ACM DL DIGITAL LIBRARY
(http://www.acm.org)

☰ Art... ...gation

(http://www.acm.org)

PANDA: Partitioned Data Security on Outsourced Sensitive and Non-sensitive Data

SHARAD MEHROTRA and SHANTANU SHARMA, University of California, Irvine, USA
JEFFREY D. ULLMAN, Stanford University, USA
DHRUBAJYOTI GHOSH, PEEYUSH GUPTA, and ANURAG MISHRA, University of California, Irvine, USA

Despite extensive research on cryptography, secure and efficient query processing over outsourced data remains an open challenge. This article continues along with 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. We first provide a new security definition, entitled *partitioned data security*, for guaranteeing that the joint processing of non-sensitive data (in cleartext) and sensitive data (in encrypted form) does not lead to any leakage. Then, this article proposes a new secure approach, entitled *query binning* (QB), that allows secure execution of queries over non-sensitive and sensitive parts of the data. QB maps a query to a set of queries over the sensitive and non-sensitive data in a way that no leakage will occur due to the joint processing over sensitive and non-sensitive data. In particular, we propose secure algorithms for selection, range, and join queries to be executed over encrypted sensitive and cleartext non-sensitive datasets. Interestingly, in addition to improving 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 past two decades have witnessed the emergence of several public clouds (e.g., Amazon Web Services, Google APP Engine, and Microsoft Azure) as the dominant computation, storage, and data management platform. Many small to medium size organizations, including some large organizations such as Netflix, have adopted the cloud model shifting their data management task to the cloud. The cloud offers numerous advantages including the economy of scale, low barriers to entry, limitless scalability, and a pay-as-you-go model. While benefits abound, a key challenge from data owners' perspective—that of "losing" control over one's data—still plagues the cloud model. In addition, the threat of "insider attacks" is also realistic, loss of control can lead to significant security, privacy, and confidentiality concerns. Such concerns are not a new revelation —indeed, they were identified as a key impediment for organizations adopting the *database-as-a-service* model in early work on data outsourcing [36]. Since then, the security/confidentiality challenge has been extensively studied in both the cryptography and database literature. Existing work on data security can be broadly categorized into the following three categories:

(1) **Encryption-based techniques.** These techniques include order-preserving encryption (OPE) [5], deterministic encryption [12], non-deterministic encryption [32], homomorphic encryption [29], bucketization [36], searchable encryption [18, 68], and distributed searchable symmetric encryption (DSSE) [40]). In addition, encryption-based techniques resulted in the following secure database systems: CryptDB [61], Monomi [72], TrustedDB [11], CorrectDB [10], SDB [78], ZeroDB [23], L-EncDB [46], MrCrypt [70], Crypsis [69], and Arx [60]. Likewise, Cypherbase [6], Microsoft Always Encrypted, Oracle 12c, Amazon Aurora [1], and MariaDB [2] are industrial secure encrypted databases.

(2) **Secret-sharing (SS)-based [65] techniques.** Examples of which include distributed point function [30], function secret sharing [13], accumulating-automata [21, 22], secure secret-shared MapReduce [22], OBSCURE [35], and others [25, 47, 51]. In this category, two emerging industrial systems are: *Pulsar* [1] based on function secret-sharing and *Jana* [9] based on non-deterministic, order-preserving encryption and secret-sharing.

(3) **Trusted hardware-based techniques.** They are either based on a secure coprocessor or Intel Software Guard Extensions (SGX) [17] that allow decrypting data in a secure area at the cloud and perform the computation on decrypted data. However, the secure coprocessor reveals access-patterns. Cipherbase [6, 7] CorrectDB [10], VC3 [64], Opaque [76], HardIDX [28], EnclaveDB [62], Oblix [54], Hermetic [79], and EncDBDB [27] are secure hardware-based systems.

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. Existing solutions suffer from the following limitations:

- **Non-scalability.** Cryptographic approaches that prevent leakage, e.g., fully homomorphic encryption coupled with oblivious-RAM (ORAM) [31, 58] or secret-sharing, simply do not scale to large data sets and complex queries. Most of the above-mentioned techniques do not work well, when deployed on a large-scale dataset, due to the high overheads of the techniques. For example, executing a selection query on 1M TPC-H LineItem table took (i) 22 s using secret-sharing-based techniques [22], (ii) 797 s using multiparty computation-based industrial database system, namely, Jana [9], and (iii) 3 s using Intel SGX-based Opaque [82], while executing the same query on cleartext data took only 0.0002 s.

- **Unclear security properties.** Systems such as CryptDB have tried to take a more practical approach by allowing users to explore the tradeoffs between the system functionality and the security it offers. Unfortunately, precisely characterizing the security offered by such systems given the underlying cryptographic approaches has turned out to be extremely difficult. For instance, References [42, 55] show that when order-preserving and deterministic encryption techniques are used together, on a dataset (in which the entropy of the values is not high enough), an attacker might be able to construct the entire plaintext by doing a frequency analysis of the encrypted data.

- **Vulnerability to other security attacks.** Many of the above-mentioned cryptographic techniques/systems are also susceptible to the following attacks:

  (1) *Output-size attack*: An adversary having some background knowledge can deduce the full/partial outputs by simply observing the output sizes [82]. All the above techniques/systems, except bucketization [37], are prone to output-size attacks.

  (2) *Frequency attack*: An adversary can deduce how many tuples have an identical value [55] based on the output of the query. Order-preserving encryption [5], searchable encryption [68], and secure hardware-based techniques [6, 7, 10, 20, 62, 64, 71, 82] are prone to frequency-count attacks, during query execution. In addition, deterministic encryption [12] reveals the frequency of a word, even without executing a query.

  (3) *Access-pattern attack*: An adversary can know the addresses of encrypted tuples that satisfy the query [16]. Private information retrieval (PIR) [16], oblivious-RAM (ORAM) [31], oblivious transfers [39, 63], and secret-sharing-based techniques [13, 21, 25, 30, 43, 51, 65, 75] are not prone to access-pattern attacks.

  (4) *Workload-skew attack*: An adversary, having the knowledge of frequent selection queries by observing many queries, can estimate which encrypted tuples potentially satisfy the frequent selection queries. Except access-pattern-hiding techniques, all the cryptographic techniques are prone to workload-skew attack (we will discuss workload-skew attack in detail in Section 5.5).
  To the best of our knowledge, there is no cryptographic technique that prevents all the four attacks on a *skewed dataset*.

Our contribution. While the race to develop cryptographic solutions that (i) are efficient, (ii) support complex SQL queries, (iii) offer provable security from the application's perspective is ongoing, this article departs from the above well-trodden path by exploring a *radically different (but complementary) approach to secure data processing in the cloud*. Our approach is intended for situations when only part of the data is sensitive, while the remainder (that may consist of the majority) is non-sensitive.

In particular, we propose a **partitioned computation model** that exploits such a classification of data into sensitive/non-sensitive subsets to develop efficient data processing solutions with **provable security guarantees**. In partitioned computing, sensitive data is outsourced in an appropriate encrypted form, while non-sensitive data can be outsourced in cleartext form. Partitioned computing, potentially, provides significant benefits by (i) avoiding (expensive) cryptographic operations on non-sensitive data, and (ii) allowing query processing on non-sensitive data to exploit indices. Such indices (that cannot be easily supported alongside encryption-based mechanisms in a non-interactive setting) are a key mechanism for efficient query processing in traditional database systems.[2]

While partitioned computing offers new opportunities for efficient and secure data processing on the cloud, it raises a new security challenge (Section 2)—of leakage, due to the joint processing of the encrypted (sensitive) dataset and of the plaintext (non-sensitive) datasets. We refer to this security challenge as **partitioned data security** challenge. Our work will formalize the security definition that will drive the development of the proposed prototype, entitled PANDA (to refer to *PArtitioN DatA*). In PANDA, we develop a query processing technique, entitled *query binning* (QB) to prevent leakage of sensitive data due to the simultaneous execution of sensitive and non-sensitive data. In addition, PANDA extends QB to answer selection, range, and join queries. We will, also, show two interesting effects of using QB:

(1) By avoiding cryptographic processing on non-sensitive data, *the joint cost of communication and computation of PANDA 's QB is significantly less than the computation cost of a strongly secure cryptographic technique* [3] (e.g., homomorphic encryptions, or secret-sharing-based technique [22, 35] that hides access-patterns—the identity of the tuple satisfying the query) on the entire encrypted data; and hence, QB improves the performance of strong cryptographic techniques over a large-scale dataset (Section 6).

(2) PANDA's QB provides *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 (Section 5.6).

Outline. The primary contributions of this article and its outline are as follows:

(1) The partition computation model and inference attack due to the joint processing over sensitive and non-sensitive data (Section 2).

(2)   A formal definition of *partitioned data security* when jointly processing sensitive and non-sensitive data (Section 3).

(3)   An efficient QB approach (Section 4) that guarantees partitioned data security, supporting cloud-side-indexes, and that can be built on top of any cryptographic technique.

(4)   Methods to deal with join queries, range queries, insert operation, and workload-skew attacks (Section 5).

(5)   A weak cryptographic technique (e.g., cloud-side indexable techniques [24, 60, 66]) becomes secure and efficient when mixed with QB (Section 5.6).

(6)   Experimental evaluation of PANDA under different settings and queries (Section 6).

Conference version. A preliminary version of this article was accepted and presented in IEEE ICDE [53]. The conference version includes the following additional concept, which is not provided in this version, due to space restriction: an analytical model to show when QB works better compared to a pure cryptographic technique (Section V A of Reference [53]).

2 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 the system and adversarial models under which we will develop our solutions. Table 1 enlists notations used in this article.

Table 1. Notations Used in the Article

Notations	Meaning
$\|S\|$	Number of sensitive data values
$\|NS\|$	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	ith sensitive and jth non-sensitive values
SB	The number of sensitive bins
SB_i	ith sensitive bin
$\|SB\| = y$	Sensitive values in a sensitive bin or the size of a sensitive bin
NSB	The number of non-sensitive bins
NSB_i	ith non-sensitive bin
$\|NSB\| = x$	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 cleartext 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 $W = W_s \cup W_{ns}$
$E(t_i)$	ith encrypted tuple

2.1 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. 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 [32] 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 using the proposed query binning (QB) algorithm, (on receiving a query from a user). The size of metadata is smaller than the size of the original data. The DB owner is assumed to have sufficient storage for such metadata, and also computational capabilities to execute QB algorithm, encryption (of queries keywords) and decryption (of the results).

Note. The tasks at the DB owner for metadata data storage and QB algorithm execution (which requires to execute Algorithms 1 and 2, will be explained in Section 4) could, potentially, be executed at the cloud, if the cloud supports a trusted hardware, e.g., SGX. However, using SGX for QB is nontrivial, since, now, the entire dataset needs to be encrypted and send to SGX that will decrypt and execute QB. In contrast, the task of rewriting the queries (sent by the DB owner) based on bin information, i.e., Algorithm 2, is relatively simple and can be done at SGX hosted by the cloud.

In addition, proxy reencryption in our setting is complex, since the results/answers to the query may have additional outputs, which will need to be filtered out at the trusted side. Thus, solutions based on proxy reencryption will not work in our settings.

(2)   **The untrusted public cloud** that stores the databases, executes queries, and provides answers to the DB owner.

Query execution. 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:

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

Let us illustrate partitioned computations through an example.

Example 1. Consider an *Employee* relation, see Figure 1. Note that the notation t_i ($1 \leq i \leq 8$) is not an attribute of the relation; we used this to indicate the ith tuple. 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 2(a)); (ii) *Employee2* with attributes *EId*, *FirstName*, *LastName*, *Office#*, and *Department*, where *Department* = "Defense" (see Figure 2(b)); and (iii) *Employee3* with attributes *EId*, *FirstName*, *LastName*, *Office#*, and *Department*, where *Department* <> "Defense" (see Figure 2(c)). Since the relations *Employee1* and *Employee2* (Figures 2(a) and 2(b)) contain only sensitive data, these two relations are encrypted before outsourcing, while *Employee3* (Figure 2(c)), which contains only non-sensitive data, is outsourced in cleartext. 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.

