Secure Joins with MapReduce

Xavier Bultel[1], Radu Ciucanu[2], Matthieu Giraud[3], Pascal Lafourcade[3], and Lihua Ye[4]

[1] IRISA, Université de Rennes 1, France
`xavier.bultel@irisa.fr`
[2] INSA Centre Val de Loire, Univ. Orléans, LIFO EA 4022, Bourges, France
`radu.ciucanu@insa-cvl.fr`
[3] LIMOS, Université Clermont Auvergne, France
{`matthieu.giraud,pascal.lafourcade`}`@uca.fr`
[4] Harbin Institute of Technology, China
`16s003041@stu.hit.edu.cn`

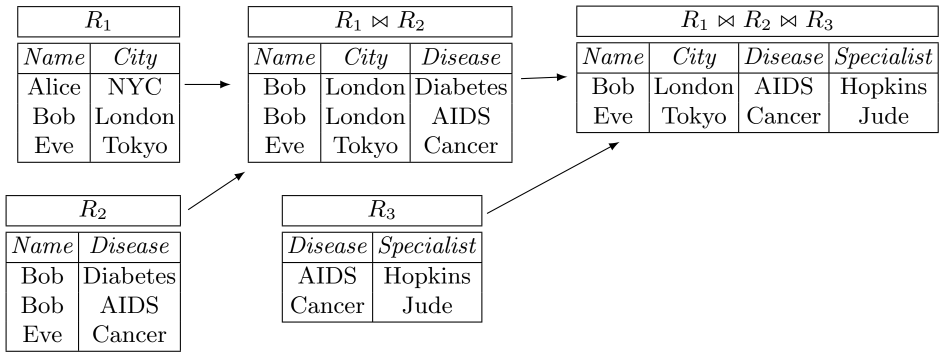
Abstract. MapReduce is one of the most popular programming paradigms that allows a user to process Big data sets. Our goal is to add privacy guarantees to the two standard algorithms of join computation for MapReduce: the *cascade* algorithm and the *hypercube* algorithm. We assume that the data is externalized in an *honest-but-curious* server and a user is allowed to query the join result. We design, implement, and prove the security of two approaches: (i) *Secure-Private*, assuming that the public cloud and the user do not collude, (ii) *Collision-Resistant-Secure-Private*, which resists to collusions between the public cloud and the user i.e., when the public cloud knows the secret key of the user.

Keywords: Database Query, MapReduce, Security, Natural Joins

1 Introduction

With the advent of Big data, new techniques have been developed to process parallel computation on a large cluster. One of them is the MapReduce programming paradigm [11], which allows a user to keep data in public clouds and to perform computations on it. A MapReduce program uses two functions (*map* and *reduce*) that are executed on a large cluster of machines in parallel. The popularity of the MapReduce paradigm comes from the fact that the programmer does not need to handle aspects such as the partitioning of the data, scheduling the program's execution across the machines, handling machine failures, and managing the communication between different machines.

MapReduce users often rent storage and computing resources from a public cloud provider (e.g., Google Cloud Platform, Amazon Web Services, Microsoft Azure). External storage and computations with a public cloud make the Big data processing accessible to users that can not afford building their own clusters. Yet, outsourcing data and computations to a public cloud involves inherent security and privacy concerns. Since the data is externalized, it can be communicated over an untrustworthy network and processed on some untrustworthy machines, where malicious public cloud users may learn private data.



R_1	
Name	*City*
Alice	NYC
Bob	London
Eve	Tokyo

R_2	
Name	*Disease*
Bob	Diabetes
Bob	AIDS
Eve	Cancer

$R_1 \bowtie R_2$		
Name	*City*	*Disease*
Bob	London	Diabetes
Bob	London	AIDS
Eve	Tokyo	Cancer

R_3	
Disease	*Specialist*
AIDS	Hopkins
Cancer	Jude

$R_1 \bowtie R_2 \bowtie R_3$			
Name	*City*	*Disease*	*Specialist*
Bob	London	AIDS	Hopkins
Eve	Tokyo	Cancer	Jude

Fig. 2: Joins between relations R_1, R_2 and R_3.

We address the fundamental problem of computing relational joins between an arbitrary number of relations in a privacy-preserving manner using MapReduce. We assume that the data is externalized in the cloud by the data owner and there is a user that is allowed to query it as shown in Fig. 1. This standard model has been used recently by Dolev et al. [14].

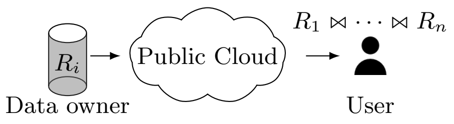


Fig. 1: The system architecture.

We next present via a running example the concept of relational joins. Then, we present MapReduce computations, our problem statement, and illustrate the privacy issues related to joins computation with MapReduce.

Example 1. The data owner is a hospital storing relations R_1, R_2, R_3 cf. Fig. 2. The (natural) join of these relations, denoted $R_1 \bowtie R_2 \bowtie R_3$, is the relation whose tuples are composed of tuples of R_1, R_2 and R_3 that agree on shared attributes. In our case, the attribute *Name* is shared between R_1 and R_2. Moreover, the attribute *Disease* is shared between intermediate join result ($R_1 \bowtie R_2$) and relation R_3. In Fig. 2, we give both the intermediate result ($R_1 \bowtie R_2$) and the final result ($R_1 \bowtie R_2$) $\bowtie R_3$. We observe that tuple (Alice, NYC) from relation R_1, tuple (Bob, Diabetes) from relation R_2, and tuple (Bob, London, Diabetes) from relation $R_1 \bowtie R_2$ do not participate to the final result.

1.1 Joins with MapReduce

Two algorithms for computing relational joins with MapReduce are presented in the literature: the *Cascade* algorithm (i.e., a generalization of the binary join from Chapter 2 of [18]) and the *Hypercube* algorithm [4,9]. In the following, a *reducer* refers to the application of the reduce function to a single key.

