TrustedMR: A Trusted MapReduce System based on Tamper Resistance Hardware

Quoc-Cuong To Benjamin Nguyen Philippe Pucheral
SMIS Project, INRIA Rocquencourt, 78153 Le Chesnay, France
PRiSM Laboratory, 45, Av. des Etats-Unis, 78035 Versailles, France
<Fname.Lname>@inria.fr , <Fname.Lname>@prism.uvsq.fr

ABSTRACT

With scalability, fault tolerance, ease of programming, and flexibility, MapReduce has gained many attractions for large-scale data processing. However, despite its merits, MapReduce does not focus on the problem of data privacy, especially when processing sensitive data on untrusted Mappers/Reducers. This paper proposes *TrustedMR*, a trusted MapReduce system based on the *Trusted Cells* with high security assurance provided by tamper-resistant hardware, to enforce the security aspect of the MapReduce. TrustedMR pushes the security to the edge of the network where data is produced and encrypted data can be processed mostly on untrusted servers without any modification to the existing MapReduce framework. Our evaluation shows that the performance overheads of TrustedMR can easily be managed to within only few percents, compared to original MapReduce framework that handles cleartexts.

Keywords

Privacy-preserving, decentralized, MapReduce, trusted hardware.

1. INTRODUCTION

We are witnessing an exponential accumulation of personal data on servers: data stored by administrations, hospitals, insurance companies; data automatically acquired by web sites, sensors and smart meters; and even digital data owned or created by individuals (e.g., photos, agendas, invoices, etc), end up in the Cloud for convenience and efficiency. Personal data has become the new oil of the Internet and is monopolized by online services [17]. To process these large-scale data, MapReduce framework [11] stands out being the most popular solution due to its scalability, fault tolerance, ease of programming, and flexibility. With MapReduce, developers can solve various cumbersome tasks of distributed programming without the need to write complicated codes. Indeed, a developer simply writes a map and a reduce function. The system automatically distributes the workload over a cluster of commodity machines, monitors the execution, and handles failures. Current trends show that MapReduce is considered as a high-productivity alternative to traditional parallel programming paradigms for a variety of applications, ranging from enterprise computing to peta-scale scientific computing[1]. For example, power meter data can be used by the national distribution company (e.g., EDF company in France) to enable new services and products for customers. The volume of data created by energy networks is substantial, leading companies like SunEdison into big data modeling and analytics (e.g., going from one meter reading a month to smart meter readings every 15 minutes results in a huge increase data volume that must be efficiently handled). Since raw data can be highly sensitive (e.g., at the 1HZ granularity provided by the French Linky power meters, most electrical appliances have a distinctive energy signature. It is thus possible to infer from the power meter data inhabitants activities [15]), it must be protected.

However, MapReduce was born to meet the demand of performance in processing big data, and it is still missing the function of protecting user's sensitive data from untrusted mappers/reducers. Although some state-of-the-art works have been proposed to focus on the security aspect of MapReduce, none of them aims at data privacy. They only solve the problem of integrity verification [23, 20] and have some weak security assumptions about untrusted servers (e.g., they require that Reducers must be trusted [19]). Furthermore, these works often require some modifications to the original MapReduce framework to enforce the system's security (e.g., [19] have to modify the original MapReduce framework to support the mandatory access control).

To preserve user's privacy, some centralized cryptography-based solutions have been proposed. Nevertheless, it has become clear that centralizing and processing all one's data in a single server is a major problem with regards to privacy concerns. Indeed, privacy violations arising from negligence, abusive use or attacks are many[2] and no current approach, including cryptography based and server-side secure hardware [12,22,5], seems capable of closing the gap. Consequently, several attempts of personal data management decentralization have appeared. To cite a few, a personal data server [1] is embedded in a tamper-proof token to securely manage the personal data of a user. In [17], the authors propose the implementation of an Open Personal Data Store on top of the Cloud. In [3], they propose a global, decentralized data platform, called Trusted Cells, which represents a sea change in the acquisition and protection of personal data. This vision pushes the security to the edges of the network, through personal data servers [1] running on secure smart phones, set-top boxes, plug computers[3] or secure portable tokens[4]. Based on the Trusted Cells' architecture, [21] proposes secure distributed protocols to perform global computations without revealing any sensitive information to central servers. All these initiatives tend to provide a better control of the user on the storage, management and sharing of her personal data, as requested by the World Economic Forum [29] and by legislations protecting the use of personal data around the globe (e.g., [28]).

Based on the Trusted Cells architecture and the protocol proposed in [21], this paper proposes a MapReduce-based system, addressing the following three important issues that every secure system must meet:

[1] http://skynet.rubyforge.org

[2] http://www.datalossdb.org

[3] freedomboxfoundation.org

[4] www.gd-sfs.com/portable-security-token

1. ***Security***: How to process data using MapReduce framework without revealing sensitive information to untrusted mappers/reducers?

2. ***Utility/Functionality***: How many types of operations (e.g., types of SQL queries) the proposed system can support? Can the system support key-value pair problem?

3. ***Performance***: How to process large amount of encrypted data using MapReduce with small overhead, compared with performance in processing cleartext data?