	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

Fig. 1. A relation: *Employee*.

	EId	SSN
	E101	111
	E259	222
	E199	333
	E152	444
	E254	555
	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*.

Fig. 2. Three relations obtained from *Employee* relation.

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.

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

2.2 Adversarial Model

We assume an honest-but-curious adversary that is *not trustworthy*. The honest-but-curious adversary is considered widely in the standard database-as-a-service query processing model, keyword searches, and join processing [14, 73, 74, 80, 81]. 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 *Employee1* relation (refer to Figure 2(c)).

(2) *Auxiliary/background* information of the sensitive data. The auxiliary information may contain metadata, schema of the relation, and the number of tuples in the relation (note that having an adversary with the auxiliary information is also considered in literature [42, 55]). 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; Figures 2(a) and 2(b). 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, denoted by AV: $AV = In_c \cup Op_c$, where In_c refers to the query arrives at the cloud and Op_c refers to the encrypted and non-encrypted tuples, transmitted in response to In_c. For example, the first row of Table 2 shows an adversarial view that shows that $Op_c = t_2$ tuples from the non-sensitive relation and encrypted $Op_c = t_4$ tuples from the sensitive relation are returned to answer the query for $In_c = $ E259.

(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.

Table 2. Queries and Returned Tuples/Adversarial View

Query value	Returned tuples/Adversarial view	
	Employee 2	**Employee 3**
E259	$E(t_4)$	t_2
E101	$E(t_1)$	null
E199	null	t_3

2.3 Inference Attacks in Partitioned Computations

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

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 2, where $E(t_i)$ denotes an encrypted tuple t_i.

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 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.

2.4 The Query Binning (QB) Approach: An Overview

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 (Section 3), 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 cleartext 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 Section 4.

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 3 illustrates the generated adversarial view when QB is used to answer queries as shown in Example 2. In this example, row 1 of Table 3 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.

Table 3. Queries and Returned Tuples/Adversarial View, Following QB

Query value	Returned tuples/Adversarial view	
	Employee 2	**Employee 3**
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

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 cleartext. Consequently, *the adversary is unable to find an intersection of the two sets, which is the exact value*.[6]

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*.

3 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]

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 $\overset{a}{=}$), 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 2(b)) is associated with the value of the attribute Eid in tuple t_2 (of *Employee3*, see Figure 2(c)), 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., $t_i[A] \overset{a}{=} t_j[A]$). 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 $v_i \overset{r}{\sim} v_j$. Note that $\overset{r}{\sim}$ can be one of $<, =$, or $>$ relationships. For instance, in Example 1, the E101 $\overset{r}{\sim}$ E259 corresponds to $=$, since both values have exactly one sensitive tuple (see Figure 2(b)), while E101 $\overset{r}{\sim}$ E199 is $>$, since there is one sensitive tuple with value E101 while there is no sensitive tuple with value 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. Before that, we wish to mention the need of a new security definition in our context. The inference attack in the partitioned computing can be considered to be related to the known-plaintext attack (KPA), wherein the adversary knows some plaintext data that is hidden

in a set of ciphertext. In KPA, the adversary's goal is to determine which ciphertext data is related to a given plaintext, i.e., determining a mapping between ciphertext and the corresponding plaintext data representing the same value. In our setup, non-sensitive values are visible to the adversary in plaintext. However, the attacks are different, since, unlike the case of KPA, in our setup, the ciphertext data might not contain any data value that is the same as some non-sensitive data visible to the adversary in plaintext.[8]

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) $Pr_{Adv}[e_i \overset{a}{=} ns_j | X] = Pr_{Adv}[e_i \overset{a}{=} ns_j | X, AV]$, where $e_i = E(t_i)[A]$ is the encrypted representation for the attribute 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) $Pr_{Adv}[v_i \overset{r}{\sim} v_j | X] = Pr_{Adv}[v_i \overset{r}{\sim} v_j | X, AV]$, for all $v_i, v_i \in Domain(A)$.

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. Satisfying this condition also prevents us in achieving success against KPA. 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, 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.

4 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) 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 Section 4.1 by developing the approach for the case when a sensitive tuple is associated with at most one non-sensitive tuple (Algorithm 1).

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

4.1 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., $\overset{a}{=}$ 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, $s_i \overset{a}{=} ns_j$ represents $E(t_i)[A] \overset{a}{=} ns_j$.

The scenario depicted in Example 1 satisfies the base case. The *EId* attribute values corresponding to sensitive tuples include ⟨ E101, E259 ,E152, E159⟩ and corresponding to non-sensitive tuples are ⟨ E199, E259, E254, E152 ⟩ for which $\overset{a}{=}$ is 1:1. We discuss QB under the above assumption, but these assumptions are relaxed in the conference version of this article (please see Sections IV.A and IV.B in Reference [53]). 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 x values. Note that we are assuming that $|S| \leq |NS|$.[9]

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] To assign sensitive values to sensitive bins, QB takes the ith sensitive value and assigns it to the $(i \bmod x)$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 the values therein from 0 to $x - 1$. To assign non-sensitive values, QB takes a sensitive bin, say j, and its ith sensitive value. Assign the non-sensitive value associated with the ith sensitive value to the jth position of the ith 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 values to size x (Line 7 of Algorithm 1).

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

Variable: $|NSB|$: non-sensitive values in a non-sensitive bin, $|SB|$: sensitive values in a sensitive bin.

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** $SB[i \bmod x \times][i^*] \leftarrow S[i]$
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
9   **Function** allocateNS($SB[i][j]$) **begin**
     find a non-sensitive value associated with the jth sensitive value of the ith sensitive bin

---

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 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 cleartext data processing; hence, by searching and retrieving fewer sensitive tuples, we decrease the encrypted data-processing time.

ALGORITHM 2: Bin-retrieval algorithm.

Inputs: w: the query value. SB and NSB: Sensitive and non-sensitive bins, created by Algorithm ??.

Outputs: SB_x and NSB_x: 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** $(i, j) \in (0, SB - 1), (0, |SB| - 1)$ **do**
3   **if** $w = SB_i[j]$ **then**
4   **return** SB_i and NSB_j; *found* $\leftarrow$ **true**; **break**
5   **if** *found* $\neq$ **true then**
6   **for** $(i, j) \in (0, NSB - 1), (0, |NSB| - 1)$ **do**
7   **if** $w = NSB_i[j]$ **then**
8   **return** NSB_i and SB_j; **break**
9   Retrieve the desired tuples from the cloud by sending encrypted values of the bin SB_i (or SB_j) and cleartext values of the bin NSB_j (or NSB_i) to the cloud

---

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 5 of Algorithm 2). If the value exists in a sensitive bin and a non-sensitive bin, then the DB owner retrieves the corresponding two bins (see Line 9). 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 cleartext. 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.
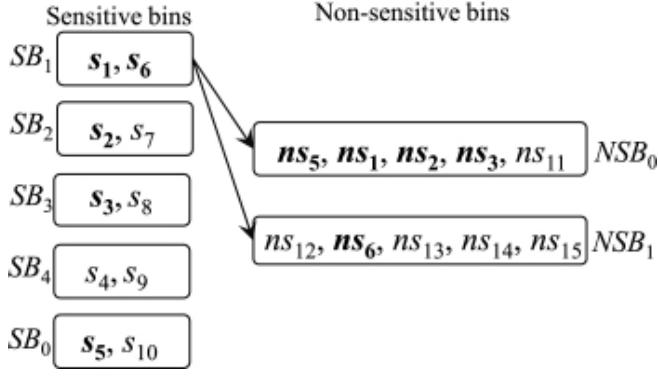

Sensitive bins		Non-sensitive bins

SB_1 : s_1, s_6
SB_2 : s_2, s_7 $ns_5, ns_1, ns_2, ns_3, ns_{11}$: NSB_0
SB_3 : s_3, s_8 $ns_{12}, ns_6, ns_{13}, ns_{14}, ns_{15}$: NSB_1
SB_4 : s_4, s_9
SB_5 : s_5, s_{10}

Fig. 3. QB for ten sensitive and ten non-sensitive values.

(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. To prevent such leakage, Algorithm 2 follows two rules stated below (see Lines 4 and 8 of Algorithm 2):

Tuple retrieval rule R1. If the query value w is a sensitive value that is at the jth position of the ith sensitive bin (i.e., $w = SB_i[j]$), then the DB owner will fetch the ith sensitive and the jth non-sensitive bins (see Line 4 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 jth position of the ith non-sensitive bin, then the DB owner will fetch the ith non-sensitive and the jth sensitive bins (see Line 8 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 cleartext to the cloud. The tuple retrieval based on the encrypted values reveals only the tuple addresses that satisfy the requested values. One can also hide the access-patterns by using PIR, ORAM, or DSSE to reach a required sensitive value. As mentioned in Section 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.[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.

Example 3: QB example Step 1: Bin Creation. We show the bin-creation algorithm for ten sensitive values and ten 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 five sensitive (say, $s_4, s_7, s_8, \ldots s_{10}$) and five 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 two non-sensitive bins and five sensitive bins, and divides ten sensitive values over the following five 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 five 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 four non-sensitive values such as $\{ns_{12}, ns_6, ns_{13}, ns_{14}, ns_{15}\}$.

Example 3: QB example (continued) Step 2: Bin-retrieval. 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 4, which shows that the adversarial view is not enough to leak information from the joint processing of sensitive and non-sensitive data, unlike Example 2. In Table 4, $E(s_i)$ shows the encrypted value of s_i, and we are showing the adversarial view only for queries for s_2, s_7, and ns_{13}. One may easily create the adversarial view for other queries. In this example, note that the bin SB_2 gets associated with both the non-sensitive bins NSB_0 and NSB_1, due to following Algorithm 2.

Table 4. Queries and Returned Tuples/Adversarial View After Retrieving Tuples According to Algorithm 2

Exact query value	Returned tuples/Adversarial view	
	Sensitive bin and data	**Non-sensitive bin and data**
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_7	$SB_2 : E(s_2), E(s_7)$	$NSB_1 : ns_6, ns_{12}, ns_{13}, ns_{14}, ns_{15}$
ns_{13}	$SB_2 : E(s_2), E(s_7)$	$NSB_1 : ns_6, ns_{12}, ns_{13}, ns_{14}, ns_{15}$

4.2 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 ten sensitive and ten 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 five nodes on one side and two nodes on the other side, see Figure 4(a). Note that since bins contain values, the surviving matches of the bins can lead to the surviving matches of the values. Hence, from Figure 4(a), the adversary can also create a bipartite graph for ten sensitive and ten non-sensitive values.
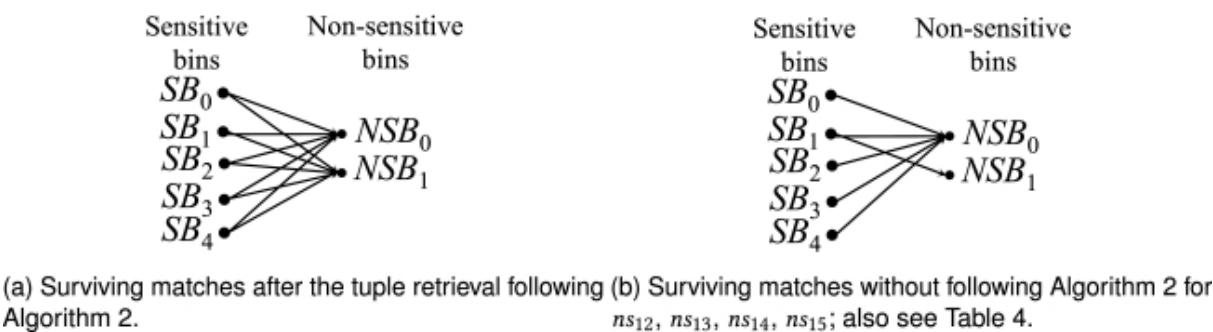

(a) Surviving matches after the tuple retrieval following Algorithm 2.   (b) Surviving matches without following Algorithm 2 for $ns_{12}, ns_{13}, ns_{14}, ns_{15}$; also see Table 4.