Cascade Algorithm. To compute an n-ary join ($n \geqslant 2$), the cascade algorithm uses $n - 1$ MapReduce rounds i.e., a sequence of $n - 1$ binary joins. A binary

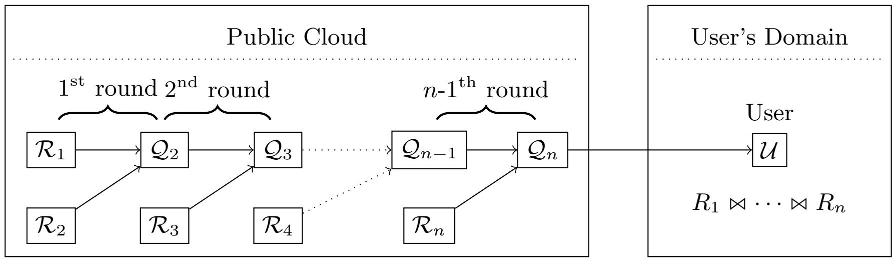
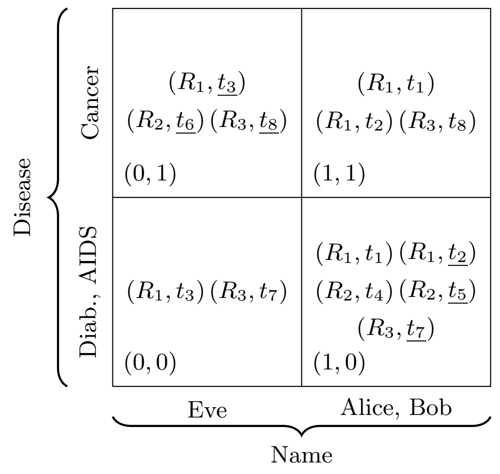


Fig. 3: Cascade of joins with MapReduce between n relations.

join works as follows: first, it applies the *map* function on the first two relations R_1 and R_2 that are spread over sets of nodes $\mathcal{R}_1$ and $\mathcal{R}_2$, respectively. The map function creates for each tuple of each relation a *key-value* pair where key is equal to values of shared attributes between the two relations, and value is equal to non-shared values of the tuple as well as the name of the relation. Then, the key-value pairs are grouped by key i.e., all key-value pairs output by the map phase which have the same key are sent to the same reducer. For each key and from the associated values coming from these two relations, the *reduce* function creates all possible tuples corresponding to the joins of these two relations. We obtain as intermediate result a new relation denoted Q_2 that is spread over a set of nodes $\mathcal{Q}_2$. This first step defines the first round of the cascade algorithm. We illustrate this process in Fig. 3.

Example 1 continued. To compute $(R_1 \bowtie R_2) \bowtie R_3$ with MapReduce following the cascade algorithm, we start by joining R_1 and R_2. Relations R_1 and R_2 share attribute *Name*. Hence from R_1, the map produces the following key-value pairs: (Alice, (R_1, NYC)), (Bob, (R_1, London)), and (Eve, (R_1, Tokyo)). These key-value pairs are sent to three different reducers depending on the key value. From relation R_2, the map produces key-value pairs (Bob, (R_2, Diabetes)), (Bob, (R_2, AIDS)), and (Eve, (R_2, Cancer)). We stress that values of pairs (Bob, (R_2, Diabetes)) and (Bob, (R_2, AIDS)) are sent to the same reducer as the pair (Bob, (R_1, London)) since all these pairs have the same key. Similarly, (Eve, (R_2, Cancer)) and (Eve, (R_1, Tokyo)) are sent to the same reducer. The pair (Alice, (R_1, NYC)) does not participate in the join result since no other pair shares the same key. Then, from values (R_1, London), (R_2, Diabetes), and (R_2, AIDS) present on the reducer associated to the key Bob, the reduce creates all possible tuples with values coming from different relations i.e., (Bob, London, Diabetes) and (Bob, London, AIDS). Similarly, the reducer associated to the key Eve produces (Eve, Tokyo, Cancer). These tuples correspond to the relation $(R_1 \bowtie R_2)$ cf. Fig. 2. We apply the map and the reduce functions on relations $(R_1 \bowtie R_2)$ and R_3 sharing the attribute *Disease*. From $(R_1 \bowtie R_2)$, the map function produces key-value pairs: (Diabetes, ($R_1 \bowtie R_2$, Bob, London)), (AIDS, ($R_1 \bowtie R_2$, Bob, London)), and (Cancer, ($R_1 \bowtie R_2$, Eve, Tokyo)). From R_3, the map produces: (AIDS, (R_3, Hopkins)), and (Cancer, (R_3, Jude)). Finally, the



From relation R_1:

- $t_1 = (\text{Alice}, \text{NYC})$
- $t_2 = (\text{Bob}, \text{London})$
- $t_3 = (\text{Eve}, \text{Tokyo})$

From relation R_2:

- $t_4 = (\text{Bob}, \text{Diabetes})$
- $t_5 = (\text{Bob}, \text{AIDS})$
- $t_6 = (\text{Eve}, \text{Cancer})$

From relation R_3:

- $t_7 = (\text{AIDS}, \text{Hopkins})$
- $t_8 = (\text{Cancer}, \text{Jude})$

Fig. 4: Running example with hypercube algorithm. Underlined tuples correspond to tuples that participate to the final join result.

reduce step produces tuples (Bob, London, AIDS, Hopkins) and (Eve, Tokyo, Cancer, Jude) corresponding to relation $(R_1 \bowtie R_2) \bowtie R_3$ cf. Fig. 2.

Hypercube Algorithm. Contrarily to cascade, the hypercube computes the join of all n relations in only one MapReduce round. The hypercube has dimension d (where d is the number of join attributes). There are $p = \prod_{1 \leqslant j \leqslant d} \alpha_j$ reducers denoted $\mathcal{H}_i$ (for $1 \leqslant i \leqslant p$), where α_j is the number of buckets associated with the j^{th} attribute. Hence, each reducer $\mathcal{H}_i$ can be uniquely identified by a point in the hypercube. For each relation R_i spread over a set of nodes $\mathcal{R}_i$, the *map* function computes the image of all tuples on the d dimensions of the hypercube to decide to which reducers $\mathcal{H}_i$ the tuple should be sent. Then, each reducer computes all possible combinations of input tuples that agree on shared attributes, only if all n relations are represented on the same reducer. All these combinations correspond to the final result of the n-ary join.

