these techniques provide an effective and secure solution, they are, however, based on a hybrid cloud, requiring data owners to maintain potentially unbounded storage locally and also suffer from significant inter-cloud communication overheads.

Our goal, in this paper, is to explore how sensitive and non-sensitive classification can be exploited by secure data processing techniques that store data in the public cloud to bring new efficiencies to secure data processing. In particular, in the envisioned model, data is stored in a partitioned way – sensitive data is secured using any existing cryptographic technique and non-sensitive data resides in plaintext. Query processing is also split into encrypted and plaintext query processing. We refer to this as *partitioned computing*. Unlike the case of the hybrid cloud, when implementing partitioned computing in the public cloud, data processing performed on the sensitive and non-sensitive parts of the data may reveal exact encrypted tuples (depending on an underlying cryptographic technique) and cleartext tuples that satisfy the query to the adversary. Consequently, this leads to inferences about sensitive data, which will be explained in detail in §3.

We first define a security model that formally states what it means to be secure in partitioned computing. We then develop a query binning (QB) approach that realizes secure partitioned computing for selection queries. We focus on selection queries for several reasons. First, selection queries are important in their own right. For instance, several key-value stores (*e.g.*, Amazon Dynamo) and document stores (*e.g.*, MongoDB) focus exclusively on selection queries (with limited support for joins). Furthermore, most cryptographic research has also focused on selection queries [13, 30, 36, 64, 21, 16, 32, 60, 38, 52, 31]. Since our goal is to speed up existing cryptographic techniques (and not to extend their functionality and make them resilient against attacks, such as order-revealing, inferences from deterministic encryptions, leakages from SGX, and different side-channel attacks [40, 53, 54, 15, 41, 48, 34, 12, 35]), we focus on selection queries. Nonetheless, there are recent work on cryptographic joins [57] and also on joins using SGX [77]. While these approaches are not yet practical (from an efficiency perspective) or leak information (*e.g.*, cache attacks [34], page table based attacks [71], and output size attacks [54]), we do extend QB to implement joins and range queries alongside such approaches.

We show two interesting effects of using QB:

1. By avoiding cryptographic processing on non-sensitive data, *the joint cost of communication and computation of QB is significantly less than the computation cost of a strongly secure cryptographic technique*[1] (*e.g.*, homomorphic encryptions, DPF [31], or secret-sharing-based technique [28, 24]) on the entire encrypted data; and hence, QB improves the performance of strong cryptographic techniques over a large-scale dataset (§6.2.1, §7).
2. *QB provides an enhanced security by preventing several attacks such as output size, frequency-count, and workload-skew attacks, even when the underlying cryptographic technique is susceptible to such attacks* (§6.2.2).

[1] As will be clear, QB trades off increased communication costs for executing queries, but can very significantly reduce cryptographic operations. This tradeoff significantly improves performance, especially, when using cryptographic mechanisms such as fully homomorphic encryption that take several seconds simply to compute a single operation [51], secret-sharing-based techniques that take a few seconds [28], or techniques such as bilinear maps that take over 1.5 hours to perform joins on a dataset of size less than 10MB [57]. When considering such cryptography, increased communication overheads are fully compensated by the savings. A similar observation, albeit in a very different context was also observed in [56] in the context of MapReduce jobs where overshuffling to prevent adversary to infer sensitive keys in the context of hybrid cloud was shown to be significantly better compared to private side operation.

Contribution. The primary contributions of this paper are listed below:

1. A formal definition of *partitioned data security* in the presence of a joint processing of sensitive and non-sensitive data (§4).
2. An efficient QB approach (§5) that guarantees partitioned security (Theorems 1 and 2), supporting insert operation (§A), cloud-side-indexes (§6), and that can be built on top of any cryptographic technique.
3. An analytical formal model to compare QB with a pure cryptographic technique under different conditions and different security levels such as preventing size, frequency-count, and workload-skew attacks (§6).
4. C-Ind-based QB approach that is both efficient and secure (§6.2.2).
5. Experiments to validate an integration of QB with existing cryptographic techniques on top of secure databases (§7).[2]
6. Extensions of QB to deal with workload-skew attacks (§A.1), non-identical searchable attribute-based column-level sensitivity (§A.2), other operators (such as join (§A.3), range (§A.4), and conjunctive queries (§A.6)) and dynamic data (insert, delete, and update) §A.5.

2. RELATED WORK ON SECURE SELECTION QUERIES

Broadly, existing research on secure selection query execution techniques can be classified into four categories:

- **Encryption-based techniques** examples of which include order-preserving encryption [6], deterministic encryption [13], homomorphic encryption [30], bucketization [36], searchable encryption [64, 21], private information retrieval (PIR) [16], practical-PIR [70], oblivious-RAM (ORAM) [32], oblivious transfers [60, 38], oblivious polynomial evaluation [52], oblivious query processing [9], searchable symmetric encryption [19], and distributed searchable symmetric encryption (DSSE) [39].
- **Secret-sharing [62] based techniques** that include distributed point function [31], function secret sharing [14], functional secret sharing [44], accumulating-automata [23], Splinter [69], and others [28, 49].
- **Trusted-hardware-based techniques** that include [8, 7, 10, 61, 22, 67, 77].
- **Sensitivity-based techniques.** The papers [43, 76, 75, 56] have explored secure MapReduce (MR) system implementations while [55] have explored secure SQL data processing. Both the secure MR and secure SQL execution solutions work on the principle of sensitivity-based data partitioning over the hybrid cloud. In [77], secure hardware, specifically Intel SGX [18], at the cloud is used to partition the computation. [17] deals with column-level sensitivity by encrypting only sensitive columns; however, they

[2] QB's performance is evaluated on two well-known systems A and B; due to legal restrictions, the real names of the systems A and B are omitted.

Table 2: Comparison of different algorithms with our algorithms.

Algorithms	Storage		Computational cost for search		Insert cost	# rounds	Size attack	Workload-skew and frequency attacks	Data sensitivity
	Cloud	DB Owner	Cloud	DB Owner					
Encryption-based *indexable* solutions									
CryptDB* [59]	D	0	$\mathcal{O}(1)$	$\mathcal{O}(r)$	$\mathcal{O}(1)$	1	Y	Y	N
Searchable encryption [21]	$2D$ or $4D$	0	$D^{1/3}\log D$ or $D^{1/5}\log D$	$\mathcal{O}(r)$	NA	1	Y	Y	N
Arx* [58]	D	V_D	$\mathcal{O}(r \cdot \log D)$	$\mathcal{O}(r)$	$\mathcal{O}(1)$	1	Y	Y	N
Hybrid-Secure** [55]	D	αD	$\mathcal{O}(r + \log((1-\alpha)D))$	$\mathcal{O}(\log(\alpha D) + r)$	$\mathcal{O}(1)$	1	N	N	Y
Our solution with indexes**	**D**	$\mathbf{V_D}$	$\mathcal{O}(\sqrt{\lvert NS\rvert} r \log(\alpha D)$+ $\sqrt{\lvert NS\rvert}(r + \log(1-\alpha)D)$	$\mathcal{O}(r\sqrt{\lvert S\rvert})$	$\mathcal{O}(1)$	**1**	**N**	**N**	**Y**
Encryption-based *non-indexable* solutions									
Searchable Encryption [64]	D	0	D	$\mathcal{O}(1)$	$\mathcal{O}(1)$	1	Y	Y	Y
Our solution without indexes**	**D**	$\mathbf{V_D}$	$\mathcal{O}(\alpha D$+ $\sqrt{\lvert NS\rvert}(r + \log(1-\alpha)D)$	$\mathcal{O}(r\sqrt{\lvert S\rvert})$	$\mathcal{O}(1)$	**1**	**N**	**N**	**Y**

Notations. $*$: Non-secure systems, due to dependence on deterministic and order-preserving encryption (CryptDB) and at the time of selection and join operations (Arx). $**$: fully secure systems. Y: Yes, a technique/system is vulnerable to a given attack. N: No. D: a database (containing only a single attribute, for simplicity). $V_D \ll D$: # unique values in the attribute that is equal to $|S| + |NS|$ in our case, where $|S|$: unique sensitive values, $|NS|$: unique non-sensitive values, and $|S| < |NS|$. r: # occurrences of a searching value. $\alpha < 1$: be a ratio between the sizes of sensitive data and the entire dataset.

are susceptible to reveal sensitive information based on background knowledge (an attack like presented in [50]), since the relation is not partitioned into two parts based on sensitivity. However, QB can also prevent such a type of attack by partitioning a relation.

Each of the above strategies has resulted in corresponding systems that support secure data processing. For instance, CryptDB [59], Monomi [68], TrustedDB [11], CorrectDB [10], SDB [72], ZeroDB [26], L-EncDB [45], MrCrypt [66], Crypsis [65], Arx [58], and Opaque [77] are some novel encryption-based systems. Likewise, Cypherbase [7], Microsoft Always Encrypted, Oracle 12c, Amazon Aurora [3], and MariaDB [4] are industrial secure encrypted databases. DSSE-based SDB [5] is a secret-sharing and encryption-based system while Arx [58] and Opaque [77] work on the data sensitivity principle.

Moreover, these systems/techniques are unable to prevent one or more of the following attacks: (*i*) size attack, *i.e.*, an adversary having some background knowledge can deduce the full/partial outputs by simply observing the output sizes [77]; (*ii*) frequency attack, *i.e.*, an adversary can deduce how many tuples have an identical value [53]; (*iii*) workload-skew attack, *i.e.*, an adversary, having the knowledge of frequent selection queries by observing many queries, can estimate which encrypted tuples potentially satisfy the frequent section selection queries; (*iv*) access-pattern attack, *i.e.*, addresses of encrypted tuples that satisfy the query [16]. Note that computationally expensive and access-pattern-hiding cryptographic techniques (*e.g.*, PIR, ORAM, DSSE, oblivious transfer, and secret-sharing) can prevent the size, frequency-count, and workload-skew attacks *only* on *non-skewed and non-deterministically encrypted* datasets. In contrast, to the best of our knowledge, there is no cryptographic technique that prevents all the four attacks on a *skewed dataset*. It is important to mention that QB mixed with a weak cryptographic technique [63, 27, 58] is efficient and secure against size and workload-skew attacks. This fact will be clear by performance analysis in §7 (Figure 8f).

Table 2 presents a comparison of different techniques. Note that both QB and the most efficient but insecure indexable technique (Arx [58]) store an identical amount of metadata at the DB owner. However, QB makes Arx completely secure against the size, frequency-count, and workload-skew attacks. However, the computational cost of QB is $\sqrt{|NS|}$ times higher than Arx, because of searching $\sqrt{|NS|}$ more predicates on encrypted data. Nevertheless, QB performs better than a recent non-indexable technique (for example, searchable encryption [31]) in terms of the number of rounds to retrieve all the occurrences of a predicate, time, and security levels. Table 2, also, clarifies that QB is better than a recent searchable encryption [21] by allowing dynamic operations, *e.g.*, insert, and stronger security guarantees.

Notations	Meaning
$\lvert S\rvert$	Number of sensitive data values
$\lvert NS\rvert$	Number of non-sensitive data values
R_s	Sensitive parts of a relation R
R_{ns}	Non-sensitive parts of a relation R
s_i and ns_j	i^{th} sensitive and j^{th} non-sensitive values
SB	The number of sensitive bins
SB_i	i^{th} sensitive bin
$\lvert SB\rvert = y$	Sensitive values in a sensitive bin or the size of a sensitive bin
NSB	The number of non-sensitive bins
NSB_i	i^{th} non-sensitive bin
$\lvert NSB\rvert = x$	Non-sensitive values in a non-sensitive bin or the size of a non-sensitive bin
$q(w)$	A query, q, for a predicate w
$q(W_{ns})(R_{ns})$	A query, q, for a set, W_{ns}, of predicates in clear-text over R_{ns}
$q(W_s)(R_s)$	A query, q, for a set, W_s, of predicates in encrypted form over R_s
$q(W)(R_s, R_{ns})[A]$	A query, q, for a set, W, of values, searching on the attribute, A, of the relations R_s and R_{ns}, where $W = W_s \cup W_{ns}$
$E(t_i)$	i^{th} encrypted tuple

Table 3: Notations used in the paper.

3. PARTITIONED COMPUTATION

In this section, we first define more precisely what we mean by partitioned computing, illustrate how such a computation can leak information due to the joint processing of sensitive and non-sensitive data, discuss the corresponding security definition, and finally discuss system and adversarial models under which we will develop our solutions. Table 3 enlists notations used in this paper.

The Partitioned Computation Model

We assume the following two entities in our model:

1. *A trusted database (DB) owner* who divides a relation R having attributes, say $A_1, A_2, \ldots, A_n$, into the following two relations

based on row-level data sensitivity: R_s and R_{ns} containing all sensitive and non-sensitive tuples, respectively.[3] The DB owner outsources the relation R_{ns} to a public cloud. The tuples of the relation R_s are encrypted using any existing non-deterministic encryption [33] mechanism before outsourcing to the same public cloud.

In our setting, the DB owner has to store metadata such as searchable values and their frequency counts, which will be used for appropriate query formulation. The DB owner is assumed to have sufficient storage for such metadata, and also computational capabilities to perform encryption and decryption. The size of metadata is smaller than the size of the original data.

2. *The untrusted public cloud* that stores the databases, executes queries, and provides answers.

Let us consider a query q over the relation R, denoted by $q(R)$. A partitioned computation strategy splits the execution of q into two independent subqueries: $q(R_s)$: a query to be executed on the encrypted sensitive relation R_s, and $q(R_{ns})$: a query to be executed on the non-sensitive relation R_{ns}. The final result is computed (using a query q_{merge}) by appropriately merging the results of the two subqueries at the DB owner side. In particular, the query q on a relation R is partitioned, as follows:

$$q(R) = q_{merge}\Big(q(R_s), q(R_{ns})\Big)$$

Let us illustrate partitioned computations through an example.

	EId	FirstName	LastName	SSN	Office#	Department
t_1	E101	Adam	Smith	111	1	Defense
t_2	E259	John	Williams	222	2	Design
t_3	E199	Eve	Smith	333	2	Design
t_4	E259	John	Williams	222	6	Defense
t_5	E152	Clark	Cook	444	1	Defense
t_6	E254	David	Watts	555	4	Design
t_7	E159	Lisa	Ross	666	2	Defense
t_8	E152	Clark	Cook	444	3	Design

Figure 1: A relation: *Employee*.

Example 1. Consider an *Employee* relation, see Figure 1. In this relation, the attribute *SSN* is sensitive, and furthermore, all tuples of employees for the *Department* = "`Defense`" are sensitive. In such a case, the *Employee* relation may be stored as the following three relations: (*i*) *Employee1* with attributes *EId* and *SSN* (see Figure 2a); (*ii*) *Employee2* with attributes *EId*, *FirstName*, *LastName*, *Office#*, and *Department*, where *Department* = "`Defense`" (see Figure 2b); and (*iii*) *Employee3* with attributes *EId*, *FirstName*, *LastName*, *Office#*, and *Department*, where *Department* <> "`Defense`" (see Figure 2c). Since the relations *Employee1* and *Employee2* (Figures 2a and 2b) contain only sensitive data, these two relations are encrypted before outsourcing, while *Employee3* (Figure 2c), which contains only non-sensitive data, is outsourced in clear-text. We assume that the sensitive data is strongly encrypted such that the property of *ciphertext indistinguishability* (*i.e.*, an adversary cannot distinguish pairs of ciphertexts) is achieved. Thus, the two occurrences of `E152` have two different ciphertexts.

Consider a query `q: SELECT FirstName, LastName, Office#, Department from Employee where FirstName = John`. In the partitioned computation, the query `q` is partitioned into two subqueries: q_s that executes on `Employee2`, and q_{ns} that executes on `Employee3`. q_s will retrieve the tuple t_4 while q_{ns} will retrieve the tuple t_2. q_{merge} in this example is simply a union operator. Note that the execution of the query `q` will also retrieve the same tuples.

[3]QB can also deal with column-level sensitivity, where sensitive and non-sensitive relations have different attributes; see §A.2.

	EId	SSN
t_1	E101	111
t_2	E259	222
t_3	E199	333
t_5	E152	444
t_6	E254	555
t_7	E159	666

(a) A sensitive relation: *Employee1*.

	EId	FirstName	LastName	Office#	Department
t_1	E101	Adam	Smith	1	Defense
t_4	E259	John	Williams	6	Defense
t_5	E152	Clark	Cook	1	Defense
t_7	E159	Lisa	Ross	2	Defense

(b) A sensitive relation: *Employee2*.

	EId	FirstName	LastName	Office#	Department
t_2	E259	John	Williams	2	Design
t_3	E199	Eve	Smith	2	Design
t_6	E254	David	Watts	4	Design
t_8	E152	Clark	Cook	3	Design

(c) A non-sensitive relation: *Employee3*.