Fig. 4. An example to show security of QB using surviving matches for ten sensitive and ten 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 4 or 8 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 a adversarial view as shown in the first six lines except the fourth line of Table 4. 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 5. We show the case when NSB_1 is only associated with bin SB_1, and bins SB_2 is only associated with bin NSB_0, since Algorithm 2 is not followed.

Table 5. Queries and Returned Tuples/Adversarial View without Following Algorithm 2

Exact query value	Returned tuples/Adversarial view	
	Sensitive bin and data	**Non-sensitive bin and data**
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_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	$SB_2 : E(s_2), E(s_7)$	$NSB_0 : ns_1, ns_2, ns_3, ns_5, ns_{11}$
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}	$SB_1 : E(s_1), E(s_6)$	$NSB_1 : ns_6, ns_{12}, ns_{13}, ns_{14}, ns_{15}$

Having such an adversarial view (Table 5), the adversary can learn the following two facts:

(1) Encrypted sensitive tuples of the bin SB_2 have associated non-sensitive tuples only in the bin NSB_0, not in NSB_1 (Figure [4](b)).

(2) Non-sensitive tuples of the bin NSB_1 have their associated sensitive tuples only in the bin SB_1 (see Figure [4](b)).

Based on this adversarial view (Table [5]), the bipartite graph drops some surviving matches of the bins (see Figure [4](b)). (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 4 or 8 of Algorithm [2] for retrieving values that are not associated, then the above-mentioned facts (1) and (2) no longer hold. Figure [4](a) 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 ten sensitive and ten non-sensitive values, QB (Algorithms [1] and [2]) is secure, and under the given assumptions (Section [3]), the adversary cannot find an exact association between a sensitive and a non-sensitive value.

Security Proof

Now, we prove that QB is secure and satisfies the definition of partitioned data security (Theorem [4.2]) by first proving that all the sensitive bins are associated with all the non-sensitive bins (Theorem [4.1]), 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 4.1. 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 ($i \bmod x$)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 ($i \bmod x$)th position of the zth 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_a 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 jth position, by following the rule R2, the bin NSB_z and the 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., $Pr_{adv}[s_i \overset{\sim}{\leftarrow} s_j|X] = Pr_{adv}[s_i \overset{\sim}{\leftarrow} s_j|X, q(w)(R_s, R_{ns})[A]]$); recall that here in the base case, we assumed that a value has only a single sensitive tuple; hence, the condition holds true.

Theorem 4.2 (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 ith sensitive tuple value in the attribute A of the relation R_s and ns_j is the jth non-sensitive value in the attribute A of the relation R_{ns}. The 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 cleartext relation, say R_{ns}, containing four cleartext tuples with values v_1, v_2, v_3, v_4. The objective of the adversary is to deduce a cleartext value corresponding to an encrypted value. Note that before executing a query, the probability of an encrypted value, say E_i, to have the cleartext 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 cleartext relation R_{ns} for v_1, v_2. After executing the queries, the adversary holds an adversarial view given in Table [6].

Table 6. Queries and Returned Tuples/Adversarial View After Executing a Query for v_1, by Following Algorithm [2]

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

In this example, we show that the probability of finding the cleartext value of an encrypted representation, say E_i, $1 \leq i \leq 4$, remains identical before and after a query. 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:

(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 cleartext 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 cleartext 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 cleartext 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 cleartext 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 cleartext value of E_1, the adversary also considers the cleartext 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 cleartext 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 cleartext 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 cleartext 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 cleartext of an encrypted value is $\frac{(n-1)! - x}{n \times ((n-1)! - x)} = \frac{1}{n}$.

Note: Handling adaptive adversaries. The above-presented approach can handle an honest-but-curious adversary, who cannot execute any query, and the case when only the DB owner executes the queries on the databases. Now, we show how to handle an adaptive adversary that can execute queries on the database based on the result of previously selected queries. Note that an adaptive adversary can use any bin structure to break QB. She may ask some queries on the non-sensitive data and some queries on the sensitive data. Her objective is to find a value that is common in sensitive and non-sensitive datasets.

We explain with the help of an example that shows how an adaptive adversary breaks QB. Consider four sensitive tuples having sensitive value, say $s_1, s_2, \ldots, s_4$, and four non-sensitive tuples having non-sensitive values, say $ns_1, ns_2, \ldots, ns_4$. Suppose that s_i is associated with ns_i, and all sensitive tuples are encrypted. A correct bin structure (not considering permuted sensitive values) will be as follows: SB_1: $\{s_1, s_3\}$, SB_0: $\{s_2, s_4\}$, NSB_1: $\{ns_1, ns_2\}$, and NSB_0: $\{ns_3, ns_4\}$.

Now, first see how an adaptive adversary can break QB, with the help of two queries: Consider the first query for ns_1. The adversary can ask the query for ns_1, ns_2, s_1, and s_2. The adversary will learn that the first and second encrypted tuples are returned. However, she cannot know which of the tuple has an encrypted representation of s_1.

Another query is for ns_3, and she asks for ns_1, ns_3, s_1, and s_3. The adversary will learn that the first and third encrypted tuples are returned. However, now, she will know that the first encrypted tuple has the encrypted representation is s_1, because it was retrieved in the first query for ns_1 as well as in the second query.[12] Thus, by observing access-patterns, the adversary can know which two tuples are associated.

To protect this attack, we need to use a cryptographic technique, e.g., ORAM or secret-sharing that hides access-patterns at the sensitive data. When using access-patterns-hiding cryptographic techniques, the adversary will learn only the fact that two tuples are returned in response to a query. But it will not lead to any inference attacks. It is important to recall that access-patterns-hiding cryptographic techniques are prone to output size attacks. Thus, when mixing these techniques with QB makes them secure against output-size attacks. Note that we cannot use SGX-based solutions at the encrypted data when dealing with an adaptive adversary, because the adversary can observe access-patterns due to cache-lines and branch shadowing [33, 77].

Note: Security offered by existing cryptographic techniques vs QB. Papers such as [15, 34, 41, 42, 55] have illustrated that formal security guarantees (e.g., as often shown in papers such as property preserving encryption [5, 12, 61] and symmetric searchable encryption [18]) does not prevent leakage through inferences. For instance, Naveed et al. [55] showed that a cryptographically secured database that is also using an order-preserving cryptographic technique (e.g., order-preserving encryption (OPE)) may reveal the entire data when mixed with publicly known databases. Note that in our setting, the proposed technique, where the non-sensitive data resides in cleartext, offers almost no security without query binning. In particular, if the cryptographic technique used to store sensitive data reveals access-patterns, then the adversary will learn about which ciphertext corresponds to which keyword simply by observing the queries on cleartext. Such inferences are prevented by query binning. Also, note that unlike the security properties of searchable encryption techniques (e.g., OPE, deterministic encryption, and symmetric searchable encryption), which formalize security as indistinguishability from chosen keyword attack (IND-CKA1) [18] other than what can be inferred from the permitted leakages, our scheme does not lead to any leakage due to the joint processing of sensitive and non-sensitive datasets. Thus, QB is safe from inference attacks, and using QB in conjunction with any cryptographic technique does not lead to any additional leakages.

4.3 A Simple Extension of the Base Case

Algorithm 1 creates bins when the number of non-sensitive data values[13] is not a prime number, by finding the two approximately square factors. However, Algorithm 1 may exhibit a relatively 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 data 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.

ALGORITHM 3: An extension to the bin-creation Algorithm ?? 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      $z \leftarrow closest\_SquareNum(|NS|)$, $cost_{in} \leftarrow 2(z/\sqrt{z})$
5   **if** $(cost_{in} + \lceil (|NS| - z)/\sqrt{z} \rceil < cost_d)$ **then**
6   Execute Algorithm ??(S, z) and add $(NS - z)/\sqrt{z}$ number of the *remaining* non-sensitive values in each non-sensitive bins
7   **else** Execute Algorithm ??(S, NS)

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 nine non-sensitive bins and nine 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 five values and a non-sensitive bin holds at most ten 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. 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).

4.4 General Case: Multiple Values with Multiple Tuples

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 ten sensitive and ten non-sensitive values to bins using Algorithm 1; see Figure 3. Assume that a sensitive value, say s_1, has 1,000 sensitive tuples and an associated non-sensitive value, say ns_1, has 2,000 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 $ns_1, ns_2, ns_3, ns_6, ns_{11}$); 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., 3,005) 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 1,000 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 [4.1] and [4.2] to be true.

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.

Example 6: (Illustrating ways to assign *sensitive* values to bins to minimize the addition of fake tuples). Consider nine sensitive values, say $s_1, s_2, \ldots, s_9$, having 10, 20, 30, 40, 50, 60, 70, 80, and 90 tuples, respectively.[15] There are multiple ways of assigning these values to three bins so that we need to add a minimum number of fake tuples to each bin. Figure [5] shows two different ways to assign these values to bins. Figure [5](b) shows the best way—to minimize the addition of fake encrypted tuples; hence minimizing the cost. However, bins in Figure [5](a) require us to add 180 and 90 fake encrypted tuples to the bins SB_0 and SB_1, respectively.

Fig. 5. An assignment of nine sensitive values to three bins.

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 nine 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). 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 $\langle SB_0, NSB_2 \rangle$.

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:
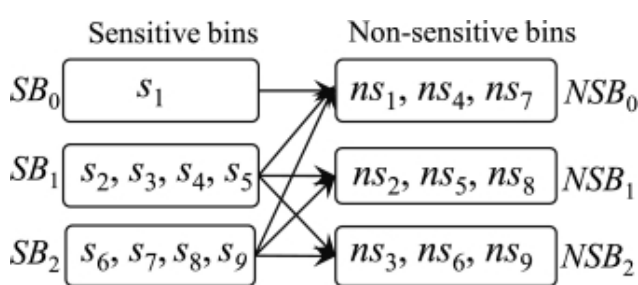

Fig. 6. An assignment of a heavy-hitter value but dropping surviving matches.

(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).

5 OTHER OPERATIONS

5.1 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 S_{ns}* . In order words, a sensitive tuple with a join key, say k_i, of the parent table R_s cannot have a non-sensitive tuple with the joining key k_i in the non-sensitive child table S_{ns}. However, a non-sensitive tuple with a join key, say k_j, of the parent table R_{ns} can have a sensitive tuple with the joining key k_j in the sensitive child table S_s. Thus, in the partitioned computing model, the primary-key-to-foreign-key join of R and S is computed as follows:

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

Table 7. Two Relations with Their Sensitive and Non-sensitive Tuples

	EID	Name
r_1	E101	Adam
r_2	E102	Bob
r_3	E103	John

(a) A relation R = Employee relation.

	EeID	Project Name
t_1	E101	Security
t_2	E102	Design
t_3	E103	Code
t_4	E103	Sale
t_5	E102	Sale

(b) A relation S = Project relation.

Let us understand it using an example of joining `Employee` and `Project` relations; shown in Tables [7](a) and [7](b). Assume that the row r_1 of Table [7](a) is sensitive, and rows t_1 and t_2 of Table [7](b) are sensitive. Based on this, we can partition those relations as shown in Table [8].

In this case, we will join r_1 of Table [8](a) with t_1 of Table [8](c), r_2 of Table [8](b) with t_2 of Table [8](c), as well as, with t_5 of Table [8](d), and r_3 of Table [8](b) with t_3, t_4 of Table [8](d).

Note that our objective is not to build a secure cryptographic technique for joining the sensitive relations. Thus, we use any existing cryptographic technique, e.g., CryptDB [61], SGX-based Opaque [82], [8], [59], or [22] to join sensitive relations. In addition, our objectives in joining two relations are:

(1) Hide which sensitive tuples (of the relation S_s) join with a non-sensitive tuple (of the relation R_{ns}). For example, we need to hide that t_2 of Tables [8](c) should join with r_2 of Table [8](b).

(2) 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 $R_{ns} \bowtie S_{ns}$. For example, we need to hide that r_2 of Table [8](b) should join with t_5 of Table [8](d).