Example 1 continued. We have two join attributes (*Name* and *Disease*), hence two hash functions h_N and h_D for attributes *Name* and *Disease*, respectively. For instance, assume 4 reducers establishing a 2×2 square cf. Fig. 4, where h_N(Eve)=0, h_N(Alice)=h_N(Bob)=1, h_D(Diabetes)=h_D(AIDS)=0, and h_D(Cancer)=1. For each tuple of each relation, we compute the value of the *Name* component (if there exists) with the hash function h_N and the value of the *Disease* component (if there exists) with the hash function h_D. For instance, the tuple $t_6 = (\text{Eve}, \text{Cancer})$ of the relation R_2 is sent to the reducer of coordinates $(0, 1)$ since $h_N(\text{Eve}) = 0$ and $h_D(\text{Cancer}) = 1$ (cf. Fig. 4). If one of these two attributes is missing in a tuple, then the tuple is replicated over all reducers associated to the different values of the missing attributes of the tuple. For example, tuple $t_1 = (\text{Alice}, \text{NYC})$ of relation R_1 has no attribute *Disease*, and consequently, is sent to reducers $(1, 0)$ and $(1, 1)$. In such a situation we may write $(1, \star)$ to simplify presentation. Finally, each reducer performs all possible combinations over tuples that agree on join attributes of the three relations R_1, R_2, and R_3. We obviously obtain the same final result as for cascade algorithm.

1.2 Problem statement

We assume three participants: the data owner, the public cloud, and the user (cf. Fig. 1). The data owner externalizes n relations $R_{1 \leqslant i \leqslant n}$ to the public cloud. We assume that the public cloud is *honest-but-curious* i.e., it executes dutifully the computation tasks but tries to learn the maximum of information on tuples of each relation. In order to preserve privacy of data owner and to allow the join computation between relations, we want that the cloud learns nothing about input data or join result. Moreover, we want that the user who queries the join result learns nothing else than the final join result i.e., she does not learn information on tuples of relations that do not participate to the final result.

Sets of nodes of type $\mathcal{R}$, $\mathcal{Q}$, and $\mathcal{H}$ are *honest-but-curious.* We denote by $\mathbb{R}_i$ the set of attributes of a relation R_i, for $1 \leqslant i \leqslant n$. In the case of the cascade algorithm, we denote by $\mathbb{Q}_i$ the set of attributes of relation Q_i for $1 \leqslant i \leqslant n$, where $R_1 = Q_1$. Finally we denote by $\mathbb{X}$ the set of shared attributes between the n relations i.e., $\mathbb{X} = | \cup_{1 \leqslant i \neq j \leqslant n} \mathbb{R}_i \cap \mathbb{R}_j |$.

We expect the following security properties:

1. Neither a set of nodes $\mathcal{R}_i$ nor data owner learn final result data.
2. A set of nodes $\mathcal{Q}_i$ (resp. $\mathcal{H}_i$) cannot learn owner's data and final result.
3. The user learns nothing else than result $R_1 \bowtie \cdots \bowtie R_n$ i.e., he does not learn tuples from the input relation that do not participate in the result.

Example 1 continued. Looking at the three security properties of the problem statement, we see that the cascade and the hypercube algorithms do not respect properties (1), (2), and (3). In fact, both algorithms reveal to the public cloud all tuples of relations R_1, R_2 and R_3 since they are not encrypted. Moreover, if the user colludes with the intermediate set of nodes $\mathcal{R}_1 \bowtie \mathcal{R}_2$, then he learns tuples that he should not, in this case the tuple $(\text{Bob}, \text{London}, \text{Diabetes})$ (Fig. 2).

Contributions. We propose two approaches that extend the two aforementioned join algorithms while ensuring the desired security properties, and remaining efficient from both computational and communication points of view.

- The *Secure-Private* (SP) approach assumes that the public cloud and the user do not collude. We encrypt all values of each tuple using a public key encryption scheme with the user public key pk_u. To be able to perform the equality joins between relations we rely on pseudo-random functions.
- The *Collision-Resistant-Secure-Private* (CRSP) approach assumes that the public cloud and the user collude, that means the public cloud knows the private key sk_u of the user. In this case, we cannot encrypt all tuples using

simply a public encryption scheme since the public cloud can decrypt all these encrypted tuples using the secret key of the user. To avoid this problem, we introduce a proxy such that the data owner also uses the public key of the proxy pk_t to encrypt ciphers of tuple values. Thus, we avoid that the public cloud decrypts tuples values received from the data owner even if the public cloud has the secret key sk_u of the user.

- We give experimental results of our SP and CRSP approaches for the cascade and the hypercube algorithms using Apache Hadoop [1] open-source MapReduce implementation and a real-world Twitter dataset [2].
- We prove that our SP and CRSP approaches satisfy the security properties using the random oracle model. We also notice a limitation regarding learning repetitions between pseudo-random values which seems to us inherent because we need to perform equi-joins. Details of proofs are given in the technical report available online [3].
- We quantify the computational and communication overhead of our two secure approaches. We observe that the overhead is linear. Due to lack of space, we include this discussion only in our technical report [3].

Related work. Since the seminal MapReduce paper [11], different protocols have been proposed to perform operations in a privacy-preserving manner [12] such as search [6] [20], count [24], matrix multiplication [7] or joins [14].

Chapter 2 of [18] presents an introduction to the MapReduce paradigm. In particular, it includes the MapReduce algorithm for cascade joins that we enhance with privacy guarantees. Very few approaches address the privacy preserving execution of relational joins in MapReduce and have different assumptions than we do. For instance, Emekçi et al. [16] proposed protocols to perform joins in a privacy-preserving manner using the Shamir's secret sharing [23]. Contrary to us, they do not consider the MapReduce paradigm and their approach cannot be trivially adopted in MapReduce because values of shared attributes are encrypted in a non-deterministic way. Laur et al. [17] also proposed a protocol to compute joins using secret sharing but do not consider the MapReduce paradigm and their approach is limited to two relations. Chow et al. [8] introduced a generic model that uses two non-colluding servers to perform join computation between n relations in a privacy-preserving manner but do not consider the MapReduce paradigm. On the other hand, we assume a more general setting where the public cloud servers collude. Dolev et al. [14] proposed a technique for executing MapReduce computations in the public cloud while preserving privacy using the Shamir's secret sharing [23] and accumulating-automata [13]. Join computation is executed on secret-shares in the public cloud and at the end, the user performs the interpolation on the outputs. Contrary to us, authors assume that the different cloud nodes do not collude, otherwise they can construct the secret from shares. Moreover in our setting, we externalized entirely the computation in the cloud and the user has only to decrypt the join result, contrary to the need of doing interpolations in [14]. Finally, none of the aforementioned approaches propose a secure approach for the hypercube algorithm with MapReduce.