Figure 2: Three relations obtained from *Employee* relation.

However, such a partitioned computation, if performed naively, leads to inferences about sensitive data from non-sensitive data. Before discussing inference attacks, we first present the adversarial model.

Adversarial Model

We assume an honest-but-curious adversary, which is considered in the standard setting for security in the public cloud [42, 73, 74] that is *not trustworthy*. An honest-but-curious adversarial public cloud stores an outsourced dataset without tampering, correctly computes assigned tasks, and returns answers; however, it may exploit side knowledge (*e.g.*, query execution, background knowledge, and the output size) to gain as much information as possible about the sensitive data.[4] Furthermore, the honest-but-curious adversary can eavesdrop on the communication channels between the cloud and the DB owner, and that may help in gaining knowledge about sensitive data, queries, or results; hence, a secure channel is assumed. In our setting, the adversary has full access to the following:

1. All the non-sensitive data. For example, for the Employee relation in Example 1, an adversary knows the complete *Employee3* relation (refer to Figure 2c).
2. *Auxiliary* information of the sensitive data. The auxiliary information may contain metadata, schema of the relation, and the number of tuples in the relation. In Example 1, the adversary knows that there are two sensitive relations, one of them containing six tuples and the other one containing four tuples, in the *Employee1* and the *Employee2* relations; refer to Example 1 (Figures 2a and 2b). In contrast, the adversary is not aware of the following information before the query execution: how many people work in a specific sensitive department, is a specific person working only in a sensitive department, only in a non-sensitive department, or both.
3. Adversarial view. When executing a query, an adversary knows which encrypted sensitive tuples and cleartext non-sensitive tuples are sent in response to a query. We refer this as the adversarial view. For example, the first row of Table 4 shows an adversarial view that shows that t_2 tuples from the non-sensitive relation and encrypted t_4 tuples from the sensitive relation are returned to answer the query for `E259`.

[4]The honest-but-curious adversary cannot launch any attack against the DB owner. We do not consider cyber-attacks that can exfiltrate data from the DB owner directly, since defending against generic cyber-attacks is outside the scope of this paper.

4. Some frequent query values. The adversary observes query predicates on the non-sensitive data, and hence, can deduce the most frequent query predicates by observing many queries.

Inference Attacks in Partitioned Computations

To see the inference attack on the sensitive data while jointly processing sensitive and non-sensitive data, consider following three queries on the *Employee2* and *Employee3* relations; refer to Figures 2b and 2c.

Example 2. (*i*) retrieve tuples corresponding to employee `E259`, (*ii*) retrieve tuples corresponding to employee `E101`, and (*iii*) retrieve tuples corresponding to employee `E199`.[5] When answering a query, the adversary knows the tuple ids of retrieved encrypted tuples and the full information of the returned non-sensitive tuples. We refer to this information gain by the adversary as the *adversarial view*, shown in Table 4, where $E(t_i)$ denotes an encrypted tuple t_i.

Query value	Returned tuples/Adversarial view	
	Employee2	Employee3
E259	$E(t_4)$	t_2
E101	$E(t_1)$	null
E199	null	t_3

Table 4: Queries and returned tuples/adversarial view.

Outputs of the above three queries will reveal enough information to learn something about sensitive data. In the first query, the adversary learns that `E259` works in both sensitive and non-sensitive departments, because the answers obtained from the two relations contribute to the final answer. Moreover, the adversary may learn which sensitive tuple has an *Eid* equals to `E259`. In the second query, the adversary learns that `E101` works only in a sensitive department, because the query will not return any answer from the Employee3 relation. In the third query, the adversary learns that `E199` works only in a non-sensitive department.

The Query Binning (QB) Approach

In order to prevent the inference attack in the partitioned computation, we need a new security definition. Before we discuss the formal definition of partitioned data security (§4), we first provide a possible solution to prevent inference attacks and then intuition for the security definition.

The query binning (QB) strategy stores a non-sensitive relation, say R_{ns}, in clear-text while it stores a sensitive relation, say R_s, using a cryptographically secure approach. QB prevents leakage such as in Example 2 by appropriately mapping a query for a predicate, say $q(w)$, to corresponding queries both over the non-sensitive relation, say $q(W_{ns})(R_{ns})$, and encrypted relation, say $q(W_s)(R_s)$. The queries $q(W_{ns})(R_{ns})$ and $q(W_s)(R_s)$, each represents a set of predicates (or selection queries) that are executed over the relation R_{ns} in plaintext and, respectively, over the sensitive relation R_s, using the underlying cryptographic method. The set of predicates in $q(W_{ns})(R_{ns})$ (likewise in $q(W_S)(R_s)$) correspond to the non-sensitive (sensitive) *bins* including the predicate w, denoted by NSB (SB). The predicates in $q(W_s)(R_s)$ are encrypted before transmitting to the cloud.

The bins are selected such that: (*i*) $w \in q(W_{ns})(R_{ns}) \cap q(W_s)(R_s)$ to ensure that all the tuples containing the predicate w are retrieved, and, (*ii*) joint execution of the queries $q(W_{ns})(R_{ns})$ and $q(W_s)(R_s)$ (hereafter, denoted by $q(W)(R_s, R_{ns})$, where $W = W_s \cup W_{ns}$) does not leak the predicate w. Results from the execution of the queries $q(W_{ns})(R_{ns})$ and $q(W_s)(R_s)$ are decrypted, possibly filtered, and merged to generate the final answer. Note that *bins are created only once for all the values of a searching attribute before any query is executed*. The details of the bin formation will be discussed in §5.

For answering the above-mentioned three queries, QB creates two bins on sensitive parts: {`E101`, `E259`}, {`E152`, `E159`}, and two sets on non-sensitive parts: {`E259`, `E254`}, {`E199`, `E152`}. Table 5 illustrates the generated adversarial view when QB is used to answer queries as shown in Example 2. In this example, row 1 of Table 5 shows that this instance of QB maps the query for `E259` to ⟨`E259`, `E254`⟩ over cleartext and to encrypted version of values for ⟨`E259`, `E101`⟩ over sensitive data. Note that simply from the generated adversarial views, the adversary cannot determine the query value w (`E259` in the example) or find a value that is shared between the two sets. Thus, while answering a query, the adversary cannot learn which employee works only in defense, design, or in both.

The reason is that the desired query value, w, is encrypted with other encrypted values of W_s, and, furthermore, the query value, w, cannot be distinguished from many requested non-sensitive values of W_{ns}, which are in clear-text. Consequently, *the adversary is unable to find an intersection of the two sets, which is the exact value*.[6]

Query value	Returned tuples/Adversarial view	
	Employee2	Employee3
E259	$E(t_4)$, $E(t_1)$	t_2, t_6
E101	$E(t_4)$, $E(t_1)$	t_3, t_8
E199	$E(t_4)$, $E(t_1)$	t_3, t_8

Table 5: Queries and returned tuples/adversarial view, following QB.

Thus, in a joint processing of sensitive and non-sensitive data, *the goal of the adversary is to find as much sensitive information as possible (using the adversarial view or background knowledge)*, and *the goal of a secure technique is to prevent information leakage through the joint processing of non-sensitive and sensitive data*.

4. PARTITIONED DATA SECURITY

In this section, we formalize the notion of *partitioned data security* that establishes when a partitioned computation over sensitive and non-sensitive data does not leak any sensitive information. Note that an adversary may seek to infer sensitive information using the adversarial view created during query processing, knowledge of output size, frequency counts, and workload characteristics. We begin by first formalizing the concepts of: *associated values*, *associated tuples*, and *relationship between counts of sensitive values*.[7]

[5]We used random *Eids*, which is also common in a real employee relation. In contrast, in sequential ids, the absence of an id from the non-sensitive relation directly informs the adversary that the given id exists in the sensitive relation.

[6]For hiding an exact selection predicate over an encrypted relation regardless of data sensitivity, an approach to create a set of selection predicates including the exact predicate is presented in [48], which, however, cannot be used to search over sensitive and non-sensitive relations or multiple relations, due to not dealing with inference attacks.

[7]To develop the notation, defining security, and developing QB (§5), we assume that search is performed on a specific attribute, A, over a relation, R. The approach trivially generalizes when several attributes are searchable – we need to maintain metadata required for QB not just for A, but for all searchable attributes in R.

Notations used in the definitions. Let $t_1, t_2, \ldots, t_m$ be tuples of a sensitive relation, say R_s. Thus, the relation R_s stores the encrypted tuples $E(t_1), E(t_2), \ldots, E(t_m)$. Let $s_1, s_2, \ldots, s_{m'}$ be values of an attribute, say A, that appears in one of the sensitive tuples of R_s. Note that $m' \leq m$, since several tuples may have an identical value. Furthermore, $s_i \in Domain(A)$, $i = 1, 2, \ldots, m'$, where $Domain(A)$ represents the domain of values the attribute A can take. By $\#_s(s_i)$, we refer to the number of sensitive tuples that have s_i as the value for attribute A. We further define $\#_s(v) = 0, \forall v \in Domain(A), v \notin s_1, s_2, \ldots, s_{m'}$. Let $t_1, t_2, \ldots, t_n$ be tuples of a non-sensitive relation, say R_{ns}. Let $ns_1, ns_2, \ldots, ns_{n'}$ be values of the attribute A that appears in one of the non-sensitive tuples of R_{ns}. In analogy with the case where the relation is sensitive, $n' \leq n$, and $ns_i \in Domain(A)$, $i = 1, 2, \ldots, n'$.

Associated values. Let $e_i = E(t_i)[A]$ be the encrypted representation of an attribute value of A in a sensitive tuple of the relation R_s, and ns_j be a value of the attribute A for some tuple of the relation R_{ns}. We say that e_i is *associated* with ns_j, (denoted by $\stackrel{a}{=}$), if the plaintext value of e_i is identical to the value ns_j. In Example 1, the value of the attribute `Eid` in tuple t_4 (of *Employee2*, see Figure 2b) is associated with the value of the attribute `Eid` in tuple t_2 (of *Employee3*, see Figure 2c), since both values correspond to `E259`.

Associated tuples. Let t_i be a sensitive tuple of the relation R_s (*i.e.*, R_s stores encrypted representation of t_i) and t_j be a non-sensitive tuple of the relation R_{ns}. We state that t_i is associated with t_j (for an attribute, say A) iff the value of the attribute A in t_i is associated with the value of the attribute A in t_j (*i.e.*, $t_i[A] \stackrel{a}{=} t_j[A]$). Note that this is the same as stating that the two values of attribute A are equal for both tuples.

Relationship between counts of sensitive values. Let v_i and v_j be two distinct values in $Domain(A)$. We denote the relationship between the counts of sensitive tuples with these A values (*i.e.*, $\#_s(v_i)$ (or $\#_s(v_j)$)) by $v_i \stackrel{r}{\sim} v_j$. Note that $\stackrel{r}{\sim}$ can be one of $<, =,$ or $>$ relationships. For instance, in Example 1, the `E101` $\stackrel{r}{\sim}$ `E259` corresponds to $=$, since both values have exactly one sensitive tuple (see Figure 2b), while `E101` $\stackrel{r}{\sim}$ `E199` is $>$, since there is one sensitive tuple with value `E101` while there is no sensitive tuple with `E199`.

Given the above definitions, we can now formally state the security requirement that ensures that simultaneous execution of queries over sensitive (encrypted) and non-sensitive (plaintext) data does not leak any information.

Definition: Partitioned Data Security. Let R be a relation containing sensitive and non-sensitive tuples. Let R_s and R_{ns} be the sensitive and non-sensitive relations, respectively. Let AV be an adversarial view generated for a query $q(w)(R_s, R_{ns})[A]$, where the query, q, for a value w in the attribute A of the R_s and R_{ns} relations. Let X be the auxiliary information about the sensitive data, and Pr_{Adv} be the probability of the adversary knowing any information. A query execution mechanism ensures the partitioned data security if the following two properties hold:

1. $Pr_{Adv}[e_i \stackrel{a}{=} ns_j|X] = Pr_{Adv}[e_i \stackrel{a}{=} ns_j|X, AV]$, where $e_i = E(t_i)[A]$ is the encrypted representation for the attribute value A for any tuple t_i of the relation R_s and ns_j is a value for the attribute A for any tuple of the relation R_{ns}.

2. $Pr_{Adv}[v_i \stackrel{r}{\sim} v_j|X] = Pr_{Adv}[v_i \stackrel{r}{\sim} v_j|X, AV]$, for all $v_i, v_i \in Domain(A)$.

The first equation (1) captures the fact that an initial probability of *associating* a sensitive tuple with a non-sensitive tuple will be identical after executing a query on the relations. Thus, an adversary cannot learn anything from an adversarial view generated after the query execution. The second equation (2) states that the probability of an adversary gaining information about the relative frequency of sensitive values does not increase after the query execution. In Example 2, an execution of any three queries (for values `E101`, `E199`, or `E259`) without using QB does not satisfy the above first equation. For example, the query for `E199` retrieves the only tuple from non-sensitive relation, and that changes the probability of estimating whether `E199` is sensitive or non-sensitive to 0 as compared to an initial probability of the same estimation, which was 1/4. Hence, an execution of the three queries violates partitioned data security. However, the query execution for `E259` and `E101` satisfies the second equation, since the count of returned tuples from *Employee2* is equal. Hence, the adversary cannot distinguish between the count of the values (`E259` and `E101`) in the domain of `Eid` of *Employee2* relation.

5. QUERY BINNING TECHNIQUE

We develop our strategy initially under the assumption that queries are only on a single attribute, say A. QB approach takes as inputs: (*i*) the set of data values (of the attribute A) that are sensitive, along with their counts, and (*ii*) the set of data values (of the attribute A) that are non-sensitive, along with their counts. QB returns a partition of attribute values that form the query bins for both the sensitive as well as for the non-sensitive parts of the query. We begin in §5.1 by developing the approach for the case when a sensitive tuple is associated with at most one non-sensitive tuple (Algorithm 1). We then (§5.2) develop a simple extension of Algorithm 1 to deal with a situation where the number of non-sensitive (or sensitive) values is close to a square number.[8] Finally, we provide a general strategy to create bins when a sensitive tuple is associated with several non-sensitive tuples, in §5.3.

Informally, QB distributes attribute values in a matrix, where rows are sensitive bins, and columns are non-sensitive bins. For example, suppose there are 16 values, say $0, 1, \ldots, 15$, and assume all the values have sensitive and associated non-sensitive tuples. Now, the DB owner arranges 16 values in a 4×4 matrix, as follows:

	NSB_0	NSB_1	NSB_2	NSB_3
SB_0	11	2	5	14
SB_1	10	3	8	7
SB_2	0	15	6	4
SB_3	13	1	12	9

In this example, we have four sensitive bins: SB_0 {11,2,5,14}, SB_1 {10,3,8,7}, SB_2 {0,15,6,4}, SB_3 {13,1,12,9}, and four non-sensitive bins: NSB_0 {11,10,0,13}, NSB_1 {2,3,15,1}, NSB_2 {5,8,6,12}, NSB_3 {14,7,4,9}. When a query arrives for a value, say 1, the DB owner searches for the tuples containing values 2,3,15,1 (viz. NSB_1) on the non-sensitive data and values in SB_3 (viz., 13,1,12,9) on the sensitive data using the cryptographic mechanism integrated into QB. We will show that in the proposed approach, while the adversary learns that the query corresponds to one of the four values in NSB_1, since query values in SB_3 are encrypted, the adversary does not learn the actual sensitive value or

[8]The rationale for explaining the strategy first for any number of non-sensitive values and then explaining it to a case of non-sensitive values close/equal to a square number will become clear shortly.

Algorithm 1: Bin-creation algorithm, the base case.

Inputs: $|NS|$: the number of values in the non-sensitive data, $|S|$: the number of values in the sensitive data.
Outputs: SB: sensitive bins; NSB: non-sensitive bins

1 **Function** $create\_bins(S, NS)$ **begin**
2 Permute all sensitive values
3 $x, y \leftarrow approx\_sq\_factors(|NS|)$: $x \geq y$
4 $|NSB| \leftarrow x$, $NSB \leftarrow \lceil |NS|/x \rceil$, $SB \leftarrow x$, $|SB| \leftarrow y$
5 **for** $i \in (1, |S|)$ **do** $SB[i \text{ modulo } x][*] \leftarrow S[i]$;
6 **for** $(i, j) \in (0, SB-1), (0, |SB|-1)$ **do** $NSB[j][i] \leftarrow allocateNS(SB[i][j])$;
7 **for** $i \in (0, NSB-1)$ **do** $NSB[i][*] \leftarrow$ fill the bin if empty with the size limit to x ;
8 **return** SB and NSB
end
9 **Function** $allocateNS(SB[i][j])$ **begin**
find a non-sensitive value associated with the j^{th} sensitive value of the i^{th} sensitive bin
end

the actual non-sensitive value that is identical to a clear-text sensitive value.