To solve this problem, we consider the asymmetric architecture where personal data is produced, and kept in distributed Trusted Data Servers (TDSs) embedded in secure devices. Then, to process this data, each TDS's owner sends his data to mappers/reducers. To ensure the data privacy, data will be obfuscated appropriately so that MapReduce framework can process encrypted data as much as possible while still maintaining the privacy. To ensure the utility, we transfer the encrypted data to TDSs to decrypt and compute. Since TDSs compute on the cleartext, it can support any functions. To ensure the performance, especially when we have to transfer large amount of data to TDSs, we employ the parallel computing where each mapper/reducer splits big data into smaller ones and transfers to multiple TDSs so that they can process in parallel, reducing the transferring and computing time. We assume that each personal store can trust others, regardless of the origin of this trust. This assumption capitalizes on emerging practices and technological advances which can no longer be ignored. In the FreedomBox context, trust takes its root in the decentralization of the platform (up to the individual) and in simple plug computers running open-source code of basic functionality. Alternatively, growing families of client devices are now reaching very high security standards thanks to secure hardware. For example, the use of *smart tokens* is actively investigated by many countries for healthcare and e-governance applications [1]. Smart tokens have different form factors (e.g., SIM card, USB token, Secure MicroSD) and names but share similar characteristics (low cost, high portability, high storage capacity, high security), introducing a real breakthrough in the management of personal data. But smart tokens no longer hold the monopoly of client-side hardware security. TrustZone technology pushed by ARM is disrupting the design of secure systems by pushing hardware security to any mobile device (e.g., tablets, smartphones) equipped with Cortex-A processors[5]. According to ARM, a full Trusted Execution Environment (TEE) will soon be present in any client device at low cost. In this paper, and up to the experiments section, we consider that personal data stores are hosted by secure devices but make no additional assumption regarding the technical solution they rely on.

Hence, the contribution of this paper is to propose a secure MapReduce-based system that can: (1) preserve data's privacy from untrusted mappers/reducers, (2) support unlimited types of operations, key/value pair problems and (3) have acceptable and controllable performance overhead.

The rest of this paper is organized as follows. Section 2 discusses related works. Section 3 states our problem. Section 4 presents our proposed solution. Section 5 analyses the security. Section 6 measures the performance and section 7 concludes.

[5] www.arm.com/products/processors/technologies/trustzone.php

2. RELATED WORKS

This work has connections with related studies in different domains, namely security in MapReduce, secure hardware at client/server side and finally outsourced database services-DaaS without secure hardware. We review these works below.

2.1 Security in MapReduce

The related works address different security aspects of MapReduce as follows.

MAC and differential privacy: [19] proposes the Airavat that integrates mandatory access control with differential privacy in MapReduce framework. Since Airavat adds noise to the output in the reduce function to achieve differential privacy, it requires that reducers must be trusted. Furthermore, the types of computation supported by Airavat are limited (e.g., SUM, COUNT). If they want to support more kinds of computation, the mappers must also be trusted. The other drawback of Airavat is that the security mechanisms, including the integrity verification mechanisms, are implemented inside the open infrastructure — that is, they are still services provided by the infrastructure. Hence, their trustworthiness (i.e. whether they are enforced as expected) should still be verified. Although Airavat does not trust the computation provider who writes the map and reduce functions, it does trust the cloud provider and the cloud computing infrastructure. Finally, they have to modify the original MapReduce framework to support the mandatory access control.

Integrity verification: In other directions, [23] replicates some map/reduce tasks and assign them to different mappers/reducers to validate the integrity of map/reduce tasks. Any inconsistent intermediate results from those mappers/reducers reveal attacks. However, even if those malicious mappers/reducers ensure the data integrity, they cannot preserve the data privacy since the mappers/reducers directly access to sensitive data in cleartexts. Recent research [20] also focuses on the integrity verification, but missing the data privacy. So, these works are orthogonal to our works in which we aim at protecting the data privacy.

Data anonymization: [26] claims that it is challenging to process large-scale data to satisfy k-anonymity in a tolerable elapsed time. So they anonymize data sets via generalization to satisfy k-anonymity requirement in a highly scalable way using MapReduce. Data sets are partitioned and anonymized in parallel in the first phase, producing intermediate results. Then, the intermediate results are merged and further anonymized to produce consistent k-anonymous data sets in the second phase.

Hybrid Cloud: In stating that the data can be classified into secure and public data, some works [24, 25] propose the hybrid cloud including the private cloud and the public cloud. The main idea is to split the task, keeping the computation on the private data within an organization's private cloud while moving the rest to the public commercial cloud. Sedic [24] automatically partitions a job according to the security levels of the data and tries to outsource as much workload to the public commercial cloud as possible, given sensitive data always stay on the private cloud. However, this solution requires that reduction operations must be associative and the original MapReduce framework must be modified. Also, the sanitization approach taken by Sedic does not fit well with chained or iterative MapReduce, may still reveal relative locations and length of sensitive data, which could lead to crucial information leakage in certain applications [25]. To overcome this weakness, [25] proposes tagged-MapReduce that augments each key-value pair in MR with a sensitivity tag.