The DB owner-side. To join, the relations R_{ns} and S_s, we follow the approach given in Reference [56] 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. For example, Table [9](a) shows the relation R_{ns} with row r_2 as pseudo-sensitive tuple (and Table [9](b) shows the non-sensitive relation). In Reference [56], 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 (see Algorithm [4]), 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 (Line 3 of Algorithm [4]). Now, the DB owner outsources (i) encrypted relations R_{ps} and S_s, and (ii) cleartext relations R_{ns}, S_{ns} (Line 5 of Algorithm [4]). Additionally, the DB owner maintains the information for bin-creation (Algorithm [1]), which can be used to retrieve tuples after join. Thus, in our case, the join of R and S is converted into the following join:

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

Table 8. Sensitive and Non-sensitive Relations Created from Two Relations of Table [7]

	EID	Name
r_1	E101	Adam

(a) R_s.

	EID	Name
r_2	E102	Bob
r_3	E103	John

(b) R_{ns}.

	EeID	Project Name
t_1	E101	Security
t_2	E102	Design

(c) S_s.

	EeID	Project Name
t_3	E103	Code
t_4	E103	Sale
t_5	E102	Sale

(d) S_{ns}.

Table 9. Sensitive Relation with Pseudosensitive Tuples and Non-sensitive Relation, Created from R_s = `Employee` Relation of Table [8](a)

	EID	Name
r_1	E101	Adam
r_2	E102	Bob

(a) R_{ps}.

	EID	Name
r_2	E102	Bob
r_3	E103	John

(b) R_{ns} same as Table 8(b).

The cloud-side. We use any cryptographic technique for $R_{ps} \bowtie S_s$, and of course, join of the relations R_{ns} and S_{ns} is carried out in the cleartext.

ALGORITHM 4: Algorithm for execution join queries.

Inputs: Two relations: $R(key, A_1, A_2, \cdots A_m)$ and $S(key, B_1, B_2, \cdots B_{m'})$
Outputs: $R \bowtie S$
DB owner

1   Create R_s, R_{ns}, S_s, and S_{ns}
2   $pseudo\_sensitive\_key[] \leftarrow \{key \in R_{ns} \mid \Pi_{key}(R_{ns}) \cap \Pi_{key}(S_s) \neq \emptyset\}$ // Retrieve all the keys in R_{ns} that joins with the relation S_s
3   $R_{ps} \leftarrow R_s \cup (\sigma_{key \in pseudo\_sensitive\_key[]}(R_{ns}))$ // Retrieving tuples from R_{ns} based on the the keys present in $pseudo\_sensitive\_key[]$ and merging them with the relation R_s
4   Encrypt R_{ps} and S_s
5   Outsource encrypted R_{ps}, encrypted S_s, cleartext R_{ns}, and cleartext S_{ns} to cloud
     Cloud
6   $join_{sensitive\_output} \leftarrow R_{ps} \bowtie S_s$, $join_{non\_sensitive\_output} \leftarrow R_{ns} \bowtie S_{ns}$
     DB owner
7   **if** $\sigma_{A_i = s_j}(join_{sensitive\_output} \cdot join_{non\_sensitive\_output})$ **then**
     Execute Algorithm 2 // If the user is interested to fetch a tuple having s_j in attribute A_i

Note: non-foreign-key joins. The above strategy can also be extended to non-foreign-key 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 the encrypted domain, since these tuples will also join in cleartext. It can be done if the DB owner can add an attribute to each sensitive relation to mark such pseudo-sensitive tuples.

5.2 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 answered using QB. For example, 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 7 shows a way to assign these values to bins.
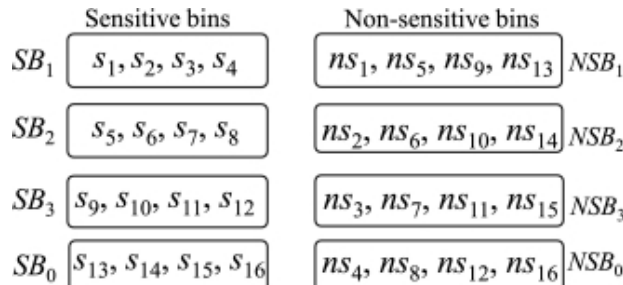
	Sensitive bins	Non-sensitive bins	
SB_1	s_1, s_2, s_3, s_4	$ns_1, ns_5, ns_9, ns_{13}$	NSB_1
SB_2	s_5, s_6, s_7, s_8	$ns_2, ns_6, ns_{10}, ns_{14}$	NSB_2
SB_3	$s_9, s_{10}, s_{11}, s_{12}$	$ns_3, ns_7, ns_{11}, ns_{15}$	NSB_3
SB_0	$s_{13}, s_{14}, s_{15}, s_{16}$	$ns_4, ns_8, ns_{12}, ns_{16}$	NSB_0

Fig. 7. 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 tuples' retrieval.

We describe a procedure for the case $|S| \leq |NS|$, as the restriction is followed by Algorithm 1 in Section 4.1. We use the example of 16 sensitive and 16 non-sensitive values of the attribute A. 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 [19, 37, 48, 49] 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[16] for the values of the attribute A of the non-sensitive relation; see Figure 8 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 four sensitive and four non-sensitive bins, by following Lines 3–7 of Algorithm 1.
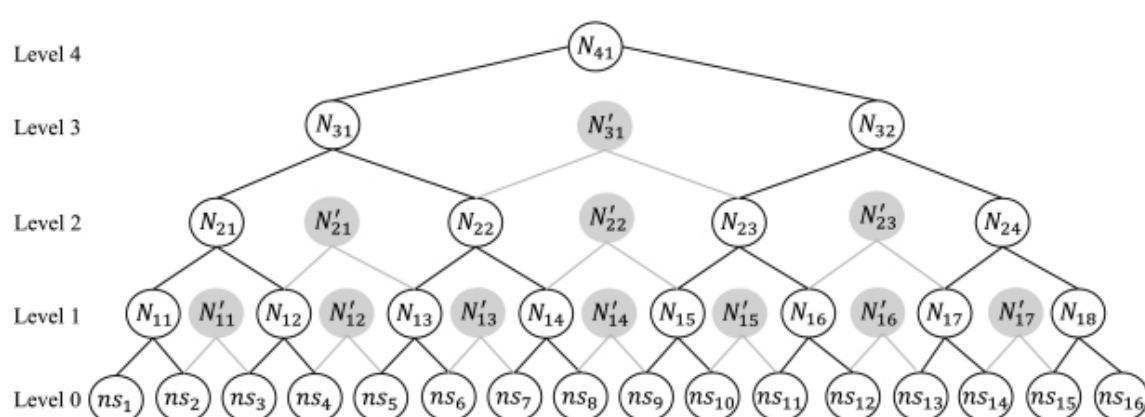
Fig. 8. A full binary tree with some additional nodes for 16 non-sensitive values.

At the level 1, Algorithm 1 takes eight inputs that represent the nodes ($N_{11}, N_{12}, \ldots, N_{18}$; see white nodes in Figure 8) 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 non-sensitive values ns_1, ns_2. For the eight values, Algorithm 1 provides four sensitive bins (each is containing eight values) and four sensitive bins (each is containing four values). Let NSB_{ij} be the jth non-sensitive bin at the ith level, and let SB_{ij} be the jth sensitive bin at the ith level. Thus, Algorithm 1 produces the following bins:

NSB_{10} containing $\langle N_{11}, N_{12}, \ldots, N_{14} \rangle$,

NSB_{11} containing $\langle N_{15}, N_{16}, \ldots, N_{18} \rangle$,

SB_{10} containing $\langle s_1, s_2, s_9, s_{10} \rangle$,

SB_{11} containing $\langle s_3, s_4, s_{11}, s_{12} \rangle$,

SB_{12} containing $\langle s_5, s_6, s_{13}, s_{14} \rangle$,

SB_{13} containing $\langle s_7, s_8, s_{15}, s_{16} \rangle$.

At level 2, Algorithm 1 takes four inputs that represent the nodes ($N_{21}, N_{22}, N_{23}, N_{24}$; see white nodes in Figure 8) 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 four input values, Algorithm 1 provides two non-sensitive bins (each is containing eight values) and two sensitive bins (each is containing eight values), as follows:

NSB_{20} containing $\langle N_{21}, N_{22} \rangle$,

NSB_{21} containing $\langle N_{23}, N_{24} \rangle$,

SB_{20} containing $s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}$,

SB_{21} 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 8) that become parent nodes of the lower level's two adjacent nodes that do not have a common parent node. The algorithm given in Reference [19] also uses these additional nodes for answering a range query. For example, at the level 2 in Figure 8, the DB owner creates seven such nodes. Let NSB'_{ij} be the jth non-sensitive bin at the ith level for these additional nodes, and let SB'_{ij} be the jth sensitive bin at the ith level for these additional nodes. Algorithm 1 takes these seven inputs and produces two non-sensitive bins (each is containing eight values) and four sensitive bins (each is containing eight values), as follows[17]:

NSB'_{10} containing $\langle N'_{11}, N'_{12}, \ldots, N'_{14} \rangle$,

NSB'_{11} containing $\langle N'_{15}, N'_{16}, N'_{17}, \text{two fake tuples} \rangle$,

SB'_{10} containing $\langle s_2, s_3, s_{10}, s_{11} \rangle$,

SB'_{11} containing $\langle s_4, s_5, s_{12}, s_{13} \rangle$,

SB'_{12} containing $\langle s_6, s_7, s_{14}, s_{15} \rangle$,

SB'_{13} containing $\langle s_8, s_9, \text{two fake tuples} \rangle$.

Note. The self-explainable pseudocode in Algorithm 5, given in Appendix A, describes all the above steps of binary-tree creation and bin-creation for range queries.

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. For example, if the query is for values ns_1 to ns_4, then by traversing the tree (see Figure 8) 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}. The self-explainable pseudocode in Algorithm 6, given in Appendix A, describes the best-match method for answering range queries.

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 the 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 a 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 four tuples in each; see Figure 7), one sensitive bin and non-sensitive bin for s_7 (containing four tuples in each; see Figure 7), and the bin NSB_{10} (containing eight tuples) and SB_{12} (containing four tuples) for answering the query for a range ns_5 to ns_6. However,

the second query ($Q2$) will be worse in terms of retrieving the tuples. It will result in the retrieval of the entire non-sensitive data (see Figure 7).

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 eight tuples), SB'_{11} (containing four tuples), SB'_{12} (containing four 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.

5.3 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 the DB owner can either update the existing bins (by increasing an identical size of each bin) or create all the new bins.

Particularly, the DB owner waits for new tuples until the DB owner collects new sensitive and non-sensitive values equals to the number of existing sensitive and non-sensitive bins, such that each bin receives a single value.[18] 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. Note that if the new values become a part of only one sensitive and one non-sensitive bin, it reveals an association of values. Further, note that in outsourcing new tuples, the DB owner sends appropriately encrypted sensitive data and cleartext non-sensitive data to the cloud. Thus, the cloud does not have access to any sensitive data in cleartext, and hence, cannot launch an attack, if it is an honest-but-curious adversary, i.e., a passive attacker, as we mentioned in Section 2.2.

However, the insertion of more values in existing bins incurs the overhead, as will be shown in Experiment 6 in Section 6. 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 non-sensitive data in cleartext.

(2) *Inserting new-S and new-NS.* The DB owner increases the size of each bin by one and inserts the values into existing bins.

(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 Lines 6 and 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 the 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.

5.4 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 query execution, the query processing algorithm will choose the corresponding attribute subset that is most beneficial (will result in the 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.