5.1 The Base Case

QB consists of two steps. First, query bins are created (information about which will reside at the DB owner) using which queries will be rewritten. The second step consists of rewriting the query based on the binning.

Here, QB is explained for the base case, where a sensitive tuple, say t_s, is associated with at most a single non-sensitive tuple, say t_{ns}, and vice versa (*i.e.*, $\stackrel{a}{=}$ is a 1:1 relationship). Thus, if the value has two tuples, then one of them must be sensitive and the other one must be non-sensitive, but both the tuples cannot be sensitive or non-sensitive. A value can also have only one tuple, either sensitive or non-sensitive. Note that if $t_1, t_2, \ldots, t_l$ are sensitive tuples, with values of an attribute A being $s_1, s_2, \ldots s_n$, $s_i \neq s_j$ if $i \neq j$.

Thus, in the remainder of the section, we will refer to association between encrypted value $E(t_i)[A]$ and a non-sensitive value ns_j simply as an association between values s_i and ns_j, where s_i is the cleartext representation of $E(t_i)[A]$ and ns_j is a value in the attribute A of a non-sensitive relation. That is, $s_i \stackrel{a}{=} ns_j$ represents $E(t_i)[A] \stackrel{a}{=} ns_j$.

The scenario depicted in Example 1 satisfies the base case. The *EId* attribute values corresponding to sensitive tuples include $\langle$E101, E259, E152, E159$\rangle$ and corresponding to non-sensitive tuples are $\langle$E199, E259, E254, E152$\rangle$ for which $\stackrel{a}{=}$ is 1:1. We discuss QB under the above assumption, but relax the assumption in §5.3. Before describing QB, we first define the concept of *approximately square factors of a number*.

Approximately square factors. *We say two numbers, say x and y, are* approximately square factors of a number*, say $n > 0$, if $x \times y = n$, and x and y are equal or close to each other such that the difference between x and y is less than the difference between any two factors, say x' and y', of n such that $x' \times y' = n$.*

Step 1: Bin-creation. QB, described in Algorithm 1, finds two approximately square factors of $|NS|$, say x and y, where $x \geq y$. QB creates $SB = x$ sensitive bins, where each sensitive bin contains at most y values. Thus, we assume $|S| \geq x$. QB, further, creates $NSB = \lceil |NS|/x \rceil$ non-sensitive bins, where each non-sensitive bin contains at most $|NSB| = x$ values. Note that we are assuming that $|S| \leq |NS|$.[9]

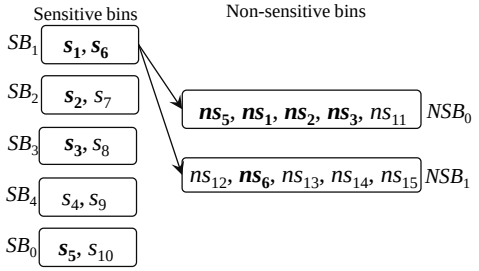


Figure 3: QB for 10 sensitive and 10 non-sensitive values.

Assignment of sensitive values. We number the sensitive bins from 0 to $x-1$ and the values therein from 0 to $y-1$. To assign a value to sensitive bins, QB first permutes the set of sensitive values. Such a permutation is kept secret from the adversary by the DB owner.[10] In order to assign sensitive values to sensitive bins, QB takes the i^{th} sensitive value and assigns it to the $(i \ modulo \ x)^{th}$ sensitive bin (see Lines 2 and 5 of Algorithm 1).

Assignment of non-sensitive values. We number the non-sensitive bins from 0 to $\lceil |NS| \rceil / x - 1$ and values therein from 0 to $x-1$. In order to assign non-sensitive values, QB takes a sensitive bin, say j, and its i^{th} sensitive value. Assign the non-sensitive value associated with the i^{th} sensitive value to the j^{th} position of the i^{th} non-sensitive bin. Here, if each value of a sensitive bin has an associated non-sensitive value and $|S| = |NS|$, then QB has assigned all the non-sensitive values to their bins (Line 6 of Algorithm 1). Note that it may be the case that only a few sensitive values have their associated non-sensitive values and $|S| \leq |NS|$. In this case, we assign the sensitive and their associated non-sensitive values to bins like we did in the previous case. However, we need to assign the non-sensitive values that are not associated with a sensitive value, by filling all the non-sensitive bins to size x (Line 7 of Algorithm 1).

Example 3: (Query binning example). We show the bin-creation algorithm for 10 sensitive values and 10 non-sensitive values. We assume that only five sensitive values, say s_1, s_2, s_3, s_5, s_6, have their associated non-sensitive values, say $ns_1, ns_2, ns_3, ns_5, ns_6$, and the remaining 5 sensitive (say, $s_4, s_7, s_8, \ldots s_{10}$) and 5 non-sensitive values (say, $ns_{11}, ns_{12}, \ldots, ns_{15}$) are not associated. For simplicity, we use different indexes for non-associated values.

QB creates 2 non-sensitive bins and 5 sensitive bins, and divides 10 sensitive values over the following 5 sensitive bins: SB_0 $\{s_5, s_{10}\}$, SB_1 $\{s_1, s_6\}$, SB_2 $\{s_2, s_7\}$, SB_3 $\{s_3, s_8\}$, SB_4 $\{s_4, s_9\}$; see Figure 3. Now, QB distributes non-sensitive values associated with the sensitive values over two non-sensitive bins, resulting in the bin NSB_0 $\{ns_5, ns_1, ns_2, ns_3, *\}$ and NSB_1 $\{*, ns_6, *, *, *\}$, where a $*$ shows an empty position in the bin. In the sequel, QB needs to fill the non-sensitive bins with the remaining 5 non-sensitive values; hence, ns_{11} is assigned to the last position of the bin NSB_0, and the bin NSB_1 contains the remaining 4 non-sensitive values such as $\{ns_{12}, ns_6, ns_{13}, ns_{14}, ns_{15}\}$.

Aside. Note that QB assigned at least as many values in a non-sensitive bin as it assigned to a sensitive bin. QB may form the non-sensitive and sensitive bins in such a way that the number of values

[9]QB can also handle the case of $|S| > |NS|$ by applying Algorithm 1 in a reverse way, *i.e.*, factorizing $|S|$.

[10]We emphasize to first permute sensitive values to prevent the adversary to create bins at her end; *e.g.*, if the adversary is aware of a fact that employee ids are ordered, then she can also create bins by knowing the number of resultant tuples to a query. However, for simplicity, we do not show permuted sensitive values in any figure.

Algorithm 2: Bin-retrieval algorithm.

Inputs: w: the query value.
Outputs: SB_a and NSB_b: one sensitive bin and one non-sensitive bin to be retrieved for answering w.
Variables: $found \leftarrow$ **false**

```
1 Function retrieve_bins(q(w)) begin
2     for (i, j) ∈ (0, SB − 1), (0, |SB| − 1) do
          if w = SB_i[j] then
3             return SB_i and NSB_j; found ← true; break
          end
      end
4     if found ≠ true then
5         for (i, j) ∈ (0, NSB − 1), (0, |NSB| − 1) do
6             if w = NSB_i[j] then
                  return NSB_i and SB_j; break
              end
          end
      end
7     Retrieve the desired tuples from the cloud by sending
        encrypted values of the bin SB_i (or SB_j) and clear-text
        values of the bin NSB_j (or NSB_i) to the cloud
  end
```

in sensitive bins is higher than the non-sensitive bins. We chose sensitive bins to be smaller since the processing time on encrypted data is expected to be higher than clear-text data processing; hence, by searching and retrieving fewer sensitive tuples, we decrease the encrypted data-processing time.

Step 2: Bin-retrieval – answering queries. Algorithm 2 presents the pseudocode for the bin-retrieval algorithm. The algorithm, first, checks the existence of a query value in sensitive bins and/or non-sensitive bins (see Lines 2 and 4 of Algorithm 2). If the value exists in a sensitive bin and a non-sensitive bin, the DB owner retrieves the corresponding two bins (see Line 7). Note that here the adversarial view is not enough to leak the query value or to find a value that is shared between the two bins. The reason is that the desired query value is encrypted with a set of other encrypted values and, furthermore, the query value is obscured in many requested non-sensitive values, which are in clear-text. Consequently, the adversary is unable to find an intersection of the two bins, which is the exact value.

There are the following three other cases to consider:

1. Some sensitive values of a bin are not associated with any non-sensitive value. For example, in Figure 3, the sensitive values s_4, s_7, s_8, s_9, and s_{10} are not associated with any non-sensitive value.
2. A sensitive bin does not hold any value that is associated with any non-sensitive value. For example, the sensitive bin SB_4 in Figure 3 satisfies this clause.
3. A non-sensitive bin containing no value that is associated with any sensitive value.

In all the three cases, if the DB owner retrieves only either a sensitive or non-sensitive bin containing the value, then it will lead to information leakage similar to Example 2. In order to prevent such leakage, Algorithm 2 follows two rules stated below (see Lines 3 and 6 of Algorithm 2):

Tuple retrieval rule R1. If the query value w is a sensitive value that is at the j^{th} position of the i^{th} sensitive bin (*i.e.*, $w = SB_i[j]$), then the DB owner will fetch the i^{th} sensitive and the j^{th} non-sensitive bins (see Line 3 of Algorithm 2). By Line 2 of Algorithm 2, the DB owner knows that the value w is either sensitive or non-sensitive.

Tuple retrieval rule R2. If the query value w is a non-sensitive value that is at the j^{th} position of the i^{th} non-sensitive bin, then the DB owner will fetch the i^{th} non-sensitive and the j^{th} sensitive bins (see Line 6 of Algorithm 2).

Note that if query value w is in both sensitive and non-sensitive bins, then both the rules are applicable, and they retrieve *exactly the same* bins. In addition, if the value w is neither in a sensitive or a non-sensitive bin, then there is no need to retrieve any bin.

Aside. After knowing the bins, the DB owner sends all the sensitive values in the encrypted form and the non-sensitive values in clear-text to the cloud. The tuple retrieval based on the encrypted values reveals only the tuple addresses that satisfy the requested values. We can also hide the access-patterns by using PIR, ORAM, or DSSE on each required sensitive value. As mentioned in §1, access-pattern-hiding techniques are prone to size and workload-skew attacks. Nonetheless, the use of QB with access-pattern-hiding techniques makes them secure against these attacks, which will be discussed in detail in §6.[11]

Associated bins. *We say a sensitive bin is associated with a non-sensitive bin, if the two bins are retrieved for answering at least one query.*

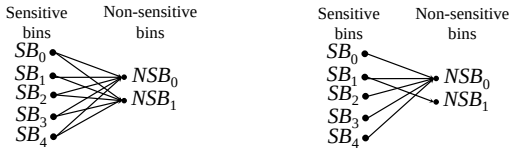
Our aim when answering queries for all the sensitive and non-sensitive values using Algorithm 2 is to associate each sensitive bin with each non-sensitive bin; resulting in the adversary being unable to predict which (if any) is the value shared between two bins.

Exact query value	Returned tuples/Adversarial view	
	Sensitive bin and data	Non-sensitive bin and data
s_1 or ns_1	SB_1:$E(s_1)$,$E(s_6)$	NSB_0:ns_1,ns_2,ns_3,ns_5,ns_{11}
s_2 or ns_2	SB_2:$E(s_2)$,$E(s_7)$	NSB_0:ns_1,ns_2,ns_3,ns_5,ns_{11}
s_3 or ns_3	SB_3:$E(s_3)$,$E(s_8)$	NSB_0:ns_1,ns_2,ns_3,ns_5,ns_{11}
s_4	SB_4:$E(s_4)$,$E(s_9)$	NSB_0:ns_1,ns_2,ns_3,ns_5,ns_{11}
s_5 or ns_5	SB_0:$E(s_5)$,$E(s_{10})$	NSB_0:ns_1,ns_2,ns_3,ns_5,ns_{11}
s_6 or ns_6	SB_1:$E(s_1)$,$E(s_6)$	NSB_1:ns_6,ns_{12},ns_{13},ns_{14},ns_{15}
s_7	SB_2:$E(s_2)$,$E(s_7)$	NSB_1:ns_6,ns_{12},ns_{13},ns_{14},ns_{15}
s_8	SB_3:$E(s_3)$,$E(s_8)$	NSB_1:ns_6,ns_{12},ns_{13},ns_{14},ns_{15}
s_9	SB_4:$E(s_4)$,$E(s_9)$	NSB_1:ns_6,ns_{12},ns_{13},ns_{14},ns_{15}
s_{10}	SB_0:$E(s_5)$,$E(s_{10})$	NSB_1:ns_6,ns_{12},ns_{13},ns_{14},ns_{15}
ns_{11}	SB_4:$E(s_4)$,$E(s_9)$	NSB_0:ns_1,ns_2,ns_3,ns_5,ns_{11}
ns_{12}	SB_0:$E(s_5)$,$E(s_{10})$	NSB_1:ns_6,ns_{12},ns_{13},ns_{14},ns_{15}
ns_{13}	SB_2:$E(s_2)$,$E(s_7)$	NSB_1:ns_6,ns_{12},ns_{13},ns_{14},ns_{15}
ns_{14}	SB_3:$E(s_3)$,$E(s_8)$	NSB_1:ns_6,ns_{12},ns_{13},ns_{14},ns_{15}
ns_{15}	SB_4:$E(s_4)$,$E(s_9)$	NSB_1:ns_6,ns_{12},ns_{13},ns_{14},ns_{15}

Table 6: Queries and returned tuples/adversarial view after retrieving tuples according to Algorithm 2.

Example 3: (Query binning example: continued.) Now, we show how to retrieve tuples. If a query is for a sensitive value, say s_2 (refer to Figure 3), then the DB owner fetches two bins SB_2 and NSB_0. If a query is for a non-sensitive value, say ns_{14}, then the DB owner fetches two bins NSB_1 and SB_3. Thus, it is impossible for the adversary to find (by observing the adversarial view) which is an exact query value from the non-sensitive bin and which is the sensitive value associated with one of the non-sensitive values. This fact is also clear from Table 6, which shows that the adversarial view is not enough to leak information from the joint processing

[11] QB is designed as a general mechanism that provides partitioned data security when coupled with any cryptographic technique. For special cryptographic techniques that hide access-patterns, it may be possible to design a different mechanism that may provide partitioned data security.



(a) Surviving matches after the tuple retrieval following Algorithm 2.

(b) Surviving matches without following Algorithm 2 for ns_{12}, ns_{13}, ns_{14}, ns_{15}; also see Table 6.

Figure 4: An example to show security of QB using surviving matches for 10 sensitive and 10 non-sensitive values.

of sensitive and non-sensitive data, unlike Example 2. In Table 6, $E(s_i)$ shows the encrypted value of s_i, and we are showing the adversarial view only for some of the possible queries. In this example, note that the bin SB_2 gets associated with the bin NSB_0, when answering the query for the value s_2. In a similar manner, the bin NSB_1 gets associated with the bin SB_3, when answering the query for the value ns_{14}.

Algorithm Correctness

We will prove that QB does not lead to information leakage through the joint processing of sensitive and non-sensitive data. To prove correctness, we first define the concept of *surviving matches*. Informally, we show that QB maintains surviving matches among all sensitive and non-sensitive values, resulting in all sensitive bins being associated with all non-sensitive bins. Thus, an initial condition: a sensitive value is assumed to have an identical value to one of the non-sensitive value is preserved.

Surviving matches. We define surviving matches, which are classified as either *surviving matches of values* or *surviving matches of bins*, as follows:

Before query execution. Observe that before retrieving any tuple, under the assumption that no one except the DB owner can decrypt an encrypted sensitive value, say $E(s_i)$, the adversary cannot learn which non-sensitive value is associated with the value s_i. Thus, the adversary will consider that the value $E(s_i)$ is associated with one of the non-sensitive values. Based on this fact, the adversary can create a complete bipartite graph having $|S|$ nodes on one side and $|NS|$ nodes on the other side. The edges in the graph are called *surviving matches of the values*. For example, before executing any query, the adversary can create a bipartite graph for 10 sensitive and 10 non-sensitive values.

After query execution. Recall that the query execution on the datasets creates an adversarial view that guides the adversary to create a (new) bipartite graph containing SB nodes on one side and NSB nodes on the other side. The edges in the new graph (obtained after the query execution) are called *surviving matches of the bins*. For example, after executing queries according to Algorithm 2, the adversary can create a bipartite graph having 5 nodes on one side and 2 nodes on the other side, see Figure 4a. Note that since bins contain values, the surviving matches of the bins can lead to the surviving matches of the values. Hence, from Figure 4a, the adversary can also create a bipartite graph for 10 sensitive and 10 non-sensitive values.