However, both solutions are not suitable for MapReduce job where all data is sensitive and/or data owner does not want to reveal any data.

Encrypting part of dataset: In arguing that encrypting all data sets in cloud is not effective, [27] proposes an approach to identify which intermediate data sets need to be encrypted while others are in cleartexts, in order to be cost-effective while the privacy requirements of data holders can still be satisfied. The main idea is that the data with high frequency of accessing will be encrypted while the others are unencrypted. This solution is not suitable for the case where all data have the same frequency of accessing or data owner does not want to reveal even a single tuple to untrusted cloud.

Private information retrieval (PIR): To hide the user's access pattern in retrieving large files from an untrusted cloud, [16] proposes PIRMAP that is particularly suited to MapReduce to achieve good communication complexity with query times significantly faster than traditional PIR. This is contradictory to our work where we aim at data privacy but not user's access pattern.

Other works **support very specific operations.** [7] searches encrypted keywords on the cloud so that the cloud must not learn any information about the content it hosts and search queries performed. [6] presents EPiC to count the number of occurrences of a pattern specified by user in an oblivious manner on the untrusted cloud. In contrast to these works, our work addresses more general problems, supporting any kind of operations.

2.2 Security in other Systems

Secure hardware at server side: Some works [5, 4] deploy the secure hardware at server side to ensure the confidentiality of the system. By leveraging server-hosted tamper-proof hardware, [5] designs TrustedDB, a trusted hardware based relational database with full data confidentiality and no limitations on query expressiveness. However, TrustedDB does not deploy any parallel processing, limiting its performance. [4] also bases on the trusted hardware to securely decrypt data on the server and perform computations in plaintext. In this setting, since the data access pattern from untrusted storage has the potential to reveal sensitive information, they present oblivious query processing algorithms so that an adversary observing the query execution learns nothing about the underlying database.

Secure hardware at client side. Even equipped with secure hardware on server with strong encryption, [5, 4] does not solve the two intrinsic problems of centralized approaches. First, users get exposed to sudden changes in privacy policies. Second, users are exposed to sophisticated attacks, whose cost-benefit is high on a centralized database [3]. So some works [21, 2, 3] are based on secure hardware at client side to solve these problems. The work in [2] proposes a generic Privacy-Preserving Data Publishing protocol composed of low cost secure tokens and a powerful but untrusted supporting server. The main purpose of this protocol is to publish different sanitized releases to recipients. With similar architecture, [21] proposes distributed querying protocols to compute general queries while maintaining strong privacy guarantees.

Centralized DaaS without secure hardware: Outsourced database services or DaaS allow users to store sensitive data on a remote, untrusted server and retrieve desired parts of it on request. Many works [18,22] have addressed the security of DaaS by encrypting the data at rest and pushing part of the processing to the server side but none of them can achieve all aspects of security, utility, and performance. In terms of utility and security, the best approach would be to consider theoretical solutions such as fully homomorphic encryption [12], which allows servers to compute arbitrary functions over encrypted data, while only clients see decrypted data. However, this construction is prohibitively expensive in practice, requiring slowdowns on the order of $10^9\times$ [22]. In term of performance, CryptDB [18] is a system that provides provable confidentiality by executing SQL queries over encrypted data using a collection of efficient SQL-aware encryption schemes. However, this system is not completely secure since it still uses some weak encryption schemes (e.g., deterministic encryption, order-preserving encryption [8]). Similarly, [22] proposes the MONOMI system that securely executes arbitrarily complex queries over sensitive data on an untrusted database server with a median overhead of only 1.24× compared to an un-encrypted database. However, this system still uses some weak encryption schemes (e.g., deterministic encryption) to perform some SQL operations (e.g., GROUP BY, equi-join).

As a conclusion, and to the best of our knowledge, no state-of-the-art MapReducre works can satisfy the three requirements of security, utility, performance, and our work is the first MapReduce-based proposal, that inherits the strong privacy guarantees from [21], achieving a secure solution to process large-scale encrypted data using a large set of tamper-resistant hardware with low performance overhead. In other words, our MapReduce solution meets the three requirements above.

3. CONTEXT OF THE STUDY

3.1 Architecture

The architecture we consider is decentralized by nature. As pictured in Fig. 1, each individual is assumed to manage her data by means of a Trusted Data Server embedded in a secure device. We make no assumption about how this data is actually gathered and refer the reader to other papers addressing this issue [1, 17]. We detail next the main components of the architecture.