5.5 Handling Workload-skew Attack

The query execution and the corresponding accessed tuples (in the absence of an access-pattern-hiding scheme) allows an adversary to deduce the frequency of queries, without knowing the cleartext value of the query keyword. Such information coupled with background knowledge may reveal additional information to the adversary. For example, in Table 1, if many queries access tuples t_2 and t_4, then the adversary may deduce that these two tuples may have identical values in one or more attributes. Furthermore, if the adversary has background knowledge that John is the company's CEO (for which people ask many queries), then the adversary may deduce that t_2 and t_4 belong to John. We call such an attack as *workload-skew attack*. We illustrate the workload-skew-based attack (see Figure 9(a)) and also our approach for addressing it for the base case of QB. In our proposed solution, we consider that a query predicate is either highly frequent or infrequent. Our solution can be extended to the general case of different frequency query predicates by creating groups of query keywords based on their query frequencies and allocating such groups to non-sensitive data such that the number of values in each bin is equal.
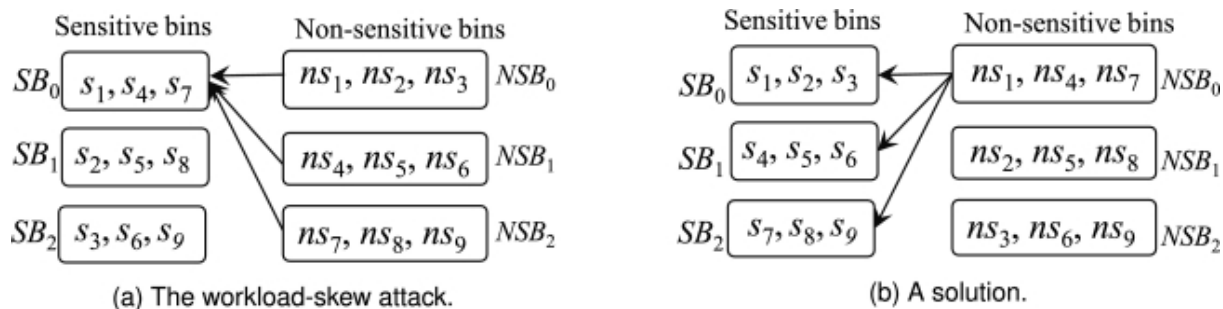
Sensitive bins	Non-sensitive bins		Sensitive bins	Non-sensitive bins
SB_0 s_1, s_4, s_7	ns_1, ns_2, ns_3 NSB_0		SB_0 s_1, s_2, s_3	ns_1, ns_4, ns_7 NSB_0
SB_1 s_2, s_5, s_8	ns_4, ns_5, ns_6 NSB_1		SB_1 s_4, s_5, s_6	ns_2, ns_5, ns_8 NSB_1
SB_2 s_3, s_6, s_9	ns_7, ns_8, ns_9 NSB_2		SB_2 s_7, s_8, s_9	ns_3, ns_6, ns_9 NSB_2
(a) The workload-skew attack.			(b) A solution.	

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

Figure 9(a) shows bins created by Algorithm 1 for nine sensitive values and their associated nine 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 nine 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.[19] To prevent the workload-skew attack, we need to allocate sensitive values carefully, thereby the sensitive values associated with frequent selection predicates are distributed over all the bins.

Below, we provide a strategy for handling the workload-skew attack in QB. The idea is to find groups of either $x (= |NSB|)$ non-sensitive query keywords or $y = |SB|$ sensitive query keywords that are queried almost equally, and then follow the given steps to create bins appropriately. The self-explainable pseudocode in Algorithm 7, given in Appendix A, and the following steps describe bin-creation method for the case of x frequent non-sensitive keywords. The similar method can be used for bin creation in the case of y frequent sensitive keywords.

Step 1: Bin-creation to deal with workload-skew attack. To deal with workload-skew attack, we, first, need to modify Algorithm 1 for bin-creation. The modified bin-creation algorithm (Algorithm 7, given in Appendix A) contains the following steps:

(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 (Line 3 of Algorithm 1).

(2) *Assign non-sensitive values.* Create groups, each of size x, of the frequent non-sensitive predicates, resulting in $u \leq NSB$ groups. Assign the ith group to the ith non-sensitive bin. Now, assign all the remaining non-sensitive values, if any, as follows: Find those non-sensitive bins that are not full, i.e., having less than x non-sensitive values. Then, assign the remaining non-sensitive values to all such non-sensitive bins, such that each non-sensitive bin should contain at most x non-sensitive values.

(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 jth sensitive bin at the zth position.

By following the above steps, Figure 9(b) shows three 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.[20]

Step 2: Bin-retrieval—answering queries. To retrieve tuples satisfying a query keyword, we use Algorithm 2.

5.6 Enhancing Security Levels of Indexable Techniques

We show how QB can be integrated with an indexable cryptographic technique, namely, Arx [60] that uses a non-deterministic encryption mechanism. In Arx, the DB owner stores each domain value v and the frequency of v in the database. The technique encrypts the ith 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. The query processing using Arx as efficient as cleartext version due to using an index.

The use of QB with Arx makes it secure against output-size, frequency-count, and workload-skew attacks. Of course, QB takes more time as compared to Arx, 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 index. In the worst case, we traverse the index at most $|SB|$ times, unlike Arx, which traverses the index only once for a single selection query. It, however, significantly enhances the security of Arx by preventing output size, frequency count, and workload-skew attacks. However, QB does not protect access-patterns being revealed, which could be prevented using ORAM. Determining whether coupling ORAM with Arx with QB or using a more secure cryptographic solution, e.g., secret-sharing, which uses a linear scan to prevent access-patterns, with QB, more efficient (while QB with both the solutions hiding the underlying cryptographic technique) is an open question.

6 EXPERIMENTAL EVALUATION

This section presents experimental evaluations of PANDA. As we mentioned that PANDA does not need any specific cryptographic technique for encrypted the data, we used PANDA with two cryptographic techniques/systems, such as SGX-based Opaque [82] and multi-party computations (MPC)-based Jana [9]. We installed Jana on a machine of 3.5 GHz, Intel Xeon 8-core processor, 64 GB RAM, 3 TB disk, and Opaque on a machine 3.5 GHz, Intel i7 4-core processor, 32 GB RAM with SGX support, 350 GB disk.[21] We used Order and LineItem tables of TPC-H benchmark. The DB owner stores sensitive and non-sensitive bins, whose size was propositional to the domain size of the searchable attributes and independent of the database size. For example, for LineItem table, metadata for attributes OrderKey, PartKey, and SuppKey were ≈3 MB, ≈1 MB and ≈0.1 MB, respectively, while the size of LineItem table having 12M rows was ≈1.5 GB.

In the following, we show our experimental results: (i) We evaluate PANDA's QB mixed with Jana on 1M LineItem for selection and range queries.[22] (ii) We evaluate PANDA's QB mixed with Opaque on 6M and 12M LineItem for selection, range, and join queries.

Exp 1: Selection query execution. Table 10 shows the time taken when using QB with Opaque and Jana at different levels of sensitivity. Without using QB for answering a selection query, Opaque [82] took 89 seconds on 6M rows of LineItem table and Jana [9] took 797 seconds on 1M rows of LineItem table.[23] Note that the time to execute the same query on cleartext data was only 0.0002 s. QB improves not only the performance of Opaque and Jana but also makes them to work securely on partitioned data and resilient to output-size attack.

Table 10. Exp 1: Time (in seconds) for Executing a Selection Query, When Mixing QB with Opaque and Jana at Different Levels of Sensitivity

Technique	20%	40%	60%	80%
MPC-based Jana [9] (1M)	168	318	481	661
SGX-based Opaque [82] (6M)	26	42	59	78

Exp 2: Range query execution. We executed a range query on OrderKey column of LineItem table. Opaque and Jana scan the entire data for answering any query. Thus, range query execution time was not impacted by the size of a range. We selected a range of length 400. Table 11 shows the time taken by a range query on different sensitivity datasets using QB mixed with Jana and Opaque. Note that the time taken by the same range query without using QB on 1M rows was 841 seconds using Jana, and was 124 seconds on 6M rows and 288 seconds on 12M rows using Opaque. It shows that though fetching more rows according to QB, it does not incur overheads in terms of the computation time.

Table 11. Exp 2: Time (in seconds) for Executing a Range Query, When Mixing QB with Opaque and Jana at Different Levels of Sensitivity

Technique	20%	40%	60%	80%
MPC-based Jana [9] (1M)	170	319	483	688
SGX-based Opaque [82] (6M)	25	50	74	98
SGX-based Opaque [82] (12M)	49	98	146	216

Exp 3: Join query execution. We executed a join query mixed with a selection predicate covering 400 OrderKey of Order and LineItem tables. We used Order table of 1.5M rows with 6M rows of LineItem table as smaller datasets, and Order table of 3M rows with 12M rows of LineItem table as larger datasets. Table 12 shows the time taken by a join query on different sensitivity datasets using QB mixed with Opaque. Note that the time taken by the same join query without using QB was 154 seconds on the smaller dataset and 364 seconds on the larger dataset using Opaque. We also tried to execute a join query without selection; however, Opaque does not support neither amount of rows.

Table 12. Exp 3: Time (in seconds) for Executing a Join Query, When Mixing QB with Opaque at Different Levels of Sensitivity

Technique	20%	40%	60%	80%
SGX-based Opaque [82] (6M)	51	77	102	129
SGX-based Opaque [82] (12M)	102	155	207	284

Exp 4: Impact of communication cost. Since QB fetches more data than the desired data, it may affect the overall performance. We fetched the maximum number of rows in the case of range queries. Particularly, in the case of 80% of 12M LineItem table, we fetched ≈70,000 rows, whose size was ≈14 MB. When using slow (100 MB/s), medium (500 MB/s), and fast (1 GB/s) speed of data transmission, the data transmission time was negligible.

Exp 5: Impact of bin size. Table 13 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 $|SB| \approx |NSB|$.

Table 13. Exp 5: Impact of Bin-size

$||SB| - |NSB||$	400	500	750	1,000
Time to execute a selection query on 20% dataset using Opaque	25	28	31	34

Exp 6: Impact of insert operation. In the experiment, insertions were processed (as per the method, given in Section 5.3) in batches of 10,000 and after each batch, selection queries were executed to determine the overhead due to insertion. Finally, after seven batches of insertion, Algorithm 1 was re-executed to recreate bins. Table 14 confirms that the query cost increases but only marginally in the presence of insertion and (as shown by the last column) reduces by re-binning. In Table 14, the first row shows the size of increasing data after each 10,000 rows' insertion and the second row shows the time in executing a selection on the dataset.

Table 14. Exp 6: Impact of Insert

#inserted rows	10,000	20,000	30,000	40,000	50,000	60,000	70,000	Re-bin
Time to execute a selection query on 20% dataset using Opaque	27	28	29	30	31	32	34	30

Exp 7. Number of fake tuples. Table 15 summarizes the number of fake tuples added for TPC-H LineItem data at different levels of sensitivity. The reason of decreasing fake tuples when increasing sensitivity is that more real tuples take place of the fake tuples. In general, the addition of fake tuples will adversely affect QB, especially, if data is skewed. However, as shown in Tables 10, 11, and 12, QB remains significantly better compared to fully cryptographic approaches at all levels of sensitivity despite fake tuples being added.

Table 15. Exp 7: Number of Fake Tuple Inserted Due to QB

LineItem entire size	1%	5%	20%	40%	60%
6M	34,244	34,048	29,568	24,024	22,736

7 CONCLUSION

This article proposes a prototype, PANDA, and its query processing technique, query binning (QB), that serves as a meta approach on top of existing cryptographic techniques to support secure selection, range, and join queries, when a relation is partitioned into cryptographically secure sensitive and cleartext non-sensitive sub-relations. Further, we develop a new notion of *partitioned data security* that restricts exposing sensitive information due to the joint processing of the sensitive and non-sensitive relations. Besides improving efficiency, while supporting partitioned security, interestingly, PANDA enhances the security of the underlying cryptographic technique by preventing size, frequency-count, and workload-skew attacks. As a result, combining PANDA's QB with efficient but non-secure cloud-side indexable cryptographic approaches can result in an efficient and significantly more secure search. Furthermore, existing cryptographic techniques that prevent access-patterns can also benefit from the added security that PANDA offers. In future, one may extend the proposed technique for answering queries in different situations, such as the cases of different relations that are encrypted using different cryptographic techniques and the case of a single relation that is vertically partitioned into multiple relations that are encrypted using different cryptographic techniques.