We show that a technique for retrieving tuples that drops some surviving matches of the bins leading to drop of the surviving matches of the values is not secure, and hence, results in the information leakage through non-sensitive data.

Example 4: Dropping surviving matches. In Figure 3, for answering queries for associated values s_1, s_2, s_3, s_5, s_6, ns_1, ns_2, ns_3, ns_5, or ns_6, the DB owner must follow Line 3 or 6 of Algorithm 2 for retrieving the two bins holding corresponding sensitive and non-sensitive data; otherwise, the DB owner cannot retrieve two bins that share a common value. Now, retrieved tuples for these values create an adversarial view as shown in the first six lines except the fourth line of Table 6. However, for answering values s_4, s_7, s_8, s_9, s_{10}, ns_6, ns_{12}, ns_{13}, ns_{14}, or ns_{15} (recall that these values are not associated), if the DB owner does not follow Algorithm 2 and retrieves the bin containing the desired value with any randomly selected bin of the other side, then it could result in the following adversarial view; see Table 7.

Exact query value	Returned tuples/Adversarial view	
	Sensitive bin and data	Non-sensitive bin and data
s_1 or ns_1	SB_1:$E(s_1)$,$E(s_6)$	NSB_0:ns_1,ns_2,ns_3,ns_5,ns_{11}
s_2 or ns_2	SB_2:$E(s_2)$,$E(s_7)$	NSB_0:ns_1,ns_2,ns_3,ns_5,ns_{11}
s_3 or ns_3	SB_3:$E(s_3)$,$E(s_8)$	NSB_0:ns_1,ns_2,ns_3,ns_5,ns_{11}
s_4	SB_4:$E(s_4)$,$E(s_9)$	NSB_0:ns_1,ns_2,ns_3,ns_5,ns_{11}
s_5 or ns_5	SB_0:$E(s_5)$,$E(s_{10})$	NSB_0:ns_1,ns_2,ns_3,ns_5,ns_{11}
s_7	SB_2:$E(s_2)$,$E(s_7)$	NSB_0:ns_1,ns_2,ns_3,ns_5,ns_{11}
s_8	SB_3:$E(s_3)$,$E(s_8)$	NSB_0:ns_1,ns_2,ns_3,ns_5,ns_{11}
s_9	SB_4:$E(s_4)$,$E(s_9)$	NSB_0:ns_1,ns_2,ns_3,ns_5,ns_{11}
s_{10}	SB_0:$E(s_5)$,$E(s_{10})$	NSB_0:ns_1,ns_2,ns_3,ns_5,ns_{11}
ns_{11}	SB_1:$E(s_1)$,$E(s_6)$	NSB_0:ns_1,ns_2,ns_3,ns_5,ns_{11}
s_6 or ns_6	SB_1:$E(s_1)$,$E(s_6)$	NSB_1:ns_6,ns_{12},ns_{13},ns_{14},ns_{15}
ns_{12}	SB_1:$E(s_1)$,$E(s_6)$	NSB_1:ns_6,ns_{12},ns_{13},ns_{14},ns_{15}
ns_{13}	SB_1:$E(s_1)$,$E(s_6)$	NSB_1:ns_6,ns_{12},ns_{13},ns_{14},ns_{15}
ns_{14}	SB_1:$E(s_1)$,$E(s_6)$	NSB_1:ns_6,ns_{12},ns_{13},ns_{14},ns_{15}
ns_{15}	SB_1:$E(s_1)$,$E(s_6)$	NSB_1:ns_6,ns_{12},ns_{13},ns_{14},ns_{15}

Table 7: Queries and returned tuples/adversarial view without following Algorithm 2.

Having such an adversarial view (Table 7), the adversary can learn two facts that

1. Encrypted sensitive tuples of the bins SB_0, SB_2, SB_3, SB_4 have associated non-sensitive tuples in the bin NSB_0, not in NSB_1 (see Figure 4b).
2. Non-sensitive tuples of the bin NSB_1 have their associated sensitive tuples only in the bin SB_1 (see Figure 4b).

Based on this adversarial view (Table 7), the bipartite graph drops some surviving matches of the bins (see Figure 4b). (That fact leads to the dropping of the surviving matches of the values, specifically, surviving matches between sensitive values s_3, s_4, s_5, s_8, s_9, s_{10} and non-sensitive value ns_6, ns_{12}, ns_{13}, ns_{14}, ns_{15}.) Hence, a random retrieval of bins is not a secure technique to prevent information leakage through non-sensitive data accessing.

In contrast, if the DB owner uses Line 3 or 6 of Algorithm 2 for retrieving values that are not associated, the above-mentioned facts (*i*) and (*ii*) no longer hold. Figure 4a shows the case when each sensitive bin is associated with each non-sensitive bin, if Algorithm 2 is followed. Thus, we can see that all the surviving matches of the bins and values are preserved after answering queries. Therefore, for the example of 10 sensitive and 10 non-sensitive values, QB (Algorithms 1 and 2) is secure, and under the given assumptions (§4), the adversary cannot find an exact association between a sensitive and a non-sensitive value.

Security and correctness proofs. are presented in §B.

5.2 A Simple Extension of the Base Case

Algorithm 1 creates bins when the number of non-sensitive data values[12] is not a prime number, by finding the two approximately square factors. However, Algorithm 1 may exhibit a rela-

[12]Recall that we considered the case of $|S| \leq |NS|$.

Algorithm 3: An extension to the bin-creation Algorithm 1 for the base case, $|S| < |NS|$.

Inputs: $|NS|$, $|S|$. **Outputs:** SB, NSB
1 **Function** $bin\_extension(S, NS)$ **begin**
2 Permute all sensitive values
3 $x, y \leftarrow approx\_sq\_factors(|NS|)$: $x \geq y$; $cost_d \leftarrow x + y$
4 $z \leftarrow closest\_SquareNum(|NS|)$, $cost_{sn} \leftarrow 2(z/\sqrt{z})$
5 **if** $(cost_{sn} + \lceil(|NS| - z)/\sqrt{z}\rceil < cost_d)$ **then**
6 Execute Algorithm 1(S, z) and add $(NS - z)/\sqrt{z}$ number of the *remaining* non-sensitive values in each non-sensitive bins
end
7 **else** Execute Algorithm 1(S, NS)
end

tively higher *cost* (*i.e.*, the number of the retrieved tuple) when the sum of the approximately square factors is high.

For example, if there are 41 sensitive data values and 82 non-sensitive data values, then Algorithm 1 creates 2 non-sensitive bins having 41 values in each and 41 sensitive bins having exactly one value in each (Line 4 of Algorithm 1). Consequently, answering a query results in retrieval of 42 tuples. (We may also create two sensitive bins and 41 non-sensitive bins containing exactly two non-sensitive values in each, resulting in retrieval of 23 tuples.) However, the cost can be further reduced by a significant amount, which is explained below.

Example 5: (An example of QB extension — Algorithm 3). Consider again the example of 41 sensitive and 82 non-sensitive values. In this case, 81 is the closest square number to 82. Here, Algorithm 3, described next, creates 9 non-sensitive bins and 9 sensitive bins. By Lines 5 and 6 of Algorithm 1, sensitive values and associated non-sensitive values are allocated, resulting in that a sensitive bin holds at most 5 values and a non-sensitive bin holds at most 10 values. Thus, at most 15 tuples are retrieved to answer a query.

Algorithm 3 description. An extension to the bin-creation Algorithm 1 is provided in Algorithm 3 that handles the case when the number of non-sensitive values ($|S| < |NS|$) is close to a square number.[13] Algorithm 3 first finds two approximately square factors of non-sensitive values and the cost; Line 3. Algorithm 3 also finds a square number, say z, closest to the non-sensitive values and the cost; Line 4. Now, Algorithm 3 creates bins using a method that results in fewer retrieved tuples (Line 5). When Algorithm 3 creates bins using the square number closest to the non-sensitive values (Line 6), the *remaining* non-sensitive values (*i.e.*, $|NS| - z^2$) can be handled by assigning an equal number of the remaining non-sensitive values in the bins. Note that the sensitive and associated non-sensitive values are assigned to bins in an identical manner as in Algorithm 1 (Lines 5-7).

Aside. In QB, the bin size impacts the overall performance, which will be validated by the experiments (§7). Hence, a careful selection of the bin size is mandatory. Note that the retrieval of one sensitive bin having one value and one non-sensitive bin having 41 values can be beneficial (will be analyzed in §6.2.2), if the cryptographic algorithm on the sensitive data requires more time than searching and moving 41 non-sensitive tuples from the cloud to the DB owner side.

5.3 General Case: Multiple Values with Multiple Tuples

[13]The case of $|S| > |NS|$ can be handled by applying Algorithm 3 in a reverse way.

In this section, we will generalize Algorithms 1-3 to consider a case when different data values have different numbers of associated tuples. First, we will show that sensitive values with different numbers of tuples may provide enough information to the adversary leading to the size, frequency-count attacks, and may disclose some information about the sensitive data. Hence, in the case of multiple values with multiple tuples, Algorithms 1-3 cannot be directly implemented. We, thus, develop a strategy to overcome such a situation.

Size attack scenario in the base QB. Consider an assignment of 10 sensitive and 10 non-sensitive values to bins using Algorithm 1; see Figure 3. Assume that a sensitive value, say s_1, has 1000 sensitive tuples and an associated non-sensitive value, say ns_1, has 2000 tuples, while all the other values have only one tuple each. Further, assume that *each data value represents the salary of employees*.

In this example, consider a query execution for a value, say ns_1. The DB owner retrieves tuples from two bins: SB_1 (containing encrypted tuples of values s_1 and s_6) and NSB_0 (containing tuples of values $ns_1, ns_2, ns_3, ns_5, ns_{11}$); see Figure 3. Obviously, the number of retrieved tuples satisfying the values of the bins SB_1 and NSB_0 will be highest (*i.e.*, 3005) as compared to the number of tuples retrieved based on any two other bins. Thus, the retrieval of the two bins SB_1 and NSB_0 provides enough information to the adversary to determine which one is the sensitive bin associated with the bin holding the value ns_1. Moreover, after observing many queries and having background knowledge, the adversary may estimate that 1000 people in the sensitive relation earn a salary equal to the value ns_1.

Thus, in the case of different sensitive values having different numbers of tuples, Algorithm 1 cannot satisfy the *second condition of partitioned data security* (*i.e.*, the adversary is able to distinguish two sensitive values based on the number of retrieved tuples, which was not possible before the query execution, and concludes that a sensitive value (s_1 in the above example) has more tuples than any other sensitive value) though preserving all surviving matches, and holding Theorems 1 and 2 to be true.

In order for the second condition of partitioned data security to hold (and for the scheme to be resilient to the size and frequency-count attacks, as illustrated above), sensitive bins need to hold identical numbers of tuples. A trivial way of doing this is to outsource some encrypted fake tuples such that the number of tuples in each sensitive bin will be identical. However, we need to be careful; otherwise, adding fake tuples in each sensitive bin may increase the *cost*, if all the heavy-hitter sensitive values are allocated to a single bin. This fact will be clear in the following example.

(a) The first way. (b) The second way.

Figure 5: An assignment of 9 sensitive values to 3 bins.

Example 6: (Illustrating ways to assign *sensitive* values to bins to minimize the addition of fake tuples). Consider 9 sensitive values, say $s_1, s_2, \ldots, s_9$, having 10, 20, 30, 40, 50, 60, 70, 80, and 90 tuples, respectively.[14] There are multiple ways of assigning these values to three bins so that we need to add a minimum number

[14]We assume that there are 9 non-sensitive values, and computed that we need 3 sensitive and 3 non-sensitive bins.

of fake tuples to each bin. Figure 5 shows two different ways to assign these values to bins. Figure 5b shows the best way – to minimize the addition of fake encrypted tuples; hence minimizing the cost. However, bins in Figure 5a require us to add 180 and 90 fake encrypted tuples to the bins SB_0 and SB_1, respectively.

Note that there is no need to add any fake tuple if the non-sensitive values have identical numbers of tuples. In that case, the adversary cannot deduce which sensitive bin contains sensitive tuples associated with a non-sensitive value. However, it is obvious that any fake non-sensitive tuple cannot be added in clear-text.

Before describing how to add fake encrypted tuples to bins, we show that a partitioning of sensitive values over SB bins may lead to identical numbers of tuples in each bin, where a bin is not required to hold at most y values, is not a communication-efficient solution. For example, consider 9 sensitive values, where a value, say s_1, has 100 tuples and all the other values, say $s_2, s_3, \ldots, s_9$, have 25 tuples each. In this case, we may get bins as shown in Figure 6. Note that the bins SB_1 and SB_2 are associated with all the three non-sensitive bins while the bin SB_0 is associated with only NSB_0 (thus, the given bins do not prevent the surviving matches). In order to associate each sensitive bin with each non-sensitive bin (and hence, preventing all the surviving matches), we need to ask fake queries for bins $\langle SB_0, NSB_1\rangle$ and $\langle SB_0, NSB_2\rangle$.

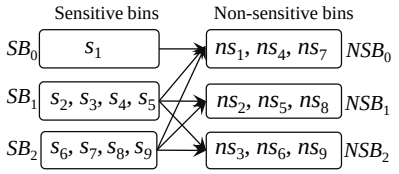


Figure 6: An assignment of a heavy-hitter value but dropping surviving matches.

Adding fake encrypted tuples. As an assumption, we know the number of sensitive bins, say SB, using Algorithm 1 or 3. Here, our objective is to assign sensitive values to bins such that each bin holds identical numbers of tuples while minimizing the number of fake tuples in each bin. To do this, the strategy is given below:

1. Sort all the values in a decreasing order of the number of tuples.
2. Select SB largest values and allocate one in each bin.
3. Select the next value and find a bin that is containing the fewest number of tuples. If the bin is holding less than y values, then add the value to the bin; otherwise, select another bin with the fewest number of tuples. Repeat this step, for allocating all the values to sensitive bins.
4. Add fake tuples' values to the bins so that each bin contains identical numbers of tuples.
5. Allocate non-sensitive values as per Algorithm 1 (Lines 6 and 7).

In Appendix C, we will provide a lemma that states how many fake tuples' values are required to add when using the above strategy and a proof outline for partitioned data security for the above strategy.

6. COMPARING QB TO FULLY CRYPTOGRAPHIC SOLUTIONS

In §5, we described QB, which eliminates the necessity of encrypted data processing over non-sensitive data, while still ensuring security guarantees. In this section, we compare QB to simply storing the entire data (both sensitive and non-sensitive) in an encrypted form and processing queries over encrypted representation. We compare QB to a fully cryptographic approach from both the perspective of performance and security.

From the performance perspective, QB results in saving of encrypted data processing over non-sensitive data – the more the non-sensitive data, the more potential savings. Nonetheless, QB incurs overhead – it converts a single predicate selection query into a set of predicates selection queries over clear-text non-sensitive data, and, a set of encrypted predicates selection queries albeit over a smaller database consisting only of sensitive data. §6.2 discusses how QB performance can be compared to the pure cryptographic approaches for non-indexable and indexable cryptographic mechanisms.

From the security perspective, we have already shown that QB offers no less security than the underlying cryptographic technique it is implemented on. In particular, we showed that QB offers partitioned data security in the sense that the adversary learns nothing from the joint processing of sensitive and non-sensitive data. What is more interesting is that QB offers additional security properties compared to the host cryptographic technique in that it ensures provable security from the size, frequency-count, and workload-skew attacks even if the host technique does not guarantee such properties.

In particular, we incorporate QB with an efficient but a weak cryptographic mechanism. Coupled with QB, the mechanism offers similar security guarantees as much stronger cryptographic techniques, albeit at lower overheads.

6.1 Preliminaries

For our model, we will need the following notations: (*i*) C_{com}: Communication cost of moving one tuple over the network. (*ii*) C_p (or C_e): Processing cost of a single selection query on a plaintext (or encrypted) *dataset*.

The following three parameters dictate the overhead of QB:

α is the ratio between the sizes of the sensitive data (denoted by S) and the entire dataset (denoted by $D = S + NS$, where NS is non-sensitive data).

β is the ratio between the selection query execution time on encrypted data using a cryptographic technique and on clear-text data for a *fixed dataset* on a specific *database system* (in both cases). Note that $\beta = C_e/C_p$.

γ: is the ratio between the processing time of a single selection query on encrypted data and the time to transmit the single tuple over the network from the cloud to the DB owner. Note that $\gamma = C_e/C_{com}$.

Clearly, the parameter β captures the overhead of a cryptographic technique and is dominated by the type of underlying encryption and the cryptographic technique used for searching a predicate. For example, the β value will be higher for search operation using a fully homomorphic encryption technique (because of a higher value of the cryptographic matching operation time) as compared to deterministic encrypted search. Obviously, for the most efficient cryptographic technique, β should be 1.