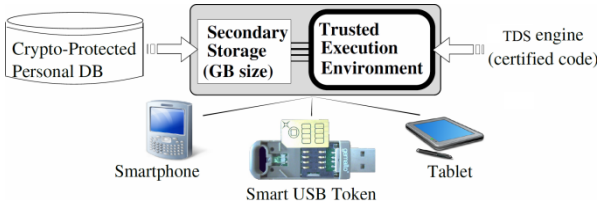


Fig. 1. Secure Devices

The Trusted Data Servers (TDSs). A TDS (as defined in [1]) is a DBMS engine embedded in an individual's secure device. It manages the individual's personal data and can participate in distributed queries while enforcing access control rules and opt-in/out choices of the individual. A TDS inherits its security from the Secure Device hosting it. Despite the diversity of existing hardware solutions, a Secure Device can be abstracted by (1) a Trusted Execution Environment and (2) a (potentially untrusted) mass storage area. E.g., the former can be provided by a tamper-resistant microcontroller while the latter can be provided by Flash memory (see Fig. 1). The important assumption is that code executed by the Secure Device cannot be tampered. This given, the contents of the mass storage area can be protected using cryptographic protocols. Each Secure Device exhibits the following properties:

High Security. This is due to a combination of factors: (1) the microcontroller tamper-resistance, making hardware and side-channel attacks highly difficult, (2) the certification of the embedded code making software attacks also highly difficult, (3) the ability to be auto-administered, in contrast with traditional multi-user servers, precluding DBA attacks, and (4) the fact that the device holder cannot directly access the data stored locally (she must authenticate and can only access data according to her own privileges). This last point is of utmost importance because it allows the definition of distributed protocols where data is securely exchanged among TDSs with no confidentiality risk.

Low Availability. The Secure Device is physically controlled by its owner who may connect or disconnect it at will, providing no availability guarantee.

Modest Computing Resource. Most Secure Devices provide modest computing resources (see section 6) due to the hardware constraints linked to their tamper-resistance. On the other hand, a dedicated cryptographic co-processor usually handles cryptographic operations very efficiently (e.g., AES and SHA).

Hence, even if there exist differences among Secure Devices (e.g., smart tokens are more robust against tampering but less powerful than TrustZone devices), all provide *much stronger security guarantees* combined with a *much weaker availability and computing power* than any traditional server.

The MapReduce Server. Because of their low computing capacity and connectivity, TDSs need a powerful and highly available Supporting Server running MapReduce framework to provide communication, intermediate storage and global processing services that TDSs cannot provide on their own. Because mappers/reducers are implemented on regular server(s), e.g., in the Cloud, it exhibits the following properties: (1) *Low Security*, because mappers/reducers can be compromised by internal and external attacks, (2) *24/7 availability* and (3) *High Computing Resources.*

3.2 Threat Model

TDSs are the unique element of trust in the architecture and are considered *honest*. No trust assumption needs to be made on the querier either because (1) TDSs will not accept to participate to queries sent by a querier with insufficient privileges and (2) the querier can gain access only to the final result of the query computation (not to the raw data), as in traditional database systems. Preventing inferential attacks by combining the result of a sequence of authorized queries as in statistical databases and PPDP work is orthogonal to this study.

The potential adversary is consequently the mappers/reducers. We consider *honest-but-curious* mappers/reducers (i.e., which try to infer any information they can but strictly follows the protocol). Considering *malicious* mappers/reducers (i.e., which may tamper the protocol with no limit, including denial-of-service) is of little interest to this study. Indeed, a malicious mappers/reducers is likely to be detected with an irreversible political/financial damage and even the risk of a class action (remember that mappers/reducers is in the Cloud service provider).

4. PROPOSED SOLUTION

4.1 MapReduce Job Execution Phases

The MapReduce programming model consists of a map(k_1; v_1) function and a reduce(k_2; list(v_2)) function. The map(k_1; v_1) function is invoked for every key-value pair <k_1; v_1> in the input data to output zero or more key-value pairs of the form <k_2; v_2>. The reduce(k_2; list(v_2)) function is invoked for every unique key k_2 and corresponding values list(v_2) in the map output. reduce(k_2; list(v_2)) outputs zero or more key-value pairs of the form <k_3; v_3>. The MapReduce programming model also allows other functions such as (i) partition(k_2), for controlling how the map output key-value pairs are partitioned among the reduce tasks, and (ii) combine(k_2; list(v_2)), for performing partial aggregation. The keys k_1, k_2, and k_3 as well as the values v_1, v_2, and v_3 can be of different and arbitrary types. The detail of map and reduce tasks is depicted in Figure 2.

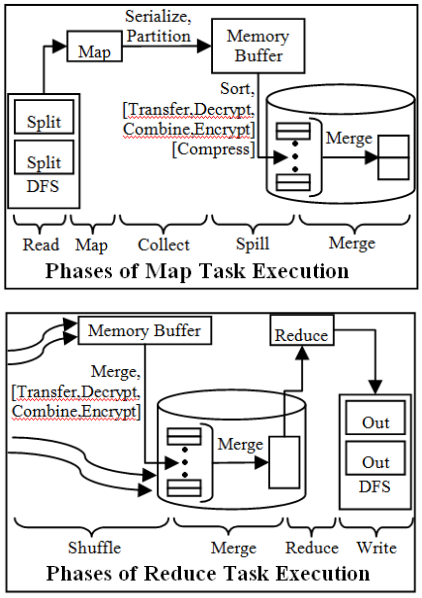


Fig. 2. Detail execution of map and reduce task [13]

In the next section, we propose a solution so that we do not need to modify this original model. We use the encryption scheme to allow the untrusted mappers/reducers participate in the computation as much as possible and transfer the necessary computations that cannot be processed on server to TDSs. These transfer and computation on TDSs happen in parallel to speed up the running time.