Finally, the system that is most closely related to our work is Popa et al.'s CryptDB [21,22]. CryptDB provides practical and provable confidentiality in the face of curious server for applications backed by SQL databases. It works by executing SQL queries over encrypted data using a collection of efficient SQL-aware encryption schemes. However, they do not consider the MapReduce paradigm.

To the best of our knowledge, we are the first to propose two secure approaches of join computation for the cascade and the hypercube MapReduce algorithms, where the user has only to decrypt the result received from the cloud.

Outline. We present the cascade and hypercube algorithms for the n-ary join computation with MapReduce in Section 2. We present our SP and CRSP approaches for both algorithms in Section 3. In Section 4, we compare experimentally the performance of our approaches vs the insecure algorithms. Then, we prove the security of the SP and CRSP approaches in Section 5. Finally, we outline conclusion and future work in Section 6.

2 n-ary Joins with MapReduce

We formally present the standard algorithms for computing n-ary joins $Q = R_1 \bowtie \cdots \bowtie R_n$ with MapReduce: *cascade* i.e., a sequence of $n-1$ rounds of binary joins [18] and *hypercube* [4] i.e., a single round doing all the $n-1$ joins. We have already presented examples for both algorithms in Section 1.1.

```
Algorithm: BinaryJoin($Q, R$)
Map function
Input: ($key, value$)
// $key$: id of chunk of $Q$ or $R$
// $value$: coll. of $t_q \in Q$ or $t_r \in R$
foreach $t_q \in Q$ do
    emit ($\pi_{\mathbb{Q}\cap\mathbb{R}}(t_q), (\mathtt{Q}, t_q)$);
foreach $t_r \in R$ do
    emit ($\pi_{\mathbb{Q}\cap\mathbb{R}}(t_r), (\mathtt{R}, t_r)$);
```

```
Algorithm: BinaryJoin($Q, R$)
Reduce function
Input: ($key, value$)
// $key$: $\pi_{\mathbb{Q}\cap\mathbb{R}}(t)$ with $t \in Q$ or $t \in R$
// $values$: coll. of $(\mathtt{Q}, t_q)$ or $(\mathtt{R}, t_r)$
foreach $(\mathtt{Q}, t_q) \in values$ do
    foreach $(\mathtt{R}, t_r) \in values$ do
        emit ($t_q \bowtie t_r, t_q \bowtie t_r$);
```

Fig. 5: BinaryJoin algorithm for natural join with MapReduce between Q and R.

2.1 Cascade Algorithm

We recall that the i^{th} round of the cascade algorithm takes action between sets of nodes $\mathcal{Q}_i$ and $\mathcal{R}_{i+1}$, with $1 \leqslant i \leqslant n-1$ and that relation R_1 is denoted Q_1. The term chunk refers to a fragment of information. Moreover, $\mathbb{R}$ denotes the schema of the relation R i.e. the set of attributes of the relation R.

We present in Fig. 5 the binary join with MapReduce between two relations. To compute join between n relations $R_1, \ldots, R_n$, we apply $n-1$ times the binary join (Fig. 5) as presented in the cascade algorithm in Fig. 6. The final relation Q_n corresponds to $R_1 \bowtie \cdots \bowtie R_n$.

```
Algorithm: CascadeJoin($R_1, \ldots, R_n$)
for $1 \leqslant i \leqslant n-1$ do
    $Q_{i+1} \leftarrow$ BinaryJoin($Q_i, R_{i+1}$);
```

Fig. 6: Cascade Algorithm.

2.2 Hypercube Algorithm

We assume that we have an hypercube of dimension d, where $d = |\mathbb{X}| = |\{X_1, \ldots, X_d\}| = |\cup_{1 \leqslant i \neq j \leqslant n} \mathbb{R}_i \cap \mathbb{R}_j|$ i.e., d is the number of join attributes.

Moreover, we assume that we have d (non-cryptographic) hash functions h_ℓ (where $1 \leqslant \ell \leqslant d$) such that $h_\ell \colon X_\ell \to [\![0, \alpha_\ell]\!]$ where α_ℓ is the number of buckets for the attribute X_ℓ. Hence, the hypercube is composed of $\alpha_1 \cdots \alpha_\ell$ reducers where each reducer is uniquely identified by a d-tuple $(x_1, \ldots, x_d)$ with $x_\ell \in [\![0, \alpha_\ell]\!]$ for $1 \leqslant \ell \leqslant d$. In the following, we denote by A^i_j the j-th attribute of the relation R_i where $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant |\mathbb{R}_i|$.

We present in Fig. 7 the hypercube algorithm for the join computation with MapReduce between n relations $R_1, \ldots, R_n$. The map function sends the pair to the corresponding reducer of the hypercube associated to the coordinates of the key-value pair's key where the star $\star$ in the ℓ-th coordinate means that we duplicate the tuple t on all the α_ℓ buckets of the ℓ-th dimension of the hypercube. Then, if the same reducer of the hypercube has at least one tuple coming from all the n relations and that these tuples agree on their shared attributes then the reduce function produces all possible key-values pairs of the form $(t_1 \bowtie \cdots \bowtie t_n, t_1 \bowtie \cdots \bowtie t_n)$ where $t_i \in R_i$ (with $1 \leqslant i \leqslant n$).

```
Algorithm:
 HypercubeJoin($R_1, \ldots, R_n$)
Map function
Input: ($key, value$)
// $key$: id of chunk of $R_{1 \leqslant i \leqslant n}$
// $value$: collection of $t \in R_i$
foreach $t \in R_i$ do
    for $1 \leqslant \ell \leqslant d$ do
        if $X_\ell \in \mathbb{R}_i$ then
            $x_\ell \leftarrow h_\ell(\pi_{X_\ell}(t))$;
        else $x_\ell \leftarrow \star$;
    emit $((x_1, \ldots, x_d), (\mathtt{R}_i, t))$.
Reduce function
Input: ($key, values$)
// $key$: $(x_1, \ldots, x_d)$, $x_\ell \in [\![0, \alpha_\ell]\!]$
// $values$: collection of $(\mathtt{R}_i, t)$
for $1 \leqslant i \leqslant n$ do
    $R'_i \leftarrow \bigcup_{(\mathtt{R}_i, t) \in values} \{t\}$;
for $t \in R'_1 \bowtie \cdots \bowtie R'_n$ do
    emit $(t, t)$.
```

Fig. 7: Hypercube algorithm.

3 Secure n-ary Joins with MapReduce

Before formally presenting our secure algorithms, we present the needed cryptographic tools. We illustrate the intuition of each of our algorithms while relying on our running example from the Introduction.