Based on the above parameters, we can compute the cost of cryptographic and non-cryptographic selection operations as follows:

$Cost_{plain}(x, D)$: is the sum the processing cost of x selection queries on plaintext data and the communication cost of moving all the tuples having x predicates from the cloud to the DB owner, *i.e.*, $x(\log(D)P_p + \rho DC_{com})$.

$Cost_{crypt}(x, D)$: is the sum the processing cost of x selection queries on encrypted data and the communication cost of moving all the tuples having x predicates from the cloud to

the DB owner, *i.e.*, $x(P_eD + \rho DC_{com})$, where ρ is the selectivity of the query.

Given the above, we can compare QB with cryptographic techniques and define a parameter η to be the ratio between the cost of performing a search using QB and the cost of performing the search when the entire data (viz. sensitive and non-sensitive data) is fully encrypted using the cryptographic mechanism. In particular,

$$\eta = \frac{Cost_{crypt}(|SB|, S)}{Cost_{crypt}(1, D)} + \frac{Cost_{plain}(|NSB|, NS)}{Cost_{crypt}(1, D)}$$

Note that if the value of η is less than 1, then QB performs better than a fully cryptographic approach, else it performs worse.

6.2 Comparing QB at Different Levels of Security

6.2.1 Enhancing performance of a cryptographic technique

Several cryptographic search techniques [64, 31, 28, 23, 24] perform a linear scan over encrypted tuples to determine the set of tuple-ids that satisfy the query. Note that cost of evaluating x queries over encrypted data using techniques [64, 31, 28, 23, 24] is amortized and can be performed using a single scan of data. Hence, x is not the factor in the cost corresponding to encrypted data processing. Thus, $Cost_{crypt}(x, D) = P_eD + \rho xDC_{com}$, and so η equation becomes after filling out the values from above:

$$\eta = \frac{C_eS + |SB|\rho DC_{com}}{C_eD + \rho DC_{com}} + \frac{|NSB|\log(D)C_p + |NSB|\rho DC_{com}}{C_eD + \rho DC_{com}}$$

Separating out the communication and processing costs, η becomes:

$$\frac{S}{D}\frac{C_e}{C_e + \rho C_{com}} + \frac{|NSB|\log(D)C_p}{C_eD + \rho DC_{com}} + \frac{\rho DC_{com}(|NSB| + |SB|)}{C_eD + \rho DC_{com}}$$

Substituting for various terms and cancelling common terms provides:

$$\eta = \alpha\frac{1}{(1 + \frac{\rho}{\gamma})} + \frac{\log(D)}{D}\frac{|NSB|}{\beta(1 + \frac{\rho}{\gamma})} + \frac{\rho}{\gamma}\frac{|NSB| + |SB|}{(1 + \frac{\rho}{\gamma})}$$

Note that ρ/γ is very small, thus the term $(1 + \rho/\gamma)$ can be substituted by 1. Given the above, the equation becomes:

$$\eta = \alpha + \log(D)|NSB/D\beta + \rho(|NSB| + |SB|)/\gamma$$

Note that the term $\log(D)|NSB|/D\beta$ is very small since $|NSB|$ is the number of distinct values (approx. equal to $\sqrt{|NS|}$) in a non-sensitive bin, while D, which is the size of a database, is a large number, and β value is also very large. Thus, the equation becomes:

$$\eta = \alpha + \rho(|SB| + |NSB|)/\gamma$$

QB is better than a cryptographic approach when $\eta < 1$, *i.e.*, $\alpha + \rho(|SB| + |NSB|)/\gamma < 1$. Thus, $\alpha < 1 - \frac{\rho(|SB|+|NSB|)}{\gamma}$. Note that the values of $|SB|$ and $|NSB|$ are approximately $\sqrt{|NS|}$, we can simplify the above equation to: $\alpha < 1 - 2\rho\sqrt{|NS|}/\gamma$. If we estimate ρ to be roughly $1/|NS|$ (i.e., we assume uniform distribution), the above equation becomes: $\alpha < 1 - 2/\gamma\sqrt{|NS|}$.

The equation above demonstrates that QB trades increased communication costs to reduce the amount of data that needs to be searched in encrypted form. Note that the reduction in encryption cost is proportional to α times the size of the database, while the increase in communication costs is proportional to $\sqrt{|D|}$, where $|D|$ is the number of distinct attribute values. This, coupled with the fact that γ is much higher than 1 for encryption mechanisms that offer strong security guarantees, ensures that QB almost always outperforms the full encryption approaches. For instance, the cryptographic cost for search using secret-sharing is $\approx 10ms$ [28], while the cost of transmitting a single row ($\approx$ 200 bytes for TPCH Customer table) is $\approx 4\ \mu s$ making the value of $\gamma \approx 25000$. Thus, QB, based on the model, should outperform the fully encrypted solution for almost any value of α, under ideal situations where our assumption of uniformity holds. Figure 7 plots a graph of η as a function of γ, for varying sensitivity and $\rho = 10\%$.

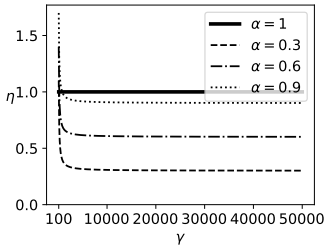


Figure 7: Efficiency graph using equation $\eta = \alpha + \rho(|SB| + |NSB|)/\gamma$.

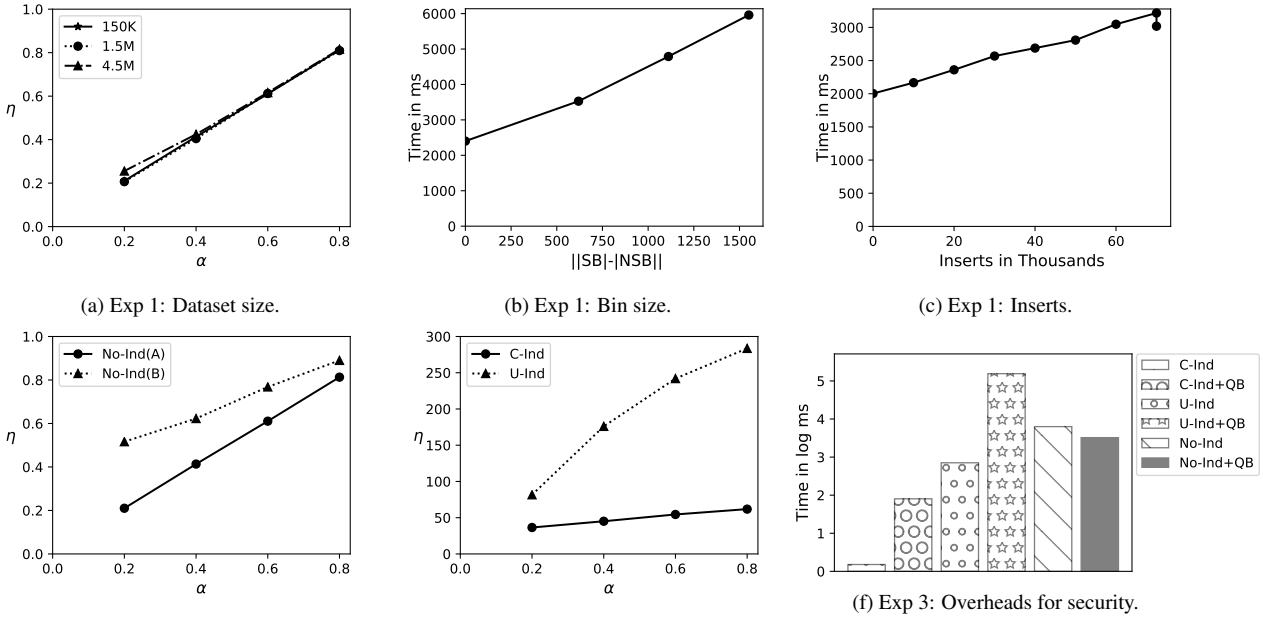
6.2.2 Enhancing Security Properties of a Cryptographic Technique

Several secure and non-secure indexable cryptographic techniques for avoiding the overhead of the linear scan operation have been proposed in the literature [63, 27, 58, 19, 39]. For instance, secure indexable techniques such as [39] require us to first create an index (particularly, a B-tree) on a key attribute at the trusted side before outsourcing the index to the cloud. Access-patterns during retrieval are hidden using ORAM execution over the index. We collectively refer to such techniques as *U-Ind*.

While U-Ind can have significant overhead, more efficient approaches that, however, compromise security have been described in [63, 27, 58]. For instance, in Arx [58], the DB owner stores each domain value v and the frequency of v in the database. The technique encrypts the i^{th} occurrence of v as a concatenated string $\langle v, i\rangle$ thereby ensuring that no two occurrences of v result in an identical ciphertext. Such a ciphertext representation can then be indexed on the cloud-side. During retrieval, the user keeps track of the histogram of occurrences for each value and generates appropriate ciphertexts that can be used to query the index on the cloud. It is not difficult to see that Arx, by itself, is susceptible to the size, frequency-count, workload-skew, and access-pattern attacks. However, the query processing using Arx as efficient as the plaintext version due to the use of an index. Let us refer to the above technique as *C-Ind*.

It is not difficult to see that C-Ind techniques, by itself is susceptible to the size, frequency-count, and workload-skew attacks. However, it is significantly more efficient (almost as efficient as the plaintext version). For instance, for the above-mentioned C-Ind technique (Arx [58]), $\beta = 1.4$ on the system A and $\beta = 2.5$ on the system B, while β values on the systems A and B for a U-Ind approach are very high, 1200 and 2200, respectively.

Let us next explore under what conditions QB improves the performance of an index-based cryptographic technique. As before η must be below 1 for QB to provide any benefit over using a cryptographic approach. Unlike the case of linear scan approaches, the effect of combining QB with indexable approaches may not be as significant, since the time of $|SB|$ searches cannot be absorbed in a single index scan unless all $|SB|$ values lie in a single node of the



(a) Exp 1: Dataset size.

(b) Exp 1: Bin size.

(c) Exp 1: Inserts.

(d) Exp 2: Non-indexable.

(e) Exp 2: Indexable.

(f) Exp 3: Overheads for security.

Figure 8: Experiments.

index. In the worst case, we traverse the index at most $|SB|$ times, unlike Arx [58], which traverses the index only once for a single selection query.

Thus, QB by itself is not expected to improve the performance of an index-based approach. Such a comparison, however, is not fully fair to QB, QB provides additional security from the size, frequency-count, and workload-skew attacks even if the underlying cryptographic technique does not. This can be of a great value for index-based approaches, since index-based approaches (given they only traverse a subset of the database using the index) are vulnerable to such attacks.

However, QB does not protect access-patterns being revealed which could be prevented using ORAM. Determining whether coupling ORAM with Arx mixed with QB or using a more secure cryptographic solutions, *e.g.*, secret-sharing, which uses a linear scan [24] to prevent access-patterns, with QB, more efficient (while QB with both the solutions strengthen the underlying cryptographic technique) is an open question.

7. EXPERIMENTS

In this section, we: (*i*) study QB under different parameter settings (*e.g.*, DB size and bin size) and overhead of insertion, (*ii*) validate the analytical model developed in §6 to identify when QB performs better than a purely cryptographic approach, (*iii*) compare different cryptographic approaches, with and without QB, in terms of security and performance with the goal to identify relative overheads needed to achieve higher levels of security.

Experimental setup. We used virtual machines (VM), each with 2.6 GHz, 4 core processor, 16 GB RAM, and 1TB physical disk in our in-house cloud. Various cryptographic approaches used in experiments (with and without QB) include: (*i*) non-deterministic encryption supported by two popular commercial systems A and B[15] combined with search implemented by retrieving the column in the query, decrypting to determine the relevant rows which are then retrieved subsequently. We name these techniques No-Ind(A) and No-Ind(B), respectively. We also used two indexable approaches: (*i*) U-Ind: a commercial implementation that uses DSSE preventing access-pattern attacks, and (*ii*) C-Ind: a cloud-side index [58]. For each of the 4 approaches, we also developed a QB approach on top.

Dataset and sensitivity. We used TPC-H benchmark to generate the dataset for our experiments. The sensitivity factor (α) was varied from 0.1 to 0.9 for experiments. For QB based approaches, the sensitive part was stored in the underlying cryptographic system while the non-sensitive data stored in plain text over which a non-clustered B^+ tree index was created. Furthermore, sensitive and non-sensitive bins were stored at the DB owner side with other metadata to formulate appropriate partitioned queries. Such metadata storage is propositional to the domain size of the searchable attributes and it is independent of the database size. For TPC-H LINEITEM table, metadata for attributes L_PARTKEY and L_SUPPKEY were 13.6MB and 0.65MB, respectively. The metadata storage can further be reduced using compression. The total execution time was computed by retrieving the tuples associated with the predicate in both the bins. Each experimental setup was repeated 50 times to minimize the effect of outliers.

Exp 1: Robustness of QB. To explore the effectiveness of QB under different DB sizes, we tested QB for 3 DB sizes: 150K, 1.5M, and 4.5M tuples using No-Ind(A) and No-Ind(B) as underlying cryptographic mechanisms. Figure 8a plots η values for the three sizes for No-Ind(A) while varying α. The figure shows that $\eta < 1$, irrespective of the DB sizes, confirming that QB scales to larger DB sizes (results over No-Ind(B) are similar). Figure 8b plots an average time for a selection query using QB with a different bin size, which is in turn governed by the values of $|SB|$ and $|NSB|$, respectively. We plot the effect of $||SB| - |NSB||$ on retrieval time and find that the minimum time is achieved when $|SB| = |NSB|$. Thus, the optimal choice is $|SB| = |SB| = \sqrt{|NS|}$ (Line 5, Algorithm 3). Finally, Figure 8c plots an average cost of search after a number of insertions. In the experiment, insertions are processed (as per the method, given in §A) in batches of 10K and after each

[15]We refer to the systems as A and B to hide their true identity.

batch, selection queries are executed to determine overhead due to insertion. Finally, after 7 batches of insertion, Algorithm 3 is re-executed to recreate bins. The figure confirms that the query cost increases but only marginally in the presence of insertion and (as shown by the last plotted point) reduces by re-binning.

Exp 2. Validation of analytical model. To validate the model for indexable and non-indexable approaches (§6), we implemented fully cryptographic approaches, *e.g.*, C-Ind [58], No-Ind(A), and No-Ind(B), and compared with the corresponding QB by varying α. Figure 8d shows the effect of α on η for non-indexable systems. The results are as per the analytical model, where QB should be effective even at 0.8 sensitivity. Figure 8e presents the results of QB on C-Ind and U-Ind, which are indexable for which, as expected, QB does not improve performance.

Exp 3: Overheads for security. Figure 8f shows average execution times for different schemes with/without QB at a given sensitivity level $\alpha = 0.5$. C-Ind+QB, No-Ind(A)+QB, No-Ind(B)+QB offer the same level of security (viz., security against size, workload-skew, and frequency attacks), which is higher security compared with C-Ind, No-Ind(A) and No-Ind(B). The figure clearly shows C-Ind+QB as the most compelling scheme from both security and performance perspective; which requires significantly less time as compared to a direct implementation of U-Ind without using QB. C-Ind+QB, however, cannot be compared with U-Ind in terms of security, which prevents access-pattern attacks but not preventing workload-skew, frequency, or size attacks. U-Ind+QB is the most secure technique that prevents workload, frequency, and size attacks while also preventing access-pattern attacks. It, however, incurs a significant overhead compared to C-Ind+QB. A more efficient approach compared to U-Ind+QB that offers security against all four attacks is an interesting direction of future work.

8. CONCLUSION

This paper proposes query binning (QB) technique that serves as a meta approach on top of existing cryptographic techniques to support secure selection queries when a relation is partitioned into cryptographically secure sensitive and clear-text non-sensitive sub-relations. Further, we develop (*i*) a new notion of partitioned data security that restricts exposing sensitive information due to the joint processing of the sensitive and non-sensitive relations, and (*ii*) an analytical model to investigate the efficiency of QB against pure cryptographic techniques. Besides improving efficiency, while supporting partitioned security, interestingly, QB enhances the security of the underlying cryptographic technique by preventing size, frequency-count, and workload-skew attacks. As a result, combining QB with efficient but non-secure cloud-side indexable cryptographic approaches can result in an efficient and significantly more secure search. Furthermore, existing indexable/non-indexable cryptographic techniques that prevent access-patterns can also benefit from the added security that QB offers. Further, we extend QB for join and range queries over sensitive and non-sensitive data.

9. REFERENCES

[1] http://www.computerworld.com/article/2834193/cloud-computing/5-tips-for-building-a-successful-hybrid-cloud.html.

[2] https://www.getfilecloud.com/blog/2015/07/5-tips-on-optimizing-your-hybrid-cloud/.