APPENDIX

A PSEUDOCODE

ALGORITHM 5: Bin-creation algorithm for range queries.

Inputs: S and NS

Outputs: $SB[*, *, *]$: Sensitive bins for each level of the tree. $NSB[*, *, *]$: non-sensitive bins for each level of the tree. In both $SB[*, *, *]$ and $NSB[*, *, *]$, the first index refers to the level of tree, and the second index refers to the bin identity.

Variable Initialization: An array $unique\_ns[] \leftarrow allocate\_distinct\_nonsensitive\_value(NS)$, $array\_tree\_node[]$: An array to store all the nodes at a particular level of the binary tree.

Tree Node Initialization: TreeNode($covered\_val[]$, TreeNode $left\_child$, TreeNode $right\_child$) // TreeNode is an object class representing a node of a binary tree, where $covered\_val[]$ refers to an array of values covered by the tree node, $left\_child$ refers to the pointer to the child tree node on the left, and $right\_child$ refers to the pointer to the child tree node on the right side of the tree node.

1   **Function** $create\_binary\_tree(unique\_ns[])$ **begin**
2   **if** $|unique\_ns[]| == 1$ **then**
     // This 'if' statement is the terminal condition of the recursive function execution and the condition for edge case where the number of elements in $unique\_ns[]$ is 1
3   TreeNode $tn \leftarrow allocate$(TreeNode())
4      $tn.covered\_val[0] \leftarrow unique\_ns[0]$
5   **return** tn
6    $length \leftarrow |unique\_ns[]|$ // Allocating the count of unique non-sensitive values to a temporary variable $length$.
7   TreeNode $root \leftarrow allocate$(TreeNode()) // Allocate an empty tree node.
8   **for** $i \in \langle 1, length \rangle$ **do** $root.covered\_val[i] \leftarrow unique\_ns[i]$
     // For a tree node, this statement stores all the values covered by the sub-tree rooted at this node
9    $root.left\_child \leftarrow create\_binary\_tree(unique\_ns[1, length/2])$ // Recursively calling $create\_binary\_tree()$ function with left half of $unique\_ns[]$ array.
10    $root.right\_child \leftarrow create\_binary\_tree(unique\_ns[length/2 + 1, length])$ // Recursively calling $create\_binary\_tree()$ function with right half of $unique\_ns[]$ array.
11   **return** $root$
12   **Function** $create\_sensitive\_bin$(TreeNode $root$, S, NS) // Allocating sensitive and non-sensitive values to bins.
   begin
13    $max\_level \leftarrow height(root)$ // Finding the height of the tree created by function $create\_binary\_tree()$
14   **for** $i \in \langle 1, max\_level - 2 \rangle$ **do**
     // This 'for loop' is used to create bins for each level of the tree, except the root node and child nodes of the root node. This 'for loop' starts from the leaf level of the tree.
15      $array\_tree\_node[] \leftarrow retrieveNodes(root, i)$; // $retrieveNodes(root, i)$ is a function that takes the 'root' node of the tree and the level i of the tree, and then, returns all nodes at the ith level of the tree.
16      $x, y \leftarrow approx\_sq\_factors(|array\_tree\_node[]|); x \geq y$
17      $|NSB| \leftarrow x, NSB \leftarrow \lceil |array\_tree\_node[]|/x \rceil, SB \leftarrow x, |SB| \leftarrow y$ // Sensitive and non-sensitive bin creation for the tree nodes of the ith-level of the tree
18   **for** $(j, k) \in \langle 1, NSB \rangle, \langle 1, |NSB| \rangle$ **do** $NSB[i][j][k] \leftarrow array\_tree\_node[(j-1) * |NSB| + k]$ // Allocating non-sensitive values to bins created for nodes of the ith-level of the tree.
19   **for** $(j, k) \in \langle 1, NSB \rangle, \langle 1, |NSB| \rangle$ **do** $SB[i][k][j] \leftarrow allocateS(NSB[i][j][k])$ // Allocating sensitive values to bins created for nodes of the ith-level of the tree.
20   **for** $j \in \langle 1, SB \rangle$ **do** $SB[j][*] \leftarrow$ fill the bin if empty with the size limit to y // Filling the sensitive bins for the ith level with the sensitive values that are non-associated with any non-sensitive value.
21   **return** $SB[i][*][*]$ and $NSB[i][*][*]$

ALGORITHM 6: Bin retrieval algorithm for range queries (best-match method) for non-sensitive values.

Inputs: $[\alpha, \beta]$: The range of query values, where $\beta > \alpha$ and $\alpha, \beta \in NS$. $Tree$: A binary tree created by Algorithm ??.

Outputs: SB: A sensitive bin; NSB: A non-sensitive bin.

Variable Initialization: $iterations \leftarrow height(Tree), found \leftarrow false$.

1   **Function** $retrieve\_bins\_binary\_tree(\alpha, \beta)$ **begin**
2   **for** $i \in \langle 2, iterations \rangle$ **do** // Traversing the tree from the parent nodes of the leaf node to the root node in a bottom-up fashion.
    do
3      $array\_tree\_node[] \leftarrow retrieveNodes(Tree.root, i)$ // $retrieveNodes(root, i)$ is a function that takes the 'root' node of the tree and the level i of the tree, and then, returns all nodes at the ith level of the tree.
4   **for** $j \in \langle 1, |array\_tree\_node[]| \rangle$ **do**
      // This 'for loop' iterates over the nodes of $array\_tree\_node[]$ array and determines if any node covers the range of $[\alpha, \beta]$ completely.
5       $len\_covered \leftarrow |array\_tree\_node[j].covered\_val[]|$
6   **if** $(array\_tree\_node[j].covered\_val[0] \leq \alpha \wedge array\_tree\_node[j].covered\_val[len\_covered] \geq \beta)$// If the range $[\alpha, \beta]$ is covered completely by $covered\_val[]$ array of the jth node at level i of the tree, which is stored in $array\_tree\_node[]$.
      then
7       $NSB.append(array\_tree\_node[j].covered\_val[*])$ // All the values covered by the node $array\_tree\_node[j]$ are added to the non-sensitive bin NSB.
8       $found \leftarrow true; level \leftarrow i; bin\_id \leftarrow j; Break$
9   **if** $found = true$ **then** $Break$
10   **return** $SB[level, bin\_id, *]$ and NSB

ALGORITHM 7: Bin-creation algorithm to deal with the workload-skew attack.

Inputs: NS: non-sensitive data, S: sensitive data.

Outputs: SB: sensitive bins; NSB: non-sensitive bins

Variable: $frequent\_non\_sensitive[]$: A set holding frequent non-sensitive keywords

1   **Function** $create\_bins\_under\_workload(S, NS)$ **begin**
2    $x, y \leftarrow approx\_sq\_factors(|NS|)$; $x \geq y$ // Finding approximately square factors of the count of unique non-sensitive values.
3    $|NSB| \leftarrow x, NSB \leftarrow \lceil |NS|/x \rceil, SB \leftarrow x, |SB| \leftarrow y$ // Creating sensitive and non-sensitive bins.
4    $frequent\_non\_sensitive[] \leftarrow find\_frequent(NS)$ // Find and allocate frequent non-sensitive values to the set $frequent\_non\_sensitive[]$.
5   **for** $(i, j) \in \langle 1, NSB \rangle, \langle 1, frequent\_non\_sensitive[] \rangle$ **do**
    $NSB[i][j] \leftarrow frequent\_non\_sensitive[(i-1) * |NSB| + j]$ // Allocating frequent non-sensitive query keywords to non-sensitive bins.
6    $remaining\_NS[] \leftarrow NS \setminus \{v \mid v \in frequent\_non\_sensitive[]\}$// The set $remaining\_NS[]$ contains all the infrequent non-sensitive values.
7   **for** $i \in \langle 1, NSB \rangle$ **do** $NSB[i][*] \leftarrow$ fill the bin if empty with the size limit to x by taking values from $remaining\_NS[]$ until $remaining\_NS[] = \emptyset$
8   **for** $(i, j) \in \langle 1, NSB \rangle, \langle 1, |NSB| \rangle$ **do** $SB[j][i] \leftarrow allocateS(NSB[i][j])$ // Assigning sensitive values associated with non-sensitive values.
9    $remaining\_S[] \leftarrow S \setminus \{v \mid v \in SB[*, *]\}$ // The set $remaining\_S[]$ contains all the sensitive values that are not associated with any non-sensitive values.
10   **for** $i \in \langle 1, SB \rangle$ **do** $SB[i, *] \leftarrow$ fill the bin if empty with the size limit to y by taking values from $remaining\_S[]$ until $remaining\_S[] = \emptyset$
11   **return** SB and NSB

REFERENCES

[1] [n.d.]. Amazon Aurora. Retrieved on September 20, 2020 from **https://aws.amazon.com/rds/aurora/ (https://aws.amazon.com/rds/aurora/)**. Navigate to ⌄

[2] [n.d.]. MariaDB. Retrieved on September 20, 2020 from **https://mariadb.com/ (https://mariadb.com/)**. Navigate to ⌄

[3] [n.d.]. Retrieved on September 20, 2020 from **http://www.computerworld.com/article/2834193/cloud-computing/5-tips-for-building-a-successful-hybrid-cloud.html (http://www.computerworld.com/article/2834193/cloud-computing/5-tips-for-building-a-successful-hybrid-cloud.html)**. Navigate to ⌄

[4] [n.d.]. Retrieved on September 20, 2020 from **https://www.getfilecloud.com/blog/2015/07/5-tips-on-optimizing-your-hybrid-cloud/ (https://www.getfilecloud.com/blog/2015/07/5-tips-on-optimizing-your-hybrid-cloud/)**. Navigate to ⌄

[5] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. 2004. Order-preserving encryption for numeric data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 563–574. DOI: **https://doi.org/10.1145/1007568.1007632 (https://doi.org/10.1145/1007568.1007632)** Navigate to ⌄

[6] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. 2013. Orthogonal security with cipherbase. In *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR'13)*. Retrieved from **http://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper33.pdf (http://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper33.pdf)**. Navigate to ⌄

[7] Arvind Arasu, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, and Ramarathnam Venkatesan. 2013. A secure coprocessor for database applications. In *Proceedings of the 23rd International Conference on Field programmable Logic and Applications (FPL'13)*. 1–8. DOI: **https://doi.org/10.1109/FPL.2013.6645524 (https://doi.org/10.1109/FPL.2013.6645524)** Navigate to ⌄

[8] Arvind Arasu and Raghav Kaushik. 2014. Oblivious query processing. In *Proceedings of the 17th International Conference on Database Theory (ICDT'14)*. 26–37. DOI: **https://doi.org/10.5441/002/icdt.2014.07 (https://doi.org/10.5441/002/icdt.2014.07)** Navigate to ⌄

[9] David W. Archer, Dan Bogdanov, Y. Lindell, Liina Kamm, Kurt Nielsen, Jakob Illeborg Pagter, Nigel P. Smart, and Rebecca N. Wright. 2018. From keys to databases - Real-world applications of secure multi-party computation. *IACR Cryptol. ePrint Arch.* 2018 (2018), 450. Retrieved from **https://eprint.iacr.org/2018/450 (https://eprint.iacr.org/2018/450)**. Navigate to ⌄

[10] Sumeet Bajaj and Radu Sion. 2013. CorrectDB: SQL engine with practical query authentication. *Proc. Very Large Data Base* 6, 7 (2013), 529–540. Retrieved from **http://www.vldb.org/pvldb/vol6/p529-bajaj.pdf (http://www.vldb.org/pvldb/vol6/p529-bajaj.pdf)**. Navigate to ⌄

[11] Sumeet Bajaj and Radu Sion. 2014. TrustedDB: A trusted hardware-based database with privacy and data confidentiality. *IEEE Trans. Knowl. Data Eng.* 26, 3 (2014), 752–765. DOI: **https://doi.org/10.1109/TKDE.2013.38 (https://doi.org/10.1109/TKDE.2013.38)** Navigate to ⌄