4.2 Proposed Solution

Our proposed solution inherits the histogram-based solution, called *ED_Hist*, proposed in [21]. Informally speaking, to prevent the frequency-based attack on deterministic encryption (*dEnc* for short) that encrypts the same cleartexts into the same ciphertexts, and to allow untrusted server group and sort the encrypted tuples (that have the same plaintext values) into the same partitions, ED_Hist transforms the original distribution of grouping attributes, called A_G, into a *nearly equi-depth histogram* (due to the data distribution, we cannot have exact equi-depth histogram). A nearly equi-depth histogram is a decomposition of the A_G domain into buckets holding *nearly* the same number of true tuples. Each bucket is identified by a hash value giving no information about the position of the bucket elements in the domain. Figure 3.a shows an example of an original distribution and Figure 3.b is its nearly equi-depth histogram.

There are three benefits in using nearly equi-depth histogram: i) allow mappers/reducers participate in the computation as much as possible (i.e., except the combine and reduce operations, all other

operations can be processed in ciphertexts), without modifying the existing MapReduce framework; ii) better balance the load among mappers/reducers for skewed dataset; and iii) prevent frequency-based attack.

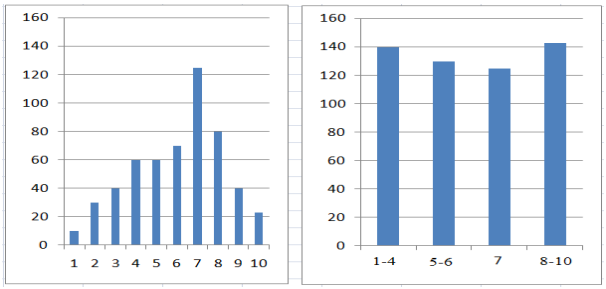


Fig. 3. Example of nearly equi-depth histogram

The protocol is divided into three tasks (see Figure 2 & 4).

Collection Task: Each TDS allocates its tuple(s) to the corresponding bucket(s) and sends to mappers/reducers tuples of the form ($F(k)$, $nEnc(u)$) where F is the mapping function that maps the keys to corresponding buckets:

$$bucketId = F(k)$$

and *nEnc* is the non-deterministic encryption that can encrypt the same cleartext into different ciphertext.

Assume the cardinality of k is n, and F maps this domain to b buckets, then we have:

$$B_1 = F(k_{11}) = F(k_{12}) = \ldots = F(k_{1d})$$
$$B_2 = F(k_{21}) = F(k_{22}) = \ldots = F(k_{2e})$$
$$\ldots$$
$$B_b = F(k_{b1}) = F(k_{b2}) = \ldots = F(k_{bz})$$

From that, the average number of distinct plaintext in each bucket is:

$$h = (d+e+\ldots+z)/b = n/b$$

When this task stops, all the encrypted data sent by TDSs are stored in DFS, and are ready for processed by mappers/reducers

Map Task: This task is divided into five phases:

1. Read: Read the input split from DFS and create the input key-value pairs: $(B_1, nEnc(u_1)), (B_2, nEnc(u_2)), \ldots (B_b, nEnc(u_m))$.

2. Map: Execute the user-defined map function to generate the map-output data: $map(B_i; nEnc(u_i)) \rightarrow (B'_i; nEnc(v_i))$. If the map function needs process complex functions that cannot be done on encrypted data (i.e., $v_i = f(u_i)$), connections to TDSs will be established to process these encrypted data.

3. Collect: Partition and collect the intermediate (map-output) data into a buffer before spilling.

4. Spill: Sort, if the combine function is specified: parallel transfer encrypted data to TDSs to decrypt, combine, encrypt, and return to mappers, perform compression if specified, and finally write to local disk to create file spills.

5. Merge: Merge the file spills into a single map output file. Merging might be performed in multiple rounds.

Reduce Task: This task includes four phases:

1. Shuffle: Transfer the intermediate data from the mapper nodes to a reducer's node and decompress if needed. Partial merging and combining may also occur during this phase.

2. Merge: Merge the sorted fragments from the different mappers to form the input to the reduce function.

3. Reduce: Execute the user-defined reduce function to produce the final output data. Since the reduce function can be arbitrary, and therefore encrypted data cannot be executed in reducers, they must be transferred to TDSs to be decrypted, executed the reduce function, encrypted, and returned to reducers. The difference between the output of the reduce function of traditional MapReduce with TrustedMR is that each input key represents different cleartext values, so the output key of the reduce function also represents different values: $(B'_1; list(nEnc(v_1)) \rightarrow (nEnc(k_{11}); nEnc(f(v_{1i}))), \ldots, (nEnc(k_{1d}); nEnc(f(v_{1m})))$.