[3] Amazon Aurora, available at:https://aws.amazon.com/rds/aurora/.

[4] MariaDB, available at:https://mariadb.com/.

[5]

[6] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order-preserving encryption for numeric data. In *SIGMOD*, pages 563–574, 2004.

[7] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with cipherbase. In *CIDR*, 2013.

[8] A. Arasu, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. A secure coprocessor for database applications. In *FPL*, pages 1–8, 2013.

[9] A. Arasu and R. Kaushik. Oblivious query processing. In *ICDT*, pages 26–37, 2014.

[10] S. Bajaj and R. Sion. Correctdb: SQL engine with practical query authentication. *PVLDB*, 6(7):529–540, 2013.

[11] S. Bajaj and R. Sion. TrustedDB: A trusted hardware-based database with privacy and data confidentiality. *IEEE Trans. Knowl. Data Eng.*, 26(3):752–765, 2014.

[12] C. Bao and A. Srivastava. Exploring timing side-channel attacks on path-ORAMs. In *HOST*, pages 68–73, 2017.

[13] M. Bellare, A. Boldyreva, and A. O'Neill. Deterministic and efficiently searchable encryption. In *CRYPTO*, pages 535–552, 2007.

[14] E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing. In *EUROCRYPT*, pages 337–367, 2015.

[15] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS*, pages 668–679, 2015.

[16] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, 1998.

[17] V. Ciriani, S. D. C. di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Fragmentation and encryption to enforce privacy in data storage. In *ESORICS*, pages 171–186, 2007.

[18] V. Costan and S. Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[19] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. *Journal of Computer Security*, 19(5):895–934, 2011.

[20] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, M. N. Garofalakis, and C. Papamanthou. Practical private range search in depth. *ACM Trans. Database Syst.*, 43(1):2:1–2:52, 2018.

[21] I. Demertzis and C. Papamanthou. Fast searchable encryption with tunable locality. In *SIGMOD*, 2017.

[22] T. T. A. Dinh, P. Saxena, E. Chang, B. C. Ooi, and C. Zhang. M2R: enabling stronger privacy in mapreduce computation. In *USENIX*, pages 447–462, 2015.

[23] S. Dolev, N. Gilboa, and X. Li. Accumulating automata and cascaded equations automata for communicationless information theoretically secure multi-party computation: Extended abstract. In *SCC@ASIACCS*, pages 21–29, 2015.

[24] S. Dolev, Y. Li, and S. Sharma. Privacy-preserving secret shared computations using mapreduce. volume abs/1801.10323, 2018. Also appeared in DBSec 2016.

[25] S. Doudalis, I. Kotsogiannis, S. Haney, A. Machanavajjhala, and S. Mehrotra. One-sided differential privacy. *CoRR*, abs/1712.05888, 2017.

[26] M. Egorov and M. Wilkison. ZeroDB white paper. *CoRR*, abs/1602.07168, 2016.

[27] Y. Elovici, R. Waisenberg, E. Shmueli, and E. Gudes. A structure preserving database encryption scheme. In *SDM*, pages 28–40, 2004.

[28] F. Emekçi, A. Metwally, D. Agrawal, and A. El Abbadi. Dividing secrets to secure data outsourcing. *Inf. Sci.*, 263:198–210, 2014.

[29] C. Farkas and S. Jajodia. The inference problem: A survey. *SIGKDD Explorations*, 4(2):6–11, 2002.

[30] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[31] N. Gilboa and Y. Ishai. Distributed point functions and their applications. In *EUROCRYPT*, pages 640–658, 2014.

[32] O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *STOC*, pages 182–194, 1987.

[33] S. Goldwasser and S. Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.

[34] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. Cache attacks on Intel SGX. In *EUROSEC*, pages 2:1–2:6, 2017.

[35] P. Grubbs, K. Sekniqi, V. Bindschaedler, M. Naveed, and T. Ristenpart. Leakage-abuse attacks against order-revealing encryption. In *SP*, pages 655–672, 2017.

[36] H. Hacigümüs, B. R. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *SIGMOD*, pages 216–227, 2002.

[37] H. Hacigümüs, S. Mehrotra, and B. R. Iyer. Providing database as a service. In *ICDE*, pages 29–38, 2002.

[38] Y. Ishai and E. Kushilevitz. Private simultaneous messages protocols with applications. In *ISTCS*, pages 174–184, 1997.

[39] Y. Ishai, E. Kushilevitz, S. Lu, and R. Ostrovsky. Private large-scale databases with distributed searchable symmetric encryption. In *RSA*, pages 90–107, 2016.

[40] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, 2012.
[41] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill. Generic attacks on secure outsourced databases. In *CCS*, pages 1329–1340, 2016.
[42] L. Kissner and D. X. Song. Privacy-preserving set operations. In *CRYPTO*, pages 241–257, 2005.
[43] S. Y. Ko, K. Jeon, and R. Morales. The HybrEx model for confidentiality and privacy in cloud computing. In *HotCloud*, 2011.
[44] I. Komargodski and M. Zhandry. Cutting-edge cryptography through the lens of secret sharing. In *TCC*, pages 449–479, 2016.
[45] J. Li, Z. Liu, X. Chen, F. Xhafa, X. Tan, and D. S. Wong. L-EncDB: A lightweight framework for privacy-preserving data queries in cloud computing. *Knowl.-Based Syst.*, 79:18–26, 2015.
[46] R. Li, A. X. Liu, A. L. Wang, and B. Bruhadeshwar. Fast range query processing with strong privacy protection for cloud computing. *PVLDB*, 7(14):1953–1964, 2014.
[47] R. Li, A. X. Liu, A. L. Wang, and B. Bruhadeshwar. Fast and scalable range query processing with strong privacy protection for cloud computing. *IEEE/ACM Trans. Netw.*, 24(4):2305–2318, 2016.
[48] C. Liu, L. Zhu, M. Wang, and Y. Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Inf. Sci.*, 265:176–188, 2014.
[49] W. Lueks and I. Goldberg. Sublinear scaling for multi-client private information retrieval. In *FC*, pages 168–186, 2015.
[50] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam. *L*-diversity: Privacy beyond *k*-anonymity. *TKDD*, 1(1):3, 2007.
[51] P. Martins, L. Sousa, and A. Mariano. A survey on fully homomorphic encryption: An engineering perspective. *ACM Comput. Surv.*, 50(6):83:1–83:33, 2017.
[52] M. Naor and B. Pinkas. Oblivious polynomial evaluation. *SIAM J. Comput.*, 35(5):1254–1281, 2006.
[53] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In *SIGSAC*, pages 644–655, 2015.
[54] O. Ohrimenko, M. Costa, C. Fournet, C. Gkantsidis, M. Kohlweiss, and D. Sharma. Observing and preventing leakage in MapReduce. In *CCS*, pages 1570–1581, 2015.
[55] K. Y. Oktay, M. Kantarcioglu, and S. Mehrotra. Secure and efficient query processing over hybrid clouds. In *ICDE*, pages 733–744, 2017.
[56] K. Y. Oktay, S. Mehrotra, V. Khadilkar, and M. Kantarcioglu. SEMROD: secure and efficient MapReduce over hybrid clouds. In *SIGMOD*, pages 153–166, 2015.
[57] H. Pang and X. Ding. Privacy-preserving ad-hoc equi-join on outsourced data. *ACM Trans. Database Syst.*, 39(3):23:1–23:40, 2014.
[58] R. Poddar, T. Boelter, and R. A. Popa. Arx: A strongly encrypted database system. *IACR Cryptology ePrint Archive*, 2016:591, 2016.
[59] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: processing queries on an encrypted database. *Commun. ACM*, 55(9):103–111, 2012.
[60] M. O. Rabin. How to exchange secrets with oblivious transfer. *IACR Cryptology ePrint Archive*, 2005:187, 2005.
[61] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: trustworthy data analytics in the cloud using SGX. In *SP*, pages 38–54, 2015.
[62] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.
[63] E. Shmueli, R. Waisenberg, Y. Elovici, and E. Gudes. Designing secure indexes for encrypted databases. In *DBSec*, pages 54–68, 2005.
[64] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *2000 IEEE Symposium on Security and Privacy*, pages 44–55, 2000.
[65] J. J. Stephen, S. Savvides, R. Seidel, and P. Eugster. Practical confidentiality preserving big data analysis. In *HotCloud*, 2014.
[66] S. D. Tetali, M. Lesani, R. Majumdar, and T. D. Millstein. MrCrypt: static analysis for secure cloud computations. In *OOPSLA*, pages 271–286, 2013.
[67] Q.-C. To, B. Nguyen, and P. Pucheral. Private and scalable execution of SQL aggregates on a secure decentralized architecture. *ACM Trans. Database Syst.*, 41(3):16:1–16:43, Aug. 2016.
[68] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. *Proc. VLDB Endow.*, 6(5).
[69] F. Wang, C. Yun, S. Goldwasser, V. Vaikuntanathan, and M. Zaharia. Splinter: Practical private queries on public data. In *NSDI*, pages 299–313, 2017.
[70] S. Wang, X. Ding, R. H. Deng, and F. Bao. Private information retrieval using trusted hardware. *IACR Cryptology ePrint Archive*, 2006:208, 2006.
[71] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *CCS*, pages 2421–2434, 2017.
[72] W. K. Wong, B. Kao, D. W. Cheung, R. Li, and S. Yiu. Secure query processing with data interoperability in a cloud database environment. In *SIGMOD*, pages 1395–1406, 2014.
[73] S. Yu, C. Wang, K. Ren, and W. Lou. Achieving secure, scalable, and fine-grained data access control in cloud computing. In *INFOCOM*, pages 534–542, 2010.
[74] S. Yu, C. Wang, K. Ren, and W. Lou. Attribute based data sharing with attribute revocation. In *ASIACCS*, pages 261–270, 2010.
[75] C. Zhang, E. Chang, and R. H. C. Yap. Tagged-MapReduce: A general framework for secure computing with mixed-sensitivity data on hybrid clouds. In *CCGrid*, pages 31–40, 2014.
[76] K. Zhang, X. Zhou, Y. Chen, X. Wang, and Y. Ruan. Sedic: privacy-aware data intensive computing on hybrid clouds. In *CCS*, pages 515–526, 2011.
[77] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI*, pages 283–298, 2017.

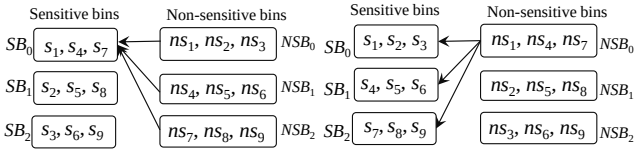
APPENDIX

A. DESIDERATA

A.1 Handling Workload-skew Attack

In the workload-skew attack, as mentioned in §2, the adversary may estimate which encrypted tuples potentially satisfy the frequent selection predicates, while knowing the frequent selection queires by observing many queries in the absence of an access-pattern-hiding scheme. Note that the workload-skew attack is entirely different from the workload attack, where an *active* adversary having the knowledge of partial workload tries the workload to break a secure scheme, and, in this paper, we are not dealing with workload attack, since we assumed an honest and curious adversary, which cannot launch any attack.

In QB, we have so far assumed that the adversary is unaware of the exact workload-skew (more than what is visible via the adversarial view of the queries) due to queries for each predicate. However, the knowledge of the workload-skew can be exploited by the adversary to learn associations between encrypted and plaintext values, breaking the security of the scheme. We illustrate the workload-skew based attack (see Figure 9a) and also our approach for addressing it for the base case of QB. The case of multiple tuples with multiple values can be generalized trivially.



(a) The workload-skew attack. (b) A solution.

Figure 9: The workload-skew attack and solution under QB, where ns_1, ns_4, and ns_7 are frequent predicates.

Figure 9a shows bins created by Algorithm 1 for 9 sensitive values and their associated 9 non-sensitive values. Consider the values ns_1, ns_4, and ns_7 occur most frequently in the query workload. Hence, in this example, the adversary can trivially figure out by observing the sensitive tuple retrieval that only the bin SB_0 has the associated sensitive values with ns_1, ns_4, and ns_7. The reason is that these four bins are retrieved more frequently compared to any other bin. Thus, the adversary can determine that the encrypted values s_1, s_4, and s_7 are associated with either ns_1, ns_4, or ns_7. This is more information than what the adversary had prior to the query execution since each sensitive value, *e.g.*, s_1, could be any of the 9 non-sensitive values. However, it is hard for the adversary to find out which sensitive value out of the three sensitive values of the bin SB_0 is exactly associated with ns_1, ns_4, or

ns_7.[16] In order to prevent the workload-skew attack, we need to allocate sensitive values carefully so that the sensitive values associated with frequent selection predicates are distributed over all the bins. The strategy for handling the workload-skew attack in QB under the assumption of having $k \cdot |SB|$, where $k > 0$, frequent selection predicates is as follows:

1. *Create bins*. Find two largest divisors, say $x \geq y$, of $|NS|$, create $NSB = \lceil |NS|/x \rceil$ non-sensitive bins, and x sensitive bins (Lines 3 of Algorithm 1 or Line 6 of Algorithm 3).
2. *Assign non-sensitive values*. Create groups, each of size x, of the frequent predicates, resulting in $u \leq NSB$ groups. Assign one group to one non-sensitive bin. Now, assign all the remaining non-sensitive values, as follows: if any existing non-sensitive bin has less than x values, then assign the remaining values to the non-sensitive bin such that bin size is x, and then, assign further remaining values to other (empty) non-sensitive bins.
3. *Assign sensitive values*. Assign the sensitive values associated with a non-sensitive value, say $ns_j = NSB_z[j]$, where $0 \leq j \leq x-1$, to the j^{th} sensitive bin at the z^{th} position.

By following the above steps, Figure 9b shows 3 sensitive bins in the case of ns_1, ns_4, and ns_7 as the frequent query predicates. Note that the execution of QB using this strategy insists on retrieving all the sensitive bins for answering frequent predicates. Thus, the adversary cannot determine which bin has a sensitive value associated with the values ns_1, ns_4, or ns_7.[17]

A.2 Non-identical Searchable Attribute-based Column-Level Sensitivity

We showed a way to deal with the column-level sensitivity when sensitive and non-sensitive sides have an identical searchable attribute. However, a slight modification is required when both the sides do not have an identical searchable attribute.

For example, consider a query to retrieve all details of `Eve` from *Employee1* (see Figure 2a) and *Employee3* relations (see Figure 2c). Note that the DB owner can retrieve all tuples belonging to one of the non-sensitive bins having `Eve` from the relation *Employee3*. However, she cannot fetch any sensitive tuple from the *Employee1* relation, because the *Employee1* relation does not contain the attribute `FirstName`.

In order to answer such a query where a searchable attribute is absent in either sensitive or non-sensitive side, we need to delay the query execution on the side that is not containing the searchable attribute. Here, we create on-the-fly bins using the following strategy, where assume that the sensitive dataset does not contain the searchable attribute.

1. *Bin creation*. Find two approximately square factors, say x, y, $x \geq y$, of $|NS|$. Create $|S|/x$ sensitive bins and $\lceil NS/x \rceil$ non-sensitive bins, each is filled with x values of a searchable attribute, say A_i.
2. *Sensitive value allocation and bin retrieval*.
   (a) Consider a query for a value ns_i on the attribute A_i, which does not exist in the sensitive side. Retrieve a non-sensitive bin B_{nsj} that contains the value ns_i.
   (b) After retrieving tuples corresponding to the bin B_{nsj}, the DB owner is able to know a common attribute in the sensitive and the non-sensitive parts. Based on this information, the DB owner creates sensitive bins by distributing the values of the common attribute over the sensitive bins so that each sensitive bin must contain at least one value corresponding to the retrieved non-sensitive bin. The rule to distribute sensitive values is as follows: if $ns_i = B_{nsj}[z]$, then allocate an *associated* sensitive value, say s_i, to a bin B_{sj} at the position z.
   (c) Follows steps 2(a) and 2(b) for other queries on the searchable attribute.

For example, in order to answer the above-mentioned query, we first create bins on the non-sensitive side on the first name attribute of the relation *Employee3*, and then, retrieve tuples of a bin containing employee first name as `Eve`. By this retrieval, the DB owner is able to know `EId` of `Eve`, *i.e.*, E103.