[12] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. 2007. Deterministic and efficiently searchable encryption. In *Proceedings of the 27th Annual International Cryptology Conference (CRYPTO'17)*. 535–552. DOI: **https://doi.org/10.1007/978-3-540-74143-5_30 (https://doi.org/10.1007/978-3-540-74143-5_30)** Navigate to ⌄

[13] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2015. Function secret sharing. In *Proceedings of the 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT'15)*. 337–367. DOI: **https://doi.org/10.1007/978-3-662-46803-6_12 (https://doi.org/10.1007/978-3-662-46803-6_12)** Navigate to ⌄

[14] Ran Canetti, Uriel Feige, Oded Goldreich, and Moni Naor. 1996. Adaptively secure multi-party computation. In *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing*. 639–648. DOI: **https://doi.org/10.1145/237814.238015 (https://doi.org/10.1145/237814.238015)** Navigate to ⌄

[15] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. 2015. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 668–679. DOI: **https://doi.org/10.1145/2810103.2813700 (https://doi.org/10.1145/2810103.2813700)** Navigate to ⌄

[16] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. 1998. Private information retrieval. *J. ACM* 45, 6 (1998), 965–981. DOI: **https://doi.org/10.1145/293347.293350 (https://doi.org/10.1145/293347.293350)** Navigate to ⌄

[17] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. *IACR Cryptol. ePrint Arch.* 2016 (2016), 86. Navigate to ⌄

[18] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. 2011. Searchable symmetric encryption: Improved definitions and efficient constructions. *J. Comput. Secur.* 19, 5 (2011), 895–934. DOI: **https://doi.org/10.3233/JCS-2011-0426 (https://doi.org/10.3233/JCS-2011-0426)** Navigate to ⌄

[19] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, Minos N. Garofalakis, and Charalampos Papamanthou. 2018. Practical private range search in depth. *ACM Trans. Database Syst.* 43, 1 (2018), 2:1–2:52. DOI: **https://doi.org/10.1145/3167971 (https://doi.org/10.1145/3167971)** Navigate to ⌄

[20] Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, and Chunwang Zhang. 2015. M2R: Enabling stronger privacy in MapReduce computation. In *Proceedings of the 24th USENIX Security Symposium (USENIXSecurity'15)*. 447–462. Retrieved from **https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/dinh (https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/dinh)**. Navigate to ⌄

[21] Shlomi Dolev, Niv Gilboa, and Ximing Li. 2015. Accumulating automata and cascaded equations automata for communicationless information theoretically secure multi-party computation: Extended abstract. In *Proceedings of the International Workshop on Security in Cloud Computing (SCC@ASIACCS'15)*. ACM, 21–29. Navigate to ⌄

[22] Shlomi Dolev, Peeyush Gupta, Yin Li, Sharad Mehrotra, and Shantanu Sharma. 2019. Privacy-preserving secret shared computations using mapreduce. *IEEE Trans. Depend. Secure Comput.* (2019). Navigate to ⌄

[23] Michael Egorov and MacLane Wilkinson. 2016. ZeroDB white paper. Retrieved from **http://arxiv.org/abs/1602.07168 (http://arxiv.org/abs/1602.07168)**. Navigate to ⌄

[24] Yuval Elovici, Ronen Waisenberg, Erez Shmueli, and Ehud Gudes. 2004. A structure preserving database encryption scheme. In *Proceedings of the Secure Data Management Workshop (VLDB'04)*. 28–40. DOI: **https://doi.org/10.1007/978-3-540-30073-1_3 (https://doi.org/10.1007/978-3-540-30073-1_3)** Navigate to ⌄

[25] Fatih Emekçi, Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2014. Dividing secrets to secure data outsourcing. *Info. Sci.* 263 (2014), 198–210. DOI: **https://doi.org/10.1016/j.ins.2013.10.006 (https://doi.org/10.1016/j.ins.2013.10.006)** Navigate to ⌄

[26] Csilla Farkas and Sushil Jajodia. 2002. The inference problem: A survey. *SIGKDD Explor.* 4, 2 (2002), 6–11. DOI: **https://doi.org/10.1145/772862.772864 (https://doi.org/10.1145/772862.772864)** Navigate to ⌄

[27] Benny Fuhry, H. A. Jayanth Jain, and Florian Kerschbaum. 2020. EncDBDB: Searchable encrypted, fast, compressed, in-memory database using enclaves. Retrieved from **https://arxiv.org/abs/2002.05097 (https://arxiv.org/abs/2002.05097)**. Navigate to ⌄

[28] Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. 2017. HardIDX: Practical and secure index with SGX. In *Proceedings of the 31st Annual Data and Applications Security and Privacy Conference (DBSec'17)*. 386–408. DOI: **https://doi.org/10.1007/978-3-319-61176-1_22 (https://doi.org/10.1007/978-3-319-61176-1_22)** Navigate to ⌄

[29] Craig Gentry. 2009. *A Fully Homomorphic Encryption Scheme*. Ph.D. Dissertation . Stanford University. Navigate to ⌄

[30] Niv Gilboa and Yuval Ishai. 2014. Distributed point functions and their applications. In *Proceedings of the 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT'14)*. 640–658. DOI: **https://doi.org/10.1007/978-3-642-55220-5_35 (https://doi.org/10.1007/978-3-642-55220-5_35)** Navigate to ⌄

[31] Oded Goldreich. 1987. Towards a theory of software protection and simulation by oblivious RAMs. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*. 182–194. DOI: **https://doi.org/10.1145/28395.28416 (https://doi.org/10.1145/28395.28416)** Navigate to ⌄

[32] Shafi Goldwasser and Silvio Micali. 1984. Probabilistic encryption. *J. Comput. Syst. Sci.* 28, 2 (1984), 270–299. DOI: **https://doi.org/10.1016/0022-0000(84)90070-9 (https://doi.org/10.1016/0022-0000(84)90070-9)** Navigate to ⌄

[33] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security (EUROSEC'17)*. 2:1–2:6. DOI: **https://doi.org/10.1145/3065913.3065915 (https://doi.org/10.1145/3065913.3065915)** Navigate to ⌄

[34] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. 2017. Leakage-abuse attacks against order-revealing encryption. In *Proceedings of the IEEE Symposium on Security and Privacy, (SP'17)*. 655–672. DOI: **https://doi.org/10.1109/SP.2017.44 (https://doi.org/10.1109/SP.2017.44)** Navigate to ⌄

[35] Peeyush Gupta, Yin Li, Sharad Mehrotra, Nisha Panwar, Shantanu Sharma, and Sumaya Almanee. 2019. Obscure: Information-theoretic oblivious and verifiable aggregation queries. *Proc. Very Large Data Base* 12, 9 (2019), 1030–1043. DOI: **https://doi.org/10.14778/3329772.3329779 (https://doi.org/10.14778/3329772.3329779)** Navigate to ⌄

[36] Hakan Hacigümüş, Balakrishna R. Iyer, Chen Li, and Sharad Mehrotra. 2002. Executing SQL over encrypted data in the database-service-provider model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 216–227. DOI: **https://doi.org/10.1145/564691.564717 (https://doi.org/10.1145/564691.564717)** Navigate to ⌄

[37] Hakan Hacigümüş, Sharad Mehrotra, and Balakrishna R. Iyer. 2002. Providing database as a service. In *Proceedings of the 18th International Conference on Data Engineering*. 29–38. DOI: **https://doi.org/10.1109/ICDE.2002.994695 (https://doi.org/10.1109/ICDE.2002.994695)** Navigate to ⌄

[38] Thomas H. Hinke. 1988. Inference aggregation detection in database management systems. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 96–106. DOI: **https://doi.org/10.1109/SECPRI.1988.8101 (https://doi.org/10.1109/SECPRI.1988.8101)** Navigate to ⌄

[39] Yuval Ishai and Eyal Kushilevitz. 1997. Private simultaneous messages protocols with applications. In *Proceedings of the 5th Israel Symposium on Theory of Computing and Systems (ISTCS'97)*. 174–184. DOI:

[40] Yuval Ishai, Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. 2016. Private large-scale databases with distributed searchable symmetric encryption. In *Proceedings of the Cryptographers' Track at the RSA Conference (CT-RSA'16)*. 90–107. DOI: https://doi.org/10.1007/978-3-319-29485-8_6 (https://doi.org/10.1007/978-3-319-29485-8_6)

[41] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS'12)*. Retrieved from http://www.internetsociety.org/access-pattern-disclosure-searchable-encryption-ramification-attack-and-mitigation (http://www.internetsociety.org/access-pattern-disclosure-searchable-encryption-ramification-attack-and-mitigation).

[42] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2016. Generic attacks on secure outsourced databases. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 1329–1340. DOI: https://doi.org/10.1145/2976749.2978386 (https://doi.org/10.1145/2976749.2978386)

[43] Ilan Komargodski and Mark Zhandry. 2016. Cutting-edge cryptography through the lens of secret sharing. In *Proceedings of the 13th International Conference on Theory of Cryptography (TCC'16)*. 449–479. DOI: https://doi.org/10.1007/978-3-662-49099-0_17 (https://doi.org/10.1007/978-3-662-49099-0_17)

[44] Guanling Lee, Chien-Yu Chang, and Arbee L. P. Chen. 2004. Hiding sensitive patterns in association rules mining. In *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)*. IEEE, 424–429.

[45] Chen Li, Houtan Shirani-Mehr, and Xiaochun Yang. 2007. Protecting individual information against inference attacks in data publishing. In *Proceedings of the 12th International Conference on Database Systems for Advanced Applications (DASFAA'07) (Lecture Notes in Computer Science)*, Kotagiri Ramamohanarao, P. Radha Krishna, Mukesh K. Mohania, and Ekawit Nantajeewarawat (Eds.), Vol. 4443. Springer, 422–433. DOI: https://doi.org/10.1007/978-3-540-71703-4_37 (https://doi.org/10.1007/978-3-540-71703-4_37)

[46] Jin Li, Zheli Liu, Xiaofeng Chen, Fatos Xhafa, Xiao Tan, and Duncan S. Wong. 2015. L-EncDB: A lightweight framework for privacy-preserving data queries in cloud computing. *Knowl.-Based Syst.* 79 (2015), 18–26. DOI: https://doi.org/10.1016/j.knosys.2014.04.010 (https://doi.org/10.1016/j.knosys.2014.04.010)

[47] Lichun Li, Michael Militzer, and Anwitaman Datta. 2014. rPIR: Ramp secret sharing based communication efficient private information retrieval. *IACR Cryptol. ePrint Arch.* 2014 (2014), 44. Retrieved from http://eprint.iacr.org/2014/044 (http://eprint.iacr.org/2014/044).

[48] Rui Li, Alex X. Liu, Ann L. Wang, and Bezawada Bruhadeshwar. 2014. Fast range query processing with strong privacy protection for cloud computing. *Proc. Very Large Data Base* 7, 14 (2014), 1953–1964. Retrieved from http://www.vldb.org/pvldb/vol7/p1953-li.pdf (http://www.vldb.org/pvldb/vol7/p1953-li.pdf).