3.1 Cryptographic Tools

We define negligible function, pseudo-random function, and public key encryption cryptosystem.

Definition 1 (Negligible function). *A function $\epsilon : \mathbb{N} \rightarrow \mathbb{R}$ is negligible in η if for every positive polynomial $p(\cdot)$ and sufficiently large η, $\epsilon(\eta) < 1/p(\eta)$.*

Definition 2 (Pseudo-random function). *Let η be a security parameter. A function $f\colon \{0,1\}^{\ell(\eta)} \times \{0,1\}^{l_0} \rightarrow \{0,1\}^{l_1}$ is a pseudo-random function if it is computable in polynomial time in η and if for all polynomial-size $\mathcal{B}$,*

$$\left| \Pr\left[\mathcal{B}^{f(k,\cdot)} = 1\colon k \xleftarrow{\$} \{0,1\}^{\ell(\eta)}\right] - \Pr\left[\mathcal{B}^{g(\cdot)} = 1\colon g \xleftarrow{\$} \mathsf{Func}[l_0,l_1]\right] \right| \leqslant \epsilon(\eta)$$

where, $\ell(\cdot)$ is a polynomial function, $\mathsf{Func}[l_0,l_1]$ is the space of functions defined over domain $\{0,1\}^{l_0}$ and codomain $\{0,1\}^{l_1}$, $\epsilon(\cdot)$ is a negligible function in η and the probabilities are taken over the choice of k and g.

In the rest of the paper, the pseudo-random function $f(k,\cdot)$ is denoted $f_k(\cdot)$.

Definition 3 (Public Key Encryption). *Let η be a security parameter. A* Public Key Encryption *(PKE) scheme Π is defined by three algorithms $(\mathcal{G},\mathcal{E},\mathcal{D})$:*

$\mathcal{G}(\eta)$**:** *it takes the security parameter η and returns a key pair $(\mathsf{pk},\mathsf{sk})$.*
$\mathcal{E}_{\mathsf{pk}}(m)$**:** *it takes a public key pk and a plaintext m and returns the ciphertext c.*
$\mathcal{D}_{\mathsf{sk}}(c)$**:** *it takes a private key sk and a ciphertext c and returns the plaintext m.*

3.2 Preprocessing and Outsourcing

To prevent the cloud from learning the content of relations, the data owner protects each relation $R_{1\leqslant i\leqslant n}$ before outsourcing. The protected relation obtained from R_i is denoted $\hat{R}_i$ and is sent to the public cloud by the data owner.

The data owner protects relations in two ways. First, it uses a pseudo-random function $f_k(\cdot)$ where k is the data owner secret key. The data owner applies $f_k(\cdot)$ on values of shared attributes of each tuples of relations $R_{1\leqslant i\leqslant n}$. Since a pseudo-random function is determinist, it allows the cloud to perform equality tests between values of join attributes. On other hand, the data owner encrypts for each user each component of tuples with an *indistinguishable under chosen plaintext attack* (IND-CPA) public key encryption scheme (e.g., ElGamal [15], RSA-OAEP [5]) using the public key pk_u of the user. Hence the encrypted values of non-shared attributes do not give any information to an adversary. Values of shared attributes are also encrypted using the public scheme encryption since we want the user can decrypt them.

We present the preprocessing algorithm in Fig. 8. The set $\mathsf{visited}$ prevents the data owner from (IND-CPA) encrypting several times the same values. We stress that A^f and $A^{\mathcal{E}}$ are just notations making explicit the correspondences between initial and outsourced data. For instance, if a relation R has one attribute "Name" that is shared with an other relation, then this attribute in the protected relation will be denoted "Name^f"; we apply the same way the notation $A^{\mathcal{E}}$. Moreover $\hat{\mathbb{R}}_i$ is the schema of the protected relation $\hat{R}_i$. We give an example for the cascade algorithm in Fig. 9 using the running example.

$\hat{R}_1$

$Name^f$	$Name^{\mathcal{E}}$	$City^{\mathcal{E}}$
41	{Alice}	{NYC}
23	{Bob}	London}
36	{Eve}	{Tokyo}

$\hat{R}_1 \bowtie \hat{R}_2$

$Name^f$	$Disease^f$	$Name^{\mathcal{E}}$	$City^{\mathcal{E}}$	$Disease^{\mathcal{E}}$
23	87	{Bob}	{London}	{Diabetes}
23	18	{Bob}	{London	{AIDS}
36	99	{Eve}	{Tokyo}	{Cancer}

$\hat{R}_2$

$Name^f$	$Disease^f$	$Disease^{\mathcal{E}}$
23	87	{Diabetes}
23	18	{AIDS}
36	99	{Cancer}

$\hat{R}_3$

$Disease^f$	$Specialist^{\mathcal{E}}$
18	{Hopkins}
99	{Jude}

$\hat{R}_1 \bowtie \hat{R}_2 \bowtie \hat{R}_3$

$Name^{\mathcal{E}}$	$City^{\mathcal{E}}$	$Disease^{\mathcal{E}}$	$Specialist^{\mathcal{E}}$
{Bob}	{London}	{AIDS}	{Hopkins}
{Eve}	{Tokyo}	{Cancer}	{Jude}

Fig. 9: Intuition of the SP approach. We denote an IND-CPA public encryption scheme by $\{\cdot\}$, and pseudo-random values by integers.

For both algorithms, we remark that the cloud knows when components of same attribute are equal since a pseudo-random function is deterministic. We see in Fig. 9 that the cloud knows that $\hat{R}_2$ and $\hat{R}_3$ share two same values of disease since values 18 and 99 are present in both relations. However, we notice that only the data owner knows the secret key k used by the pseudo-random function.

```
Algorithm: PreProc($R_1, \ldots, R_n$)
visited ← $\varnothing$;
for $1 \leqslant i \leqslant n$ do
    $\hat{R}_i \leftarrow \varnothing$;
    $\mathbb{R}_i^f \leftarrow \{A^f | A \in \mathbb{R}_i \cap \mathbb{X}\}$;
    $\mathbb{R}_i^{\mathcal{E}} \leftarrow \{A^{\mathcal{E}} | A \in \mathbb{R}_i \backslash \mathsf{visited}\}$;
    $\hat{\mathbb{R}}_i \leftarrow \mathbb{R}_i^f \cup \mathbb{R}_i^{\mathcal{E}}$;
    for $t \in R_i$ do
        $t_f \leftarrow \times_{A^f \in \mathbb{R}_i^f} f_k(\pi_A(t))$;
        $t_{\mathcal{E}} \leftarrow \times_{A^{\mathcal{E}} \in \mathbb{R}_i^{\mathcal{E}}} (\mathcal{E}_{\mathsf{pk}_u}(\pi_A(t)))$;
        $\hat{R}_i \leftarrow \hat{R}_i \cup \{t_f \times t_{\mathcal{E}}\}$;
    visited ← visited $\cup\ \mathbb{R}_i$;
```

Fig. 8: Preprocessing of relations.