Now, the DB owner creates sensitive bins on the `EId` attribute by distributing ids *associated* with the values of the retrieved non-sensitive bin over the sensitive bins. Now, the DB owner fetches all the tuples from the relation *Employee1* according to one of the sensitive bins containing `EId` = E103.[18]

A.3 Join Queries

Let R be a parent relation that is partitioned into a sensitive relation R_s and a non-sensitive relation R_{ns}. Let S be a child relation that is partitioned into a sensitive relation S_s and a non-sensitive relation S_{ns}. We assume that a tuple of the relation R_s cannot have any tuple in the child table R_{ns}. In the partitioned computing model, the primary-key-to-foreign-key join of R and S is computed as follows:

$$R \bowtie S = (R_s \bowtie S_s) \cup (R_{ns} \bowtie S_{ns}) \cup (R_{ns} \bowtie S_s)$$

Note that our objective is not to build a secure cryptographic technique for joining the sensitive relations. While we use any existing cryptographic technique, *e.g.*, CryptDB [59], SGX-based Opaque [77], [9], [57], or [24] to join sensitive relations, our objective is twofold: (*i*) hide which sensitive tuples (of the relation S_s) join with a non-sensitive tuple (of the relation R_{ns}), and (*ii*) hide which are the encrypted tuples of the output of $(R_s \bowtie S_s) \cup (R_{ns} \bowtie S_s)$ associated with a non-sensitive tuple of $R_{ns} \bowtie S_{ns}$.

In order to join, the relations R_{ns} and S_s, we follow the approach given in [55] that pre-computes all the tuples of R_{ns} that join with S_s. We call all such tuples of R_{ns} as pseudo-sensitive tuples. In [55], the authors found that the size of pseudo-sensitive data does not need to consider the entire R_{ns} as sensitive. Particularly, at 10% of sensitivity level, pseudo-sensitive data is only a fraction (25%) of the entire database.

In our join strategy, before outsourcing the relations R and S, the DB owner finds pseudo-sensitive tuples of R_{ns} and keeps them with sensitive tuples of R_s, resulting in a new relation, denoted by R_{ps} containing sensitive and pseudo-sensitive tuples. Now, the DB owner outsources encrypted relations R_{ps}, S_s and clear-text relations R_{ns}, S_{ns}. Thus,

$$R \bowtie S = (R_{ns} \bowtie S_{ns}) \cup (R_{ps} \bowtie S_s)$$

We use any cryptographic technique for $R_{ps} \bowtie S_s$, and obviously, join of the relations R_{ns} and S_{ns} is carried out in the cleartext. The DB owner uses QB to retrieve any tuple after the join operation. The above strategy can also be extended to non-foreign-key

[16] We are not assuming that a sensitive bin is not associated with each non-sensitive bin. But, because of heavy-hitter queries, the other bins are retrieved less frequently than the bins having frequent selection predicates.

[17] If there are less then y frequent predicates in a non-sensitive bin, then we need to send fake queries as frequent as frequent predicates, leading to retrieval of each sensitive bin.

[18] We consider the relations *Employee1* and *Employee3* for simplicity purposes. The reader may point out that executing a query only on these relations leaks the information that `Eve` is not working in a sensitive department; however, we are able to handle such leakage.

joins by encrypting pseudo-sensitive tuples of S_{ns} with S_s. However, in this case, we need to avoid join of pseudo-sensitive tuples of R_{ns} and S_{ns} in encrypted domain. Hence, the DB owner can add an attribute to each sensitive relation to mark such pseudo-sensitive tuples.

A.4 Range Queries

Let A be an attribute on which we want to execute a range query. For answering a range query, we convert it into the selection query, which can be executed using QB. However, a careless execution of QB for answering a range query, which is converted into selection queries, may result in retrieval of either entire sensitive or non-sensitive data. For example, consider 16 sensitive values, say $s_1, s_2, \ldots s_{16}$, and their associated non-sensitive values, say $ns_1, ns_2, \ldots, ns_{16}$, where the sensitive value s_i is associated with the non-sensitive value ns_i. Figure 10 shows a way to assign these values to bins.

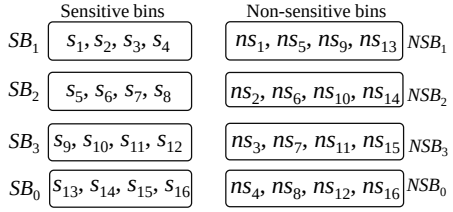


Figure 10: A way to allocate 16 sensitive and non-sensitive values to bins by following Algorithm 1.

Consider a range query for values s_1 to s_4. Answering this range query using QB will result in retrieval of the entire non-sensitive data and the bin SB_1. Our objective is to create bins in a way that results in a few tuple retrieval.

We describe a procedure for the case $|S| \leq |NS|$, as the restriction is followed by Algorithm 1 in §5.1. We use the example of 16 sensitive and 16 non-sensitive values of the attribute A. In order to answer range queries, the DB owner builds a full binary tree on the unique values of the attribute A of the non-sensitive relation and traverses the tree to find a node that covers the range. Thus, the DB owner retrieves tuples satisfying a larger range query that also covers the desired range query. Note that many papers [37, 46, 47, 20] used the same approach of fetching a large range value to satisfy the desired range value, and hence, preventing exact range values to be revealed to the adversary.

Full binary tree and bin creation. The DB owner first builds a full binary tree for the values of the attribute A of the non-sensitive relation; see Figure 11 for 16 non-sensitive values. For each level of the tree, the DB owner applies Algorithm 1 that takes nodes of the level as inputs. In particular, for the leaf nodes, *i.e.*, level 0, Algorithm 1 takes 16 non-sensitive values, and produces 4 sensitive and 4 non-sensitive bins, by following Lines 3-7 of Algorithm 1.

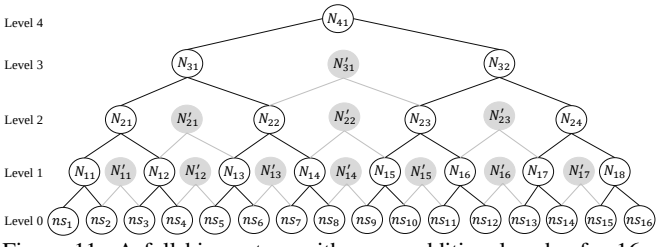


Figure 11: A full binary tree with some additional nodes for 16 non-sensitive values.

At the level 1, Algorithm 1 takes 8 inputs that represent the nodes ($N_{11}, N_{12}, \ldots, N_{18}$; see white nodes in Figure 11) at the level 1, and each input value of the level 1 holds two non-sensitive values, which are child nodes of a level 1's node. For example, the node N_{11} holds two values ns_1, ns_2. For the 8 values, Algorithm 1 provides two non-sensitive bins (each is containing 8 values) and four sensitive bins (each is containing 4 values). Let NSB_{ij} be the j^{th} non-sensitive bin at the i^{th} level, and let SB_{ij} be the j^{th} sensitive bin at the i^{th} level. Thus, Algorithm 1 produces the following bins:

$$NSB_{10} \text{ containing } \langle N_{11}, N_{12}, \ldots, N_{14} \rangle,$$
$$NSB_{11} \text{ containing } \langle N_{15}, N_{16}, \ldots, N_{18} \rangle,$$
$$SB_{10} \text{ containing } \langle s_1, s_2, s_9, s_{10} \rangle,$$
$$SB_{11} \text{ containing } \langle s_3, s_4, s_{11}, s_{12} \rangle,$$
$$SB_{12} \text{ containing } \langle s_5, s_6, s_{13}, s_{14} \rangle,$$
$$SB_{13} \text{ containing } \langle s_7, s_8, s_{15}, s_{16} \rangle.$$

At level 2, Algorithm 1 takes 4 inputs that represent the nodes ($N_{21}, N_{22}, N_{23}, N_{24}$; see white nodes in Figure 11) at the level 2, and each input value of the level 2 holds four non-sensitive values, which are grandchild nodes of a level 2's node. For the 4 input values, Algorithm 1 provides two non-sensitive bins (each is containing 8 values) and two sensitive bins (each is containing 8 values), as follows:

$$NSB_{20} \text{ containing } \langle N_{21}, N_{22} \rangle,$$
$$NSB_{21} \text{ containing } \langle N_{23}, N_{24} \rangle,$$
$$SB_{20} \text{ containing } s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12},$$
$$SB_{21} \text{ containing } s_1, s_2, s_3, s_4, s_{13}, s_{14}, s_{15}, s_{16}.$$

The DB owner follows the same procedure for the higher nodes, except the root node and child nodes of the root node.

Further, at each level except the root node, the child nodes of the root node, and the leaf nodes, the DB owner creates additional nodes (see gray-colored nodes in Figure 11) that become parent nodes of the lower level's two adjacent nodes that do not have a common parent node. The algorithm given in [20] also uses these additional nodes for answering a range query. For example, at the level 2 in Figure 11, the DB owner creates 7 such nodes. Let NSB'_{ij} be the j^{th} non-sensitive bin at the i^{th} level for these additional nodes, and let SB'_{ij} be the j^{th} sensitive bin at the i^{th} level for these additional nodes. Algorithm 1 takes these 7 inputs and produces 2 non-sensitive bins (each is containing 8 values) and 4 sensitive bins (each is containing 8 values), as follows:[19]

$$NSB'_{10} \text{ containing } \langle N'_{11}, N'_{12}, \ldots, N'_{14} \rangle,$$
$$NSB'_{11} \text{ containing } \langle N'_{15}, N'_{16}, N'_{17}, \text{ 2 fake tuples} \rangle,$$
$$SB'_{10} \text{ containing } \langle s_2, s_3, s_{10}, s_{11} \rangle,$$
$$SB'_{11} \text{ containing } \langle s_4, s_5, s_{12}, s_{13} \rangle,$$
$$SB'_{12} \text{ containing } \langle s_6, s_7, s_{14}, s_{15} \rangle,$$
$$SB'_{13} \text{ containing } \langle s_8, s_9, \text{ 2 fake tuples} \rangle.$$

Bin retrieval and answering range queries. We provide two approaches: best-match method and least-match method, for retrieving the bins in answering a range query.

Best-match method. This method traverses the tree in a bottom-up fashion and finds a node that covers the entire range. Then, it retrieves a non-sensitive bin corresponding to this node and a sensitive bin, by following Algorithm 2.

[19]Note that the bins NSB'_{11} and SB'_{13} will ask to fetch two fake tuples each to maintain an identical-sized bin.

For example, if the query is for values ns_1 to ns_4, then by traversing the tree (see Figure 11) in a bottom-up fashion, the DB owner retrieves a non-sensitive bin corresponding to the level 2, since the node N_{21} covers the entire range. Thus, the DB owner retrieves the bins NSB_{20} and SB_{21}.

Least-match method. Assume a query is for values ns_8 to ns_{12}. The best-match method will find only the root node that satisfies this query, and hence, it will result in retrieval of entire non-sensitive or sensitive relation. Thus, we propose a different method that breaks the range query into many sub-range queries and finds minimal set of nodes that cover the range.

For example, the node N_{23} satisfies the query for value ns_9 to ns_{12}, and the leaf node having the value 8 satisfies the query for the value ns_8. Thus, the DB owner retrieves the bins NSB_{21} and SB_{20} to satisfy the query for the value ns_9 to ns_{12}, and a sensitive bin and a non-sensitive bin to satisfy the query for the value ns_8.

Aside: using additional nodes for answering a range query by following Algorithm 2. Assume a query is for values ns_4 to ns_7. The best-match method finds only the root node that satisfies the query, and hence, results in retrieval of the entire non-sensitive or sensitive relation. In contrast, the least-match method will break the query into sub-range queries, such as (*Q1*) a query for ns_4, a query for ns_5, ns_6, and a query for ns_7, or (*Q2*) four selection queries one for each value.

The first query (*Q1*) will find the node N_{13} that covers the values ns_5, ns_6, and two leaf nodes one for ns_4 and another for ns_7. This will result in retrieval of 28 tuples, such as one sensitive bin and non-sensitive bin for s_4 (containing 4 tuples in each; see Figure 10), one sensitive bin and non-sensitive bin for s_7 (containing 4 tuples in each; see Figure 10), and the bin NSB_{10} (containing 8 tuples) and SB_{12} (containing 4 tuples) for answering the query for a range ns_5 to ns_6. However, the second query (*Q2*) will be worse in term of retrieving the tuples. It will result in retrieval of the entire non-sensitive data (see Figure 10).

In order to reduce the number of retrieved tuples, the DB owner can use the bins for the additional nodes. In particular, the DB owner finds that the nodes N'_{12} and N'_{13} that satisfy the value ns_4-ns_5 and ns_6-ns_7, respectively. Thus, the bins NSB'_{10} (containing 8 tuples), SB'_{11} (containing 4 tuples), SB'_{12} (containing 4 tuples) can fulfill the query, and will result in retrieval of 16 tuples.

Note that by using the bins for the additional nodes, one can answer queries for two adjacent nodes that do not share a common parent in the original full binary tree, for example, values 8 and 9.

A.5 Insert Operation and Re-binning

QB does not allow outsourcing new tuples immediately as the new tuples arrive at the DB owner. The DB owner collects enough number of tuples before outsourcing them, while she can either update the existing bins (by increasing an identical size of each bin) or create all new bins.

Particularly, the DB owner waits for new tuples until she collects new sensitive and non-sensitive values equals to the number of existing sensitive and non-sensitive bins such that each bin receives a new value. Let p and q be the number of existing sensitive and non-sensitive bins, respectively. Note that when collecting p sensitive and q non-sensitive values, the DB owner does not outsource these values if they will not become a part of each existing sensitive or non-sensitive bin. If the new values become a part of only one sensitive and one non-sensitive bin, it reveals an association of values.

However, an insertion of more values in existing bins incur the overhead when retrieving a bin, as shown in Figure 8c and Experiment 1 §7. Hence, Algorithm 1 is re-executed when the overhead crosses a user-defined threshold.

Now, we describe a procedure for outsourcing new tuples while using the existing sensitive and non-sensitive bins. Let s_i and ns_j be the value of new sensitive and non-sensitive tuples, respectively. When inserting new tuples, the value s_i or ns_j may exist in the outsourced data, and based on the existence of the values we classify them into four groups, as follows: (*i*) *old sensitive value* (old-S): the value s_i exists in the outsourced sensitive data, (*ii*) *new sensitive value* (new-S): the value s_i does not exist in the outsourced sensitive data, (*iii*) *old non-sensitive value* (old-NS): the value ns_j exists in the outsourced non-sensitive data, and (*iv*) *new non-sensitive value* (new-NS): the value ns_j does not exist in the outsourced non-sensitive data.

Based on the above-mentioned four types of values, the following four possible insert scenarios are allowed while using QB.

1. *Inserting old-S and old-NS*. This scenario is trivial to handle and does not require any update to the existing bins. The DB owner outsources the encrypted sensitive tuples and cleartext non-sensitive data.
2. *Inserting new-S and new-NS*. The DB owner increases the size of each bin by one. If the values s_i and ns_i are associated, then the DB owner inserts the values into existing associated bins. If the values s_i and ns_i are not associated, the DB owner inserts the values randomly to bins, one sensitive (or non-sensitive) value in each existing sensitive (or non-sensitive) bin.
3. *Inserting old-S and new-NS*. Inserting tuples of s_i does not require an update to the sensitive bins. The DB owner checks whether the value ns_j has an associated sensitive value or not in the outsourced data, by following Line 6-7 of Algorithm 2. If the value ns_j has an associated sensitive value, say s_k, then the DB owner updates the non-sensitive bin associated with a sensitive bin holding s_k with the value ns_j, according to Line 6 of Algorithm 1. If the value ns_j has no associated sensitive value, then the DB owner randomly inserts each non-sensitive value, one per non-sensitive bin.
4. *Inserting new-S and old-NS*. This case is just opposite of the previous case.

Note that all the four scenarios may require to outsource some fake tuples to have identical-sized sensitive bins. The update/delete operation can also be done as an insert operation, where some additional tuples are outsourced to notify the non-existence of tuples.

A.6 Conjunctive Queries

As defined, QB only works for selection queries with a single attribute in the search clause. Conjunctive queries that contain several such conjuncts can also be supported in several ways. First, note that QB can be applied to multiple attributes, say A and B, in a relation. During query processing, if a query refers to both attributes A and B, we can select the more selective index and execute QB on it without inference attacks. Using QB on both attributes simultaneously, however, unless done carefully, can lead to leakage. An approach to apply QB is to consider attributes that appear commonly together in queries as a single (paired) attribute. Thus, values of this paired attribute would be attribute value pairs on which QB can be applied. In general, the relation scheme will need to be partitioned into attribute subsets on which QB can be applied. During a query execution, the query processing algorithm will choose the corresponding attribute subset that is most beneficial (will result in least overhead) to execute the query. One such solution is to create partitions of singleton attributes but then conjunctive queries will run on a single attribute and reduce to the first solution

B. SECURITY PROOFS OF QB

We prove that QB is secure and satisfies the definition of partitioned data security (Theorem 2) by first proving that all the sensitive bins are associated with all the non-sensitive bins (Theorem 1), which is intuitively clear by Example 4. Recall that the only way a surviving match could be removed is if there is no sensitive value in a sensitive bin, say SB_j that does not have an associated non-sensitive value. In this case for answering a value belonging to SB_j, we retrieve either only the bin SB_j or the bin SB_j with any randomly selected non-sensitive bin. Note that the adversary cannot learn anything from the encrypted data, since the keys are only known to the DB owner.