4. Write: Compressing, if specified, and writing the final output to DFS.

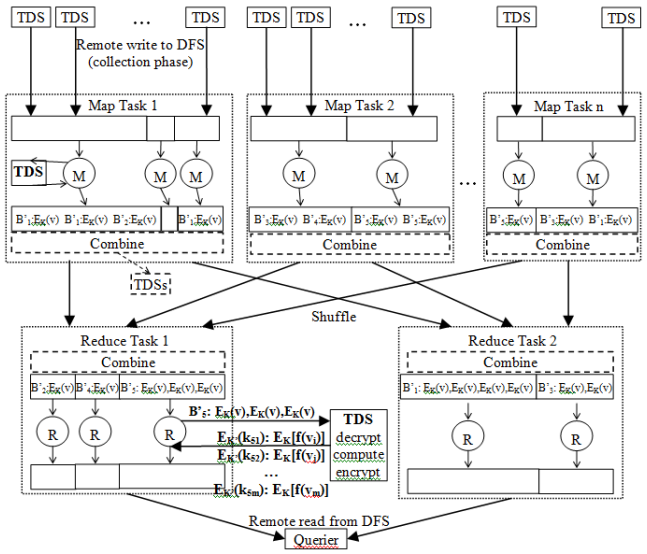


Fig. 4. Proposed solution

```
method Map (bucket B_i; encrypted value nEnc(u_i))
1. emit(bucket B'_i, nEnc(v_i))

method Combine (bucket B'_i; list [nEnc(v_1), nEnc(v_2),..])
1. form the partition: nEnc(v_1), nEnc(v_2),..nEnc(v_p)
2. create connection and send data to TDSs
3. in each TDS:
4.     unmap bucket: F^-1(B'_i) -> k_i1 , k_i2, .., k_in
5.     decrypt nEnc(v_i) -> v_i
6.     compute r_ij = f(v_i) having the same k_ij
7.     encrypt result r_ij -> nEnc(r_ij)
8.     map to bucket: F(k_i1) = F(k_i2) =..= F(k_in) = B'_i
9. emit (bucket B'_i; nEnc(r_ij))

method Reduce (bucket B'_i; list [nEnc(r_ij),..])
1-7. similar to Combine function from step 1 to 7
8.   emit (nEnc(k_ij); nEnc(r'_ij))
```

Fig. 5. Map, Combine, and Reduce methods

Among all phases in both map and reduce tasks, with the plaintext data mapped using the ED_Hist, the existing MapReduce framework can be used without being modified because each mappers/reducers can do all operations (i.e, map, partition, collect, sort, compress, merge, shuffle) on the mapped data, except the combine and reduce function. Since the combine and

reduce functions must process on cleartexts, encrypted data are transferred back to TDSs for decrypting, computing, encrypting the result and returning to mappers/reducers. To reduce the overhead of transferring large amount of data between TDSs and mappers/reducers, each mappers/reducers split the data into smaller pieces and send it in parallel to multiple TDSs. With this way, the transferring time is reduced. Fig. 5 is the pseudocode for map/reduce function.

Note that it is not possible to do the whole map and reduce tasks within TDS because the modest computing resource of TDS does not allow deploying the Hadoop. Also, data transfer between mappers/reducers and TDS are mandatory to keep the Hadoop framework unchanged. So, low power TDSs cannot do more than contributing to the internal execution of the map and reduce tasks.

4.3 How Our Proposed Solution Meets the Requirements

Informally speaking, the security, utility and efficiency of the protocol are as follows (we formally prove the efficiency and security in the next sections):

Security. Since TDSs map the attributes to nearly equi-depth histogram, mappers/reducers cannot launch any frequency-based attack. What if mappers/reducers acquire a TDS with the objective to get the cryptographic material (i.e., a sort of collusion attack between mappers/reducers and a TDS)? As stated in section 3, TDS code cannot be tampered, even by its holder. Whatever the information decrypted internally, the only output that a TDS can deliver is a set of encrypted tuples, which does not represent any benefit for mappers/reducers.

Utility. Since the data is processed by trusted TDSs in cleartext, our solution can support any operations.

Performance. The efficiency of the protocol is linked to the parallel computing of TDSs. Both the collection task and combine, reduce operations are run in parallel by all connected TDSs and no time-consuming task is performed by any of them. As the experiment section will clarify, each TDS manages incoming partitions in streaming because the internal time to decrypt the data and perform the computation is significantly less than the time needed to download the data. By combining the parallel computing, streaming data, and the crypto processor that can handles cryptographic operations efficiently in TDSs, our distributed model has acceptable and controllable performance overhead as pointed out in experiment.

Beside the three essential requirements above, our proposed solution meets other criteria as well: integrability and correctness.

Integrability: Because we do not need to modify the original MapReduce framework, our proposed solution can easily integrate with the existing framework. ED_Hist helps mappers/reducers run on encrypted data exactly as if they run on cleartext data without modifying the original MapReduce framework (i.e., as pointed out in section 4.2, the only tasks that mappers/reducers cannot run on encrypted data are *combine* and *reduce*).

Correctness. Since mappers/reducers are honest-but-curious, it will strictly follow the protocol and deliver to the querier the final output. Unlike the differential privacy, mappers/reducers do not sanitize the output (to achieve the differential privacy), so the final output is correct. If a TDS goes offline in the middle of processing a partition, and therefore cannot return result as expected, mappers/reducers will resend that partition to another available TDS after waiting the response from disconnected TDS a specific interval.