3.3 SP n-ary Joins with MapReduce

SP Cascade Algorithm. If a relation participating at the i-th round contains an attribute that will participate to the join in a following round, the algorithm must anticipate the pseudo-random values of the shared attribute to perform joins. In the original cascade algorithm presented in Section 2.1, tuples are not encrypted and the anticipation is not necessary since each tuple value is available. In the SP approach, we add in value of pairs the pseudo-random evaluations of all needed pseudo-random values allowing joins in other rounds. This is possible since the preprocessing done by the data owner outsources protected relations containing pseudo-random evaluations of values of join attributes.

```
Algorithm:
 $\mathsf{SecBinary}(\hat{Q}_i, \hat{R}_{i+1}, i)$
Map function
Input: $(key, value)$
// $key$: id of chunk of $\hat{Q}_i/\hat{R}_{i+1}$
// $value$: coll. $t \in \hat{Q}_i$ or $t \in \hat{R}_{i+1}$
if $i = 1$ then
    foreach $t_q \in \hat{Q}_1$ do
        emit $(\pi_{\mathbb{Q}_1^f \cap \mathbb{R}_2^f}(t), (\mathtt{Q}_1, t_q))$;
foreach $t_r \in \hat{R}_{i+1}$ do
    emit $(\pi_{\mathbb{Q}_i^f \cap \mathbb{R}_{i+1}^f}(t), (\mathtt{R}_{i+1}, t_r))$.
Reduce function
Input: $(key, values)$
// $key$: $\pi_{\mathbb{Q}_i^f \cap \mathbb{R}_{i+1}^f}(t)$
// $values$: coll. of $(\mathtt{Q}_i, t_q)$ and
$(\mathtt{R}_{i+1}, t_r)$
for $(\mathtt{Q}, t_q) \in values$ do
    for $(\mathtt{R}, t_r) \in values$ do
        $t = t_q \times t_r$;
        if $i \neq n-1$ then
            emit
              $(\pi_{\mathbb{Q}_{i+1}^f \cap \mathbb{R}_{i+2}^f}(t), t)$;
        else
            emit $(t, t)$.
```

Fig. 11: SecBinary algorithm.

```
Algorithm:
 $\mathsf{SecHypercube}(\hat{R}_1, \ldots, \hat{R}_n)$
Map function
Input: $(key, value)$
// $key$: id of chunk of $\hat{R}_i$ for
$1 \leqslant i \leqslant n$
// $value$: collection of $t \in \hat{R}_i$
foreach $t \in \hat{R}_i$ do
    for $1 \leqslant \ell \leqslant d$ do
        if $X_\ell^f \in \mathbb{R}_i^f$ then
          $x_\ell \leftarrow h_\ell(\pi_{X_\ell^f}(t))$;
        else $x_\ell \leftarrow \star$;
    emit
      $((x_1, \ldots, x_d), (\mathtt{R}_i, t))$.
Reduce function
Input: $(key, values)$
// $key$: $(x_1, \ldots, x_d)$
// $values$: coll. of $(\mathtt{R}_i, t)$
for $1 \leqslant i \leqslant n$ do
    $R'_i \leftarrow \bigcup_{(\mathtt{R}_i, t) \in values} \{t\}$;
for $t \in R'_1 \bowtie \cdots \bowtie R'_n$ do
    emit $(t, t)$.
```

Fig. 12: SecHypercube algorithm.

We present in Fig. 10 the general process of the SP-cascade algorithm while we present the SP approach between relations $\hat{Q}_i$ and $\hat{R}_{i+1}$ participating at the i-th round in Fig. 11, where $\hat{Q}_i = \hat{Q}_{i-1} \bowtie \hat{R}_i$ and $\hat{Q}_1 = \hat{R}_1$.

```
Algorithm: $\mathsf{SecCascade}(\hat{R}_1, \ldots, \hat{R}_n)$
for $1 \leqslant i \leqslant n-1$ do
    $\hat{Q}_{i+1} \leftarrow \mathsf{SecBinary}(\hat{Q}_i, \hat{R}_{i+1}, i)$;
```

Fig. 10: SecCascade algorithm.

SP Hypercube Algorithm. We present the SP approach for the hypercube algorithm in Fig. 12. The main difference compared to the insecure approach is that the map function receives encrypted tuples from the data owner. As for the cascade algorithm, we add pseudo-random evaluations in value of each pair allowing the reduce function to check correspondences of tuples on join attributes.

3.4 CRSP n-ary Joins with MapReduce

We present the Collision-Resistant-Secure-Private (CRSP) approach for cascade and hypercube algorithms to compute joins between $n > 2$ relations with MapReduce. We recall that we assume in the CRSP approach that the user and the public cloud collude i.e., the public cloud knows the secret key sk_u of the user. Even in this scenario, we want that the security properties are satisfied.

When the public cloud and the user collude, the SP approach does not satisfy anymore the security properties since the public cloud can decrypt all tuples using the user's private key sk_u. Hence, the user learns intermediate results that she should not know, and property security (3) is not satisfied.

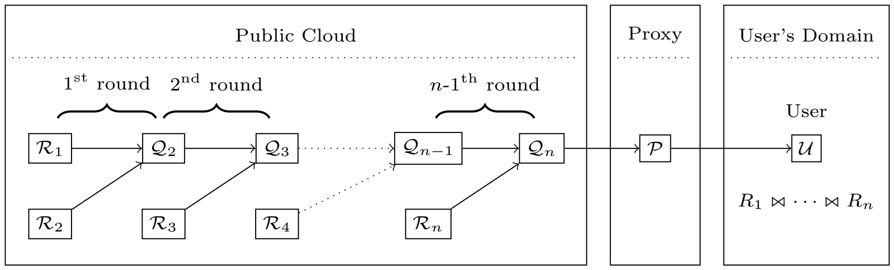


Fig. 13: CRSP n-ary joins with MapReduce.