Theorem 1 *Let $|S|$ and $|NS|$ be the number of sensitive and non-sensitive values, respectively. By following Algorithm 1, $|S|$ and $|NS|$ values are distributed over SB sensitive and NSB non-sensitive bins, respectively. Answering a set of queries using QB (Algorithm 2) will not remove any surviving matches of the bins and that leads to preserve all the surviving matches of the values.*

PROOF. We show that QB will not remove any surviving matches of the bins by showing that a sensitive bin, say SB_j, must be associated with all the non-sensitive bins. A similar argument can be proved for any non-sensitive bin. Let y be the number of sensitive values in the bin SB_j, and let $p \geq y$, ($p = NSB$) be the number non-sensitive bins. We will prove the following three arguments:

1. If a sensitive value, say $s_i \in SB_j$, is associated with a non-sensitive value (*i.e.*, $\exists ns_z \in R_{ns} : ns_z \overset{a}{=} s_i$), then two bins, SB_j, and one non-sensitive bin, holding the value ns_z, are retrieved.
2. If a sensitive value, say $s_i \in SB_j$, is not associated with any non-sensitive value (*i.e.*, $\forall ns_j \in R_{ns} : s_i \overset{a}{\neq} ns_j$), then the bin SB_j and one of the non-sensitive bins are retrieved. Following that, if all the sensitive values of the bins SB_j are not associated with any non-sensitive value (*i.e.*, $\forall ns_j \in R_{ns}, \forall s_i \in SB_j : s_i \overset{a}{\neq} ns_j$), then the bin SB_j and y different non-sensitive bins are retrieved.
   By proving the first and second arguments, we will show that if there are *only* y non-sensitive bins, then a sensitive bin must be associated with all the y non-sensitive bins. The following third argument will consider more than y non-sensitive bins.
3. If there are more than y non-sensitive bins (say, $NSB_y, NSB_{y+1}, \ldots, NSB_p$) having x values that are not associated with any sensitive value (*i.e.*, $\forall ns_j \in NSB_y \vee NSB_{y+1} \vee \ldots \vee NSB_p, ns_j \overset{a}{\neq} s_i, i = 1, 2, \ldots, |S|$), then each of these non-sensitive bins must be associated with the bin SB_j.

By satisfying the above three arguments, we prove that, thus, the bin SB_j is associated with all non-sensitive bins, and hence, all surviving matches of the bins and, eventually, values are preserved.
First case. The value s_i is allocated to $(i \ modulo \ x)^{th}$ sensitive bin at an index, say z, where $z = 0, 1, \ldots y-1$, and its associated non-sensitive value is allocated to the $(i \ modulo \ x)^{th}$ position of the z^{th} non-sensitive bin. When answering a query for s_i according to the rule R1, the bin SB_j with the bin NSB_z are retrieved. Consequently, the desired tuples containing s_i and its associated non-sensitive value are retrieved, and that are correct answers to the query.
Second case. When answering a query for the value $s_i = SB_j[u]$ ($u \in 0, 1, y-1$) that does not have any associated non-sensitive value, by following the rule R1, the bin SB_j with one of the non-sensitive bin NSB_u are retrieved. Moreover, answering queries for all the y values (0, 1, $y-1$) of the bin SB_j, by following rule R1, requires us to retrieve the SB_j with all the $y-1$ (0, 1, $y-1$) non-sensitive bins.
Third case. Since the non-sensitive bin, say NSB_z, where $z = y, y+1, \ldots, p$, must hold a value at the j^{th} position, by following the rule R2, the bin NSB_z and the sensitive bin SB_j are fetched for answering a query for ns_j.

Therefore, the bin SB_j is associated with all the non-sensitive bins, and hence, all the surviving matches between the values of the bin SB_j and all the non-sensitive bins are also maintained. □

Since we proved all sensitive bins are associated with all the non-sensitive bins, based on this fact, we will show that the first condition of partitioned data security holds to be true for any query. Here, we do not show the second equation of partitioned data security definition (*i.e.*, $Pr_{adv}[s_i \overset{r}{\sim} s_j|X] = Pr_{adv}[s_i \overset{r}{\sim} s_j|X, q(w)(R_s, R_{ns})[A]]$); recall that here in the base case, we assumed that a value has only a single sensitive tuple; hence, the condition holds true.

Theorem 2 (Preserve partitioned data security) *Let R be a relation containing sensitive and non-sensitive tuples. Let R_s and R_{ns} be the sensitive and non-sensitive relations, respectively. Let $q(w)(R_s, R_{ns})[A]$ be a query, q, for a value w in the attribute A of the R_s and R_{ns} relations. Let X be the auxiliary information about the sensitive data, and Pr_{Adv} be the probability of the adversary knowing any information. Let e_i be the i^{th} sensitive tuple value in the attribute A of the relation R_s and ns_j is the j^{th} non-sensitive value in the attribute A of the relation R_{ns}. An execution of a set of queries on the attribute A on the relations using QB leads to the following equation to be true:*

$$Pr_{adv}[e_i \overset{a}{=} ns_j|X] = Pr_{adv}[e_i \overset{a}{=} ns_j|X, AV]$$

where $i \in 1, 2, \ldots, |S|$ and $j \in 1, 2, \ldots, |NS|$.

Proof sketch. We provide an example of four values to show the correctness of the above theorem. Let v_1, v_2, v_3, and v_4 be values containing only one sensitive and one non-sensitive tuple. Let E_1, E_2, E_3, and E_4 be encrypted representations of these values in an arbitrary order, *i.e.*, it is not mandatory that E_1 is the encrypted representation of v_1. In this example, the cloud stores an encrypted relation, say R_s, containing four encrypted tuples with encrypted representations E_1, E_2, E_3, E_4 and a clear-text relation, say R_{ns}, containing four clear-text tuples with values v_1, v_2, v_3, v_4. The objective of the adversary is to deduce a clear-text value corresponding to an encrypted value. Note that before executing a query, the probability of an encrypted value, say E_i, to have the clear-text value, say v_i, $1 \leq i \leq 4$ is 1/4, which QB maintains at the end of a query.

Assume that the user wishes to retrieve the tuple containing v_1. By following QB, the user asks a query, say $q(E_1, E_3)(R_s)$, on the encrypted relation R_s for E_1, E_3, and a query, say $q(v_1, v_2)(R_{ns})$, on the clear-text relation R_{ns} for v_1, v_2. After executing the queries, the adversary holds an adversarial view given in Table 8.

In this example, we show that the probability of finding the clear-text value of an encrypted representation, say E_i, $1 \leq i \leq 4$, remains identical before and after a query. In order to show that when a query comes for $2 \times \sqrt{n}$ values by following QB, where n is the number of values in the non-sensitive relation, $\sqrt{n}$ values are asked for the sensitive relation and $\sqrt{n}$ values are asked for the non-sensitive relation, we need to figure out:

Exact query value (hidden from adversary)	Returned tuples/Adversarial view	
	Sensitive data	Non-sensitive data
v_1	E_1,E_3	v_1,v_2

Table 8: Queries and returned tuples/adversarial view after executing a query for v_1, by following Algorithm 2.

1. All possible allocations of the non-sensitive $\sqrt{n}$ values, say $v_1, v_2, \ldots, v_{\sqrt{n}}$, to $\sqrt{n}$ encrypted sensitive values, say $E_1, E_2, \ldots, E_{\sqrt{n}}$. Here, we use the term *allocation* to show the fact that the encrypted representation of E_i has the clear-text value v_i.
   In our example of four values, we find allocations of four non-sensitive values v_1, v_2, v_3, v_4 to encrypted representation E_1, E_2, E_3, E_4.
2. All possible allocations of $\sqrt{n}$ non-sensitive values, except one non-sensitive value, say v_i, that is allocated to an encrypted sensitive value, say E_i, to the remaining encrypted sensitive values.
   In the case of four values and above-mentioned queries, we find allocations of the non-sensitive values v_2, v_3, v_4 to the encrypted sensitive values E_2, E_3, E_4 while assuming that the encrypted representation of v_1 is E_1.

The ratio of the above two provides the probability of finding a clear-text value corresponding to its encrypted value after the query execution.

When the query arrives for $\langle E_1, E_3, v_1, v_2\rangle$, the adversary gets the fact that the clear-text representation of E_1 and E_3 cannot be v_1 and v_2 or v_3 and v_4. If this will happen, then there is no way to associate a sensitive bin with each non-sensitive bin. Now, if the adversary considers the clear-text representation of E_1 is v_1, then the adversary has the following four possible allocations of the values v_1, v_2, v_3, v_4 to E_1, E_2, E_3, E_4:

$$\langle v_1, v_2, v_3, v_4\rangle, \langle v_1, v_2, v_4, v_3\rangle,$$
$$\langle v_1, v_3, v_4, v_2\rangle, \langle v_1, v_4, v_3, v_2\rangle.$$

However, the allocations $\langle v_1, v_3, v_2, v_4\rangle$ and $\langle v_1, v_4, v_2, v_3\rangle$ to E_1, E_2, E_3, and E_4 cannot exist. Since the adversary is not aware of the exact clear-text value of E_1, the adversary also considers the clear-text representation of E_1 is v_2. This results in four more possible allocations of the values to E_1, E_2, E_3, and E_4, as follows:

$$\langle v_2, v_1, v_3, v_4\rangle, \langle v_2, v_1, v_4, v_3\rangle,$$
$$\langle v_2, v_3, v_4, v_1\rangle, \langle v_2, v_4, v_3, v_1\rangle.$$

However, $\langle v_2, v_3, v_1, v_4\rangle$ and $\langle v_2, v_4, v_1, v_3\rangle$ cannot exist. Similarly, assuming the clear-text representation of E_1 is v_3 or v_4, we get the following 8 more possible allocations of the values to E_1, E_2, E_3, and E_4:

$$\langle v_3, v_1, v_2, v_4\rangle, \langle v_3, v_2, v_1, v_4\rangle,$$
$$\langle v_3, v_4, v_1, v_2\rangle, \langle v_3, v_4, v_2, v_1\rangle,$$
$$\langle v_4, v_1, v_2, v_3\rangle, \langle v_4, v_2, v_1, v_3\rangle,$$
$$\langle v_4, v_3, v_1, v_2\rangle, \langle v_4, v_3, v_2, v_1\rangle.$$

Here, the following four allocations of the values to encrypted representation cannot exist:

$$\langle v_3, v_1, v_4, v_2\rangle, \langle v_3, v_2, v_4, v_1\rangle,$$
$$\langle v_4, v_1, v_3, v_2\rangle, \langle v_4, v_2, v_3, v_1\rangle.$$

Thus, the retrieval of the four tuples containing one of the following: $\langle E_1, E_3, v_1, v_2\rangle$, results in 16 possible allocations of the values v_1, v_2, v_3, and v_4 to E_1, E_2, E_3, and E_4, of which only four possible allocations have v_1 as the clear-text representation of E_1. This results in the probability of finding $E_1 = v_1$ is 1/4. A similar argument also holds for other encrypted values. Hence, an initial probability of associating a sensitive value with a non-sensitive value remains identical after executing a query.

Thus, we can conclude the following:

1. All possible allocations of $\sqrt{n}$ non-sensitive values, except one non-sensitive value, say v_1, that we allocate to an encrypted sensitive value, say E_1, to the remaining encrypted sensitive values is $(n-1)! - x$, where n is the number of values in the non-sensitive relation and x is the number of allocations of values $v_2, v_3, \ldots, v_{\sqrt{n}}$ to $E_2, E_3, \ldots, E_{\sqrt{n}}$ that cannot exist.
2. All possible allocations of the non-sensitive $\sqrt{n}$ values, say $v_1, v_2, \ldots, v_{\sqrt{n}}$, to $\sqrt{n}$ encrypted sensitive values, say $E_1, E_2, \ldots, E_{\sqrt{n}}$, is $n \times ((n-1)! - x)$. This is true because we cannot allocate any combination of the values asked in the query to any encrypted representations that are asked by the query.

Thus, the retrieval of $2\times\sqrt{n}$ values results in $n\times((n-1)!-x)$ possible allocations of $\sqrt{n}$ non-sensitive values to $\sqrt{n}$ encrypted sensitive values, while $(n-1)!-x$ allocations exist when a queried non-sensitive value is assumed to be the clear-text of a queried encrypted representation. Therefore, the probability of finding the exact allocation of the non-sensitive values to encrypted sensitive value while considering a non-sensitive value is the clear-text of an encrypted value is $\frac{(n-1)!-x}{n\times((n-1)!-x)} = \frac{1}{n}$.

C. PROOF OUTLINE FOR THE GENERAL CASE

The following lemma shows how many fake tuples are required to be added to a sensitive bin, resulting in identical numbers of tuples in each sensitive bin.

Lemma 1 *Let t_l and t_s be the largest and smallest number of tuples have an identical value, respectively. By following the strategy presented in Section 5.3, the strategy adds at most $max(t_l, t_l - t_s)$ fake tuples in a sensitive bin to have identical numbers of tuples in each bin.*

Now, we borrow notations from Theorem 2 that also holds for the general case of QB and give a proof outline of the second condition of partitioned data security.

Theorem 3 (Preserve partitioned data security) *An execution of a set of queries on an attribute, say A, on the relations R_s and R_{ns}, where different values of the attribute A have different numbers of tuples, using QB leads to the following equations to be true: $Pr_{adv}[v_i \overset{r}{\sim} v_j|X] = Pr_{adv}[v_i \overset{r}{\sim} v_j|X, q(w)(R_s, R_{ns})[A]]$, where $v_i, v_j \in Domain(A)$ of the relation R_s.*

Proof sketch. Recall that we are not dealing with how does an encryption mechanism reveal an order of sensitive values. In our context, the probability of relating any two sensitive values will not be identical when the number of output tuples to a query to one of the values is more than the number of tuples to a query to another value. Since we add fake tuples to sensitive bins, each sensitive bin has identical numbers of tuples. Hence, having a heavy-hitter non-sensitive value that is associated with a heavy-hitter sensitive value, the adversary cannot distinguish between any two sensitive bins and deduce which sensitive bin holds the heavy-hitter sensitive value. Thus, the probability of relating two sensitive values remains a constant after answering queries for any two values.

D. CLOUD COMPUTATION COST ESTIMATION FOR A SELECTION QUERY

The selection operation at the cloud is performed by accessing indexes on the non-sensitive data and applying a cryptographic search technique, which may be indexable (*e.g.*, DSSE [39]) or non-indexable, on the sensitive data. This section shows the computation cost at the cloud while using non-indexable cryptographic techniques (Lemma 2) and indexable cryptographic techniques (Lemma 3).

Lemma 2 (Computation cost at the cloud side—non-indexable) *Let d and r be unique number of values and an average number of tuples corresponding to a value, respectively, in a key attribute of a relation, say D ($= d \cdot r$). Let $|NS|$ be the number of non-sensitive data values. Let $\alpha < 1$ be a ratio between the sensitive data and the entire dataset. In QB, the computation cost at the cloud using a non-indexable cryptographic technique is $\mathcal{O}(\alpha D + \sqrt{|NS|}(r + \log(1-\alpha)D))$.*

Proof sketch. The cloud reads each sensitive tuple, resulting in $\alpha \cdot D$ cost, while at the same time the cost of searching $|SB|$ predicates can be absorbed, as explained in §6.2.1. Hence, the cost for searching the tuples is $|B_s| \cdot \alpha \cdot D$. In addition, the cloud needs to look up the index structure, particularly a B-tree on the non-sensitive relation and then retrieves all the desired tuples, resulting in $\mathcal{O}(|\sqrt{NS}(r + \log((1-\alpha)D)))$ cost.

Lemma 3 (Computation cost at the cloud side—index structure) *In QB technique, the computation cost at the cloud using an indexable technique, namely B-tree is $\mathcal{O}(|SB| \cdot r \cdot \log(\alpha D) + |NSB|(r + \log((1-\alpha)D)))$, where $|SB|$ is the number of sensitive values in a sensitive bin, and $|NSB|$ is the number of non-sensitive values in a sensitive bin.*

Proof sketch. The cloud needs to look up an encrypted index for each sensitive ($|SB| \cdot r$) that results in $\mathcal{O}(|SB| \cdot r \cdot \log(\alpha D)$ cost for traversal of the encrypted index. In addition, the cloud traverses an index on the non-sensitive data to retrieve all the tuples having $|mathitNSB|$ values, results in $|NSB|(r + \log(1-\alpha)D))$ cost. Here, the cost is dominated by an underlying cryptographic indexable structure, which is also clear form §6.2.2.