5. SECURITY ANALYSIS

We analyze the privacy in two levels of adversary's knowledge. In the first level, we assume that adversary does not know the distribution within a bucket but only the global distribution (section 5.1). In the higher level, we address stronger attackers with more knowledge in which he knows the probability distribution of the values within each bucket (section 5.2).

5.1 Information Exposure with Coefficient

To quantify the confidentiality of each algorithm, we measure the information exposure of the encrypted data they reveal to untrusted server by using the exposure coefficient, noted ε, introduced in [10]. Roughly speaking, the exposure coefficient is the probability that an attacker can reconstruct the plaintext table (or part of the table) by using the encrypted table and his prior knowledge about global distributions of plaintext attributes.

For the sake of conciseness, we give below only the result of the comparison (see Figure 6) and refer the interested readers to [21] for a more detailed analysis. In conclusion, *nDet_Enc* is the most secure encryption scheme. To reach the same highest security level as *nDet_Enc*, *ED_Hist* must pay a higher price. Specifically, *ED_Hist* must have a significant collision factor. Hence, as usual, there exists a trade-off between security and performance and the expected balance can be reached in each protocol by tuning a specific parameter (i.e., number of histograms in *ED_Hist*).

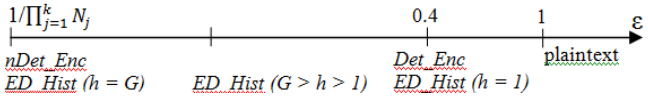


Fig. 6. Information exposure among encryption schemes

In conclusion, the information exposures of *nDet_Enc, Det_Enc* and *ED_Hist* have the following order: $\varepsilon_{nDec\_Enc} \leq \varepsilon_{ED\_Hist} \leq \varepsilon_{Dec\_Enc} < 1$, meaning that *ED_Hist* is the intermediate between *nDet_Enc* and *Det_Enc*.

5.2 Privacy Measure using Variance

In this section, we propose a stronger assumption that the adversary (A for short) possesses more knowledge of encrypted dataset than the previous section: *A knows the entire bucketization scheme and the exact probability distribution of the values within each bucket*. For example, given that bucket B has 10 elements, we assume A knows that: 3 of them have value 85, 3 have value 87 and 4 have value 95, say. However, since the elements within each bucket are indistinguishable, this does not allow A to map values to elements with absolute certainty. Then, the A's goal is to determine the precise values of sensitive attributes of some (all) individuals (records) with high degree of confidence. Eg: What is the value of salary field for a specific tuple? [14] proposes the **Variance** of the distribution of values within a bucket B as its measure of privacy guarantee. They first define the term *Average Squared Error of Estimation (ASEE)* as follows.

Definition ASEE: Assume a random variable X_B follows the same distribution as the elements of bucket B and let P_B denote its probability distribution. For the case of a discrete (continuous) random variable, we can derive the corresponding probability mass (density) function denoted by p_B. Then, the goal of the adversary is to estimate the true value of a random element chosen from this bucket. We assume that A employs a statistical

estimator for this purpose which is, itself a random variable, X'_B with probability distribution P'_B.

In other words, A guesses that the value of X'_B is x_i, with probability $p'_B(x_i)$. If there are N values in the domain of B, then we define *Average Squared Error of Estimation (ASEE)* as:

$$ASEE(X_B, X'_B) = \sum_{j=1}^{N}\sum_{i=1}^{N} p'_B(x_i) * p_B(x_j) * (x_i - x_j)^2$$

Theorem [14]: ***ASEE(X, X') = Var(X) + Var(X') + (E(X) - E(X'))^2*** *where X and X' are random variables with probability mass (density) functions p and p_0, respectively. Also* ***Var(X)*** *and* ***E(X)*** *denote variance and expectation of X respectively*.

Proof: refer to [14].

Note that unlike coefficient exposure, the smaller value of *ASEE* implies the bigger security breach because the distance between guessed values and actual values is smaller, and vice versa. So the adversary tries to minimize *ASEE* as much as he can. From the theorem above, it is easy to see that A can minimize $ASEE(X_B, X'_B)$ in two ways: 1) by reducing $Var(X'_B)$ or 2) by reducing the absolute value of the difference $E(X_B) - E(X'_B)$. Therefore, the best estimator of the value of an element from bucket B that A can get, is the constant estimator equal to the mean of the distribution of the elements in B (i.e., $E(X_B)$). For the constant estimator X'_B, $Var(X'_B) = 0$. Also, as follows from basic sampling theory, the "mean value of the sample-means is a good estimator of the population (true) mean". Thus, A can minimize the last term in the above expression by drawing increasing number of samples or, equivalently, obtaining a large sample of plaintext values from B. However, note that the one factor that A cannot control (irrespective of the estimator he uses) is the true variance of the bucket values, $Var(X_B)$. Therefore, even in the worst case scenario (i.e., $E(X'_B) = E(X_B)$ and $Var(X'_B) = 0$), A still cannot reduce the *ASEE* below $Var(X_B)$, which, therefore, forms the lowest bound of the accuracy achievable by A. Hence, the data owners try to bucketize data in order to maximize the variance of the distribution of values within each bucket. These two cases corresponds to the two extreme cases of nearly equi-depth histogram (when h = 1 and h = G) as analyzed below.