To solve this issue, we introduce a trusted set of nodes as proxy (which do not collude with the public cloud and the user) denoted $\mathcal{P}$. This proxy has a key pair $(\mathsf{pk}_t, \mathsf{sk}_t)$. The public key pk_t is used by the data owner in the preprocessing phase. In this case the value of $t_{\mathcal{E}}$ is equal to $\bigtimes_{A^{\mathcal{E}} \in \mathbb{R}_i^{\mathcal{E}}} \mathcal{E}_{\mathsf{pk}_t}(\mathcal{E}_{\mathsf{pk}_u}(\pi_A(t)))$. In fact, the data owner encrypts (with the proxy public key) each encrypted values obtained with the user public key pk_u. This avoids the public cloud to decrypt the encrypted components outsourced in the cloud. Hence, the public cloud does the join computation as usually, and sends the result to the proxy. The proxy uses his secret key sk_t and sends the result only encrypted by the user's public key to the user. We illustrate the CRSP approach in Fig. 13.

CRSP Cascade Algorithm. The CRSP approach for the cascade algorithm between n relations uses the same algorithm than the SP approach and is presented in Fig. 10. The difference lies in the preprocessing where the data owner uses the proxy public key pk_t to encrypt the encrypted values obtained using the user public key. Hence, the public cloud cannot use the user's secret key sk_u to learn information about tuples. We stress that $\mathcal{P}$ is a trusted set of nodes i.e., the proxy colludes neither with the public cloud nor the user.

CRSP Hypercube Algorithm. The CRSP approach for the hypercube algorithm between n relations presented in Fig. 12 uses the same algorithm than the

SP approach. As for the cascade algorithm in the CRSP approach, the difference lies in a second encryption of values done by the data owner using the proxy public key. This second encryption avoids the public cloud to learn information on relations sent by the data owner, even if the cloud and the user collude.

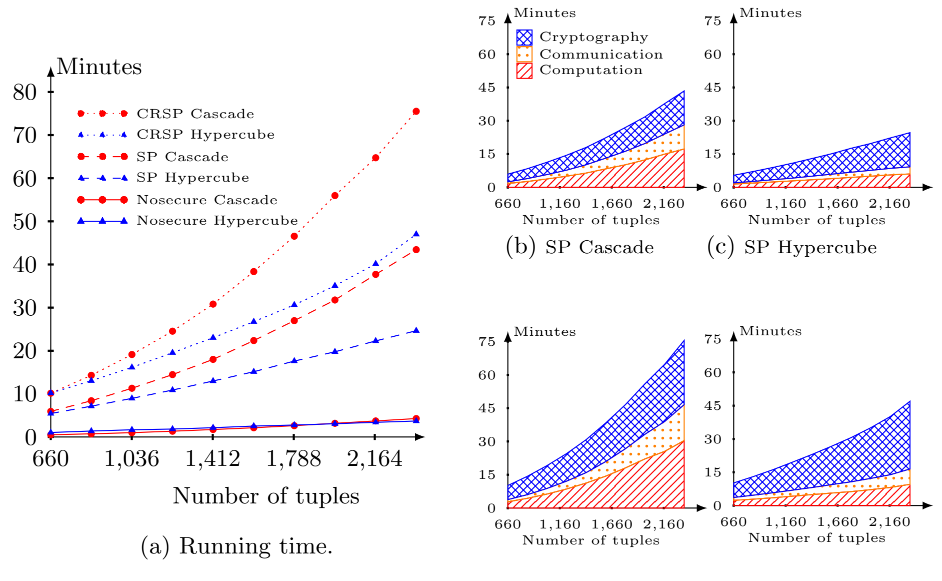
4 Experimental Results

We present the experimental results for the insecure, SP and CRSP approaches with the cascade and hypercube algorithms using the Hadoop [1] implementation of MapReduce. We have done all computations on a cluster running on Ubuntu Server 14.04 with Vanilla Hadoop 2.7.1 using Java 1.7.0. The cluster is composed of one master node and of three data nodes. The master node has four CPU cadenced to 2.4GHz, 80Gb of disk, and 8Gb of RAM. The three data nodes have of two CPU cadenced to 2.4GHz, 40Gb of disk, and 4Gb of RAM.

We use the real-world *Higgs Twitter Dataset* [2] that we denote by relation $R(A, B)$ where attributes A and B encode followee-follower relation on Twitter. The relation $R(A, B)$ has 15M tuples. To perform joins with this dataset, we generate two relations $S(B, C)$ and $T(C, A)$ that are copies of R. The join query used in our experiments is $R(A, B) \bowtie S(B, C) \bowtie T(C, A)$, consisting on all directed triangles of the *Higgs Twitter Dataset*. Using such a dataset and query is a standard practice in the database community literature to evaluate the performance of join query algorithms, as recently done e.g., in [9]. We use the AES encryption scheme [10] as the pseudo-random function, and the RSA-OAEP encryption scheme [5] as the public key encryption scheme.

Scalability. We present in Fig. 14a the running time for the cascade and hypercube algorithms, for each security approach. The different numbers of selected tuples come from the original dataset, where a sample is selected randomly. In the figures presented in Fig. 14, we consider size up to $2,356,225$ tuples because after such a size, our cluster gives out-of-memory errors hence we cannot compare meaningful results for all approaches. We report average times over five runs. For the hypercube algorithm, we use four buckets for each of the three dimensions defined by attributes A, B, and C, hence a total number of $4^3 = 64$ reducers. Without any security, the cascade and hypercube algorithms perform very similarly, although the hypercube seems a bit better for the largest input data sizes. For each of our secure approaches (SP and CRSP), the hypercube algorithm performs better than the cascade, hence the aforementioned trend is visible starting from small input data sizes. Intuitively, this happens because the hypercube avoids computing large intermediate results as may happen in practice when triangle queries are computed with a cascade approach.

Behind the curtain. We look in details at the main parts behind the algorithm execution for SP Cascade (Fig. 14b), SP Hypercube (Fig.14c), CRSP Cascade (Fig. 14d), and CRSP Hypercube (Fig. 14e). For each of the aforementioned cases, the cryptography is not the dominant cost, which confirms our intuition that the overhead needed to secure standard join algorithms is a constant factor. The communication and the computation dominate the total execution time.



(a) Running time. (b) SP Cascade (c) SP Hypercube (d) CRSP Cascade (e) CRSP Hypercube

Fig. 14: Running time and zoom-in on the different steps of our protocols. Number of tuples are expressed in thousands.