[49] Rui Li, Alex X. Liu, Ann L. Wang, and Bezawada Bruhadeshwar. 2016. Fast and scalable range query processing with strong privacy protection for cloud computing. *IEEE/ACM Trans. Netw.* 24, 4 (2016), 2305–2318. DOI: https://doi.org/10.1109/TNET.2015.2457493 (https://doi.org/10.1109/TNET.2015.2457493)

[50] Chang Liu, Liehuang Zhu, Mingzhong Wang, and Yu-an Tan. 2014. Search pattern leakage in searchable encryption: Attacks and new construction. *Info. Sci.* 265 (2014), 176–188. DOI: https://doi.org/10.1016/j.ins.2013.11.021 (https://doi.org/10.1016/j.ins.2013.11.021)

[51] Wouter Lueks and Ian Goldberg. 2015. Sublinear scaling for multi-client private information retrieval. In *Proceedings of the 19th International Conference on Financial Cryptography and Data Security (FC'15)*. 168–186. DOI: https://doi.org/10.1007/978-3-662-47854-7_10 (https://doi.org/10.1007/978-3-662-47854-7_10)

[52] Paulo Martins, Leonel Sousa, and Artur Mariano. 2017. A survey on fully homomorphic encryption: An engineering perspective. *ACM Comput. Surv.* 50, 6 (2017), 83:1–83:33. DOI: https://doi.org/10.1145/3124441 (https://doi.org/10.1145/3124441)

[53] Sharad Mehrotra, Shantanu Sharma, Jeffrey D. Ullman, and Anurag Mishra. 2019. Partitioned data security on outsourced sensitive and non-sensitive data. In *Proceedings of the 35th IEEE International Conference on Data Engineering (ICDE'19)*. 650–661. DOI: https://doi.org/10.1109/ICDE.2019.00064 (https://doi.org/10.1109/ICDE.2019.00064)

[54] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. 2018. Oblix: An efficient oblivious search index. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'18)*. 279–296. DOI: https://doi.org/10.1109/SP.2018.00045 (https://doi.org/10.1109/SP.2018.00045)

[55] Muhammad Naveed, Seny Kamara, and Charles V. Wright. 2015. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 644–655. DOI: https://doi.org/10.1145/2810103.2813651 (https://doi.org/10.1145/2810103.2813651)

[56] Kerim Yasin Oktay, Murat Kantarcioglu, and Sharad Mehrotra. 2017. Secure and efficient query processing over hybrid clouds. In *Proceedings of the 33rd IEEE International Conference on Data Engineering (ICDE'17)*. 733–744. DOI: https://doi.org/10.1109/ICDE.2017.125 (https://doi.org/10.1109/ICDE.2017.125)

[57] Kerim Yasin Oktay, Sharad Mehrotra, Vaibhav Khadilkar, and Murat Kantarcioglu. 2015. SEMROD: Secure and efficient MapReduce over HybriD clouds. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 153–166. DOI: https://doi.org/10.1145/2723372.2723741 (https://doi.org/10.1145/2723372.2723741)

[58] Rafail Ostrovsky. 1990. Efficient computation on oblivious RAMs. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*. 514–523. DOI: https://doi.org/10.1145/100216.100289 (https://doi.org/10.1145/100216.100289)

[59] HweeHwa Pang and Xuhua Ding. 2014. Privacy-preserving ad-hoc equi-join on outsourced data. *ACM Trans. Database Syst.* 39, 3 (2014), 23:1–23:40. DOI: https://doi.org/10.1145/2629501 (https://doi.org/10.1145/2629501)

[60] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. 2016. Arx: A strongly encrypted database system. *IACR Cryptol. ePrint Arch.* 2016 (2016), 591. Retrieved from http://eprint.iacr.org/2016/591 (http://eprint.iacr.org/2016/591).

[61] Raluca A. Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2012. CryptDB: Processing queries on an encrypted database. *Commun. ACM* 55, 9 (2012), 103–111. DOI: https://doi.org/10.1145/2330667.2330691 (https://doi.org/10.1145/2330667.2330691)

[62] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A secure database using SGX. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'18)*. 264–278. DOI: https://doi.org/10.1109/SP.2018.00025 (https://doi.org/10.1109/SP.2018.00025)

[63] Michael O. Rabin. 2005. How to exchange secrets with oblivious transfer. *IACR Cryptol. ePrint Arch.* 2005 (2005), 187. http://eprint.iacr.org/2005/187 (http://eprint.iacr.org/2005/187).

[64] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy data analytics in the cloud using SGX. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'15)*. 38–54. DOI: https://doi.org/10.1109/SP.2015.10 (https://doi.org/10.1109/SP.2015.10)

[65] Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (1979), 612–613. DOI: https://doi.org/10.1145/359168.359176 (https://doi.org/10.1145/359168.359176)

[66] Erez Shmueli, Ronen Waisenberg, Yuval Elovici, and Ehud Gudes. 2005. Designing secure indexes for encrypted databases. In *Proceedings of the 19th Annual IFIP WG 11.3 Working Conference on Data and Applications Security*. 54–68. DOI: https://doi.org/10.1007/11535706_5 (https://doi.org/10.1007/11535706_5)

[67] Gary W. Smith. 1991. Modeling security-relevant data semantics. *IEEE Trans. Softw. Eng.* 17, 11 (1991), 1195–1203.

[68] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. 2000. Practical techniques for searches on encrypted data. In *Proceedings of the IEEE Symposium on Security and Privacy*. 44–55. DOI: https://doi.org/10.1109/SECPRI.2000.848445 (https://doi.org/10.1109/SECPRI.2000.848445)

[69] Julian James Stephen, Savvas Savvides, Russell Seidel, and Patrick Eugster. 2014. Practical confidentiality preserving big data analysis. In *Proceedings of the 6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'14)*. Retrieved from https://www.usenix.org/conference/hotcloud14/workshop-program/presentation/stephen (https://www.usenix.org/conference/hotcloud14/workshop-program/presentation/stephen).

[70] Sai Deep Tetali, Mohsen Lesani, Rupak Majumdar, and Todd D. Millstein. 2013. MrCrypt: Static analysis for secure cloud computation. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'13; part of SPLASH'13)*. 271–286. DOI: https://doi.org/10.1145/2509136.2509554 (https://doi.org/10.1145/2509136.2509554)

[71] Quoc-Cuong To, Benjamin Nguyen, and Philippe Pucheral. 2016. Private and scalable execution of SQL aggregates on a secure decentralized architecture. *ACM Trans. Database Syst.* 41, 3, Article 16 (Aug. 2016), 43 pages. DOI: https://doi.org/10.1145/2894750 (https://doi.org/10.1145/2894750)

[72] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. 2013. Processing analytical queries over encrypted data. *Proc. VLDB Endow.* 6, 5 (Mar. 2013), 289–300. DOI: https://doi.org/10.14778/2535573.2488336 (https://doi.org/10.14778/2535573.2488336)

[73] Cong Wang, Ning Cao, Jin Li, Kui Ren, and Wenjing Lou. 2010. Secure ranked keyword search over encrypted cloud data. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS'10)*. 253–262. DOI: https://doi.org/10.1109/ICDCS.2010.34 (https://doi.org/10.1109/ICDCS.2010.34)

[74] Cong Wang, Sherman S. M. Chow, Qian Wang, Kui Ren, and Wenjing Lou. 2013. Privacy-preserving public auditing for secure cloud storage. *IEEE Trans. Comput.* 62, 2 (2013), 362–375. DOI: https://doi.org/10.1109/TC.2011.245 (https://doi.org/10.1109/TC.2011.245)

[75] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. 2017. Splinter: Practical private queries on public data. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*. 299–313. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/wang-frank (https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/wang-frank).

[76] Shuhong Wang, Xuhua Ding, Robert H. Deng, and Feng Bao. 2006. Private information retrieval using trusted hardware. *IACR Cryptol. ePrint Arch.* 2006 (2006), 208. Retrieved from http://eprint.iacr.org/2006/208 (http://eprint.iacr.org/2006/208).

[77]   Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. 2017. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*. 2421–2434. DOI: **https://doi.org/10.1145/3133956.3134038 (https://doi.org/10.1145/3133956.3134038)**   Navigate to ▾

[78]   Wai Kit Wong, Ben Kao, David Wai-Lok Cheung, Rongbin Li, and Siu-Ming Yiu. 2014. Secure query processing with data interoperability in a cloud database environment. In *Proceeding sof the International Conference on Management of Data (SIGMOD'14)*. 1395–1406. DOI: **https://doi.org/10.1145/2588555.2588572 (https://doi.org/10.1145/2588555.2588572)**   Navigate to ▾

[79]   Min Xu, Antonis Papadimitriou, Andreas Haeberlen, and Ariel Feldman. [n.d.]. Hermetic: Privacy-preserving distributed analytics without (most) side channels. Retrieved on September 20, 2020 from **https://www.cis.upenn.edu/~ahae/papers/hermetic-tr.pdf (https://www.cis.upenn.edu/~ahae/papers/hermetic-tr.pdf)**.   Navigate to ▾

[80]   Shucheng Yu, Cong Wang, Kui Ren, and Wenjing Lou. 2010. Achieving secure, scalable, and fine-grained data access control in cloud computing. In *Proceedings of the INFOCOM Conference*. 534–542.   Navigate to ▾

[81]   Shucheng Yu, Cong Wang, Kui Ren, and Wenjing Lou. 2010. Attribute based data sharing with attribute revocation. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS'10)*. 261–270. DOI: **https://doi.org/10.1145/1755688.1755720 (https://doi.org/10.1145/1755688.1755720)**   Navigate to ▾

[82]   Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2017. Opaque: An oblivious and encrypted distributed analytics platform. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*. 283–298. **https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zheng (https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zheng)**.   Navigate to ▾

Footnotes

[1] **https://www.stealthsoftwareinc.com/ (https://www.stealthsoftwareinc.com/)**.

[2] The sensitive and non-sensitive data classification, which is common in industries for secure computing [3, 4] and done via appropriately using existing techniques surveyed in Reference [26]; for example, (i) inference detection using graph-based semantic data modeling [38], (ii) user-defined relationships between sensitive and non-sensitive data [67], (iii) constraints-based mechanisms, (iv) sensitive patterns hiding using sanitization matrix [44], and (v) common knowledge-based association rules [45]. However, it is important to mention here that non-sensitive data can, over time, become sensitive and/or lead to inferences about sensitive data. This is an inevitable risk of the approaches that exploit sensitive data classification. Note that all the above-mentioned work based on sensitive/non-sensitive classification make a similar assumption.

[3] QB trades off increased communication costs for executing queries, while reducing very significantly cryptographic operations. This tradeoff significantly improves performance, especially, when using cryptographic mechanisms, e.g., fully homomorphic encryption that takes several seconds to compute a single operation [52], secret-sharing-based techniques that take a few seconds [25], or techniques such as bilinear maps that take over 1.5 h to perform joins on a dataset of size less than 10 MB [59]. 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 Reference [57] in the context of MapReduce, where overshuffling to prevent the adversary to infer sensitive keys in the context of hybrid cloud was shown to be significantly better compared to private side operations.

[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 article.

[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 Reference [50], 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 (Section 4), 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.

[8] The HBC adversary cannot launch the chosen-plaintext attack (CPA) and the chosen-ciphertext attack (CCA). Since the sensitive data is non-deterministically encrypted (by our assumption), it is not prone to the ciphertext only attack (COA).

[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.

[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.

[12] Of course, if the encrypted relation does not have any tuple having s_1, then the adversary can learn that ns_1 is not associated with any tuple. However, this can be prevented trivially by outsourcing fake tuples having s_1.

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

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

[15] We assume that there are nine non-sensitive values, and computed that we need three sensitive and three non-sensitive bins.

[16] The DB owner may also build a k-ary tree, where each node (except leaf nodes) contains k child nodes.

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

[18] In case, if the DB owner cannot collect new values equal to the number of bins, then the DB owner may outsource fake values.

[19] 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.

[20] If there are less than x—which is the size of a non-sensitive bin—frequent predicates in a non-sensitive bin, then we need to send fake queries for infrequent keyword of the bin, as frequent as frequent predicates, leading to retrieval of each sensitive bin. It will hide that the how many keywords are frequent and infrequent in the bin.

[21] Please note that we selected two different machines, because our intention is not to compare Jana and Opaque.

[22] Data insertion time was significant in Jana, thus, we used only 1M rows. Further, the current Jana version does not support joins in MPC.

[23] Note that in the conference version of this article, the execution time of Jana was higher than the time taken in Table 10. The reason is that in this article, we used a newer version of Jana.

Authors' addresses: S. Mehrotra, S. Sharma, D. Ghosh, P. Gupta, and A. Mishra, Department of Computer Science, University of California, Irvine, CA, USA; email: **shantanu.sharma@uci.edu (mailto:shantanu.sharma@uci.edu)**; J. D. Ullman, Department of Computer Science, Stanford University, CA, USA.