When h = 1 (Det_Enc), since each bucket contains only the same plaintext values, and with the assumption above about additional knowledge of adversary, he can easily infer that the expected value of X'_B equals to that of X_B: $E(X'_B) = E(X_B)$. For the variance, with h = 1, the variance of X'_B gets the minimum value $Var(X'_B) = 0$ (because variance measures how far a set of numbers is spread out, a variance of zero indicates that all the values are identical). In this case, the value of *ASEE* equals to the lowest bound $Var(X_B)$.

When h = *G*, since all plaintext values collide on the same hash value, the difference between $E(X'^2_B) - (E(X'_B))^2$ is big, leading to the big value of $Var(X'_B)$. So, the value of *ASEE* approaches highest bound.

As you can see, although the *coefficient exposure* and *average squared error of estimation* are different ways to measure privacy of equi-depth histogram depending on the adversary's knowledge, they give the same result.

6. PERFORMANCE EVALUATION

This section evaluates the performance of our solutions. We first test with the development board to see the detail time breakdown on the secure hardware (i.e., transfer, I/O, crypto, and CPU cost). Then we use the Z-token described below, which has the same hardware characteristic with this development board to test on the larger scale (i.e., running multiple Z-tokens in parallel) in the real cluster. We also compare the running time on ciphertext and that on cleartext to see how much overhead incurred. We finally increase the power of the cluster by scaling depth (i.e., increase the number of Z-tokens plugged in each node) and scaling width (i.e., increase the number of nodes) to see the difference between the two ways of scaling.

6.1 Experimental Setup

Our experiment is conducted on a cluster of Paris Nord University with 4 nodes. Each node is equipped with 4-core 3.1 GHz Intel Xeon E31220 processor, 8GB of RAM, and 128GB of hard disk. These nodes run on Debian Wheezy 7 with unmodified Hadoop 1.0.3. It is the Cloud provider who decides number of map/reduce tasks. The number of TDSs is also fixed by Cloud provider who plugs these tokens. The experiments will give hints how to choose the number of tokens and nodes.

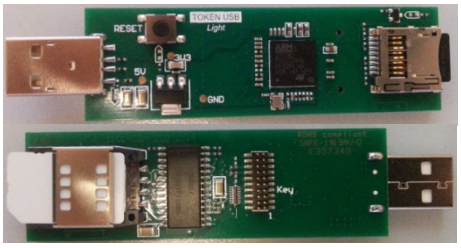


Fig. 7. ZED token (front and back sides)

We first do the unit test on a development board, and then run the Hadoop in parallel on ZED secure tokens (Fig. 7).

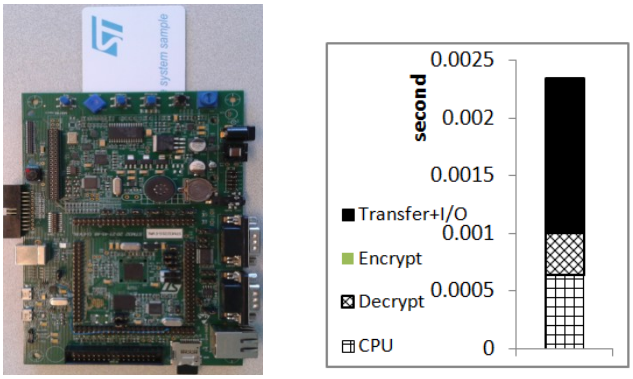
6.2 Unit Test on Development Board

To see the detail time contributing to the total execution time on the secure hardware, we performed unit tests on the development board presented in Fig. 8. This board exhibits hardware characteristics representative of foreseen future secure tokens, including those provided by Gemalto (the smartcard world leader), our industrial partner. This board has the following characteristics: the microcontroller is equipped with a 32 bit RISC CPU clocked at 120 MHz, a crypto-coprocessor implementing AES and SHA in hardware (encrypting or decrypting a block of 128bits costs 167 cycles), 64 KB of static RAM, 1 MB of NOR-Flash and is connected to a 1 GB external NAND-Flash and to a smartcard chip hosting the cryptographic material. The device can communicate with the external world through USB connection.

We measured on this device the performance of the main operations influencing the global cost, that is: encryption, decryption, communication and CPU time. Fig. 8b depicts this internal time consumption of this platform. The transfer cost dominates the other costs due to the connection latencies. The CPU cost is higher than cryptographic cost because (1) the cryptographic operations are done in hardware by the crypto-coprocessor and (2) TDS spends CPU time to convert the array of raw bytes (resulting from the decryption) to the number format for calculation later and some extra operations. Encryption time is much smaller than decryption time because only the result of the aggregation of each partition needs to be encrypted.

TDSs handle data from mappers/reducers in stream due to the fact that encryption and CPU time is less than transfer time and I/O

operations. So, TDSs can process the old data while receiving the new one at the same time.



a) b)

Fig. 8. Unit test on real hardware

6.3 Scaling with Parallel Computing

Figure 10 shows the performance overhead when