5 Security Proofs

We present briefly the security proofs of the cascade and the hypercube algorithms in our two approaches considering the *Random Oracle Model* (ROM). Complete proofs are given in the technical report available online [3]. We use the standard multi-party computations definition of security against honest-but-curious adversaries. We refer the reader to [19] for further details.

Theorem 1. *Assume f is a secure pseudo random function and Π is an IND-CPA public key encryption scheme. Then, the SP cascade and the SP hypercube algorithms securely computes joins between n relations in ROM in the presence of honest-but-curious adversaries if the cloud and the user do not collude.*

Proof (Sketch). We use the hybrid argument. For each protocol (SP cascade and SP hypercube), we first build a simulator Sim_1 where the pseudo random function is simulated by a random oracle. We show that it does not exist a polynomial-time algorithm such that it can distinguish the view of real protocols to the view of Sim_1 since we assume that f is a secure pseudo random function. Values of attributes are encrypted using a public key encryption scheme with the user's public key, hence we build a second simulator Sim_2 working as Sim_1 but where all encryptions are replaced by random values. We show that a distinguer can distinguish an execution of Sim_1 to an execution of Sim_2 only with a negligible probability if the public key encryption scheme is semantically secure. By transitivity, we prove that it does not exist a polynomial-time algorithm that can distinguish the view generated by real protocols and the view generated by the simulator Sim_2. Hence, if f is a secure pseudo random function and Π is an IND-CPA public key encryption scheme, then the SP cascade and the SP hypercube algorithms securely computes joins between n relations in ROM in the presence of honest-but-curious adversaries if the cloud and the user do not collude.

Theorem 2. *Assume f is a secure pseudo random function and Π is an IND-CPA public key encryption scheme. Then, the CRSP cascade and the CRSP hypercube algorithms securely computes joins between n relations in ROM in the presence of honest-but-curious adversaries even if the public cloud and the user collude.*

Proof (Sketch). First, we use the hybrid argument to show that it does not exist a polynomial-time algorithm that is able to distinguish the view of the cloud colluding with the user generated by the real protocols (CRSP cascade and CRSP hypercube) and the view generated by a simulator using inputs and outputs of the cloud and of the user. As in the previous proof, it relies on the secure pseudo random function f and on the IND-CPA public key encryption scheme. In the same way, we prove that we can perfectly simulate the view of the proxy using its input and output. Hence, if f is a secure pseudo random function and Π is an IND-CPA public key encryption scheme, the CRSP cascade and the CRSP hypercube algorithms securely computes joins between n relations in ROM in the presence of honest-but-curious adversaries even if the public cloud and the user collude.

6 Conclusion

We have presented two efficient approaches for computing joins with MapReduce. The SP approach assumes that the cloud and the user do not collude, whereas the CRSP approach resists to collusions, but needs more resources as it needs to communication with an honest proxy. We have thoroughly compared these two approaches with respect to their privacy guarantees and their practical performance using a standard real-world dataset.

As future work, we plan to integrate our secure join algorithms in a secure query optimizer system based on the MapReduce paradigm. We also aim at designing a protocol that is secure in the standard model and that not depends on a trusted third party.

Acknowledgements

This research was conducted with the support of the FEDER program of 2014-2020, the region council of Auvergne-Rhône-Alpes, the support of the “Digital

Trust” Chair from the University of Auvergne Foundation, the Indo-French Centre for the Promotion of Advanced Research (IFCPAR) and the Center Franco-Indien Pour La Promotion De La Recherche Avancée (CEFIPRA) through the project DST/CNRS 2015-03 under DST-INRIA-CNRS Targeted Programme.

References

1. Apache Hadoop. `https://hadoop.apache.org/`.
2. Higgs Twitter Dataset. `http://snap.stanford.edu/data/higgs-twitter.html`.
3. Secure Joins with MapReduce – Technical Report. `https://hal.archives-ouvertes.fr/hal-01903098`.
4. F. N. Afrati and J. D. Ullman. Optimizing Joins in a MapReduce Environment. In *EDBT*, 2010.
5. M. Bellare and P. Rogaway. Optimal asymmetric encryption. In *EUROCRYPT*, 1994.
6. E. Blass, R. D. Pietro, R. Molva, and M. Önen. PRISM - privacy-preserving search in mapreduce. In *PETS*, 2012.
7. X. Bultel, R. Ciucanu, M. Giraud, and P. Lafourcade. Secure matrix multiplication with mapreduce. In *ARES*, 2017.
8. S. S. M. Chow, J. Lee, and L. Subramanian. Two-party computation model for privacy-preserving queries over distributed databases. In *NDSS*, 2009.
9. S. Chu, M. Balazinska, and D. Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *SIGMOD Conference*, 2015.
10. J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.
11. J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.
12. P. Derbeko, S. Dolev, E. Gudes, and S. Sharma. Security and privacy aspects in mapreduce on clouds: A survey. *Computer Science Review*, 20, 2016.
13. S. Dolev, N. Gilboa, and X. Li. Accumulating automata and cascaded equations automata for communicationless information theoretically secure multi-party computation: Extended abstract. In *ASIACCS*, 2015.
14. S. Dolev, Y. Li, and S. Sharma. Private and secure secret shared mapreduce. In *DBSec*, 2016.
15. T. El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *CRYPTO*, 1985.
16. F. Emekçi, D. Agrawal, A. El Abbadi, and A. Gulbeden. Privacy preserving query processing using third parties. In *ICDE*, 2006.
17. S. Laur, R. Talviste, and J. Willemson. From oblivious AES to efficient and secure database join in the multiparty setting. In *ACNS*, 2013.
18. J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of Massive Datasets, 2nd Ed.* Cambridge University Press, 2014.
19. Y. Lindell, editor. *Tutorials on the Foundations of Cryptography*. Springer International Publishing, 2017.
20. T. Mayberry, E. Blass, and A. H. Chan. PIRMAP: efficient private information retrieval for mapreduce. In *FC*, 2013.
21. R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *SOSP*, 2011.

22. R. A. Popa and N. Zeldovich. Cryptographic Treatment of CryptDB's Adjustable Join. 2012.
23. A. Shamir. How to share a secret. *Commun. ACM*, 22(11), Nov. 1979.
24. T. D. Vo-Huu, E. Blass, and G. Noubir. Epic: Efficient privacy-preserving counting for mapreduce. In *NETYS*, 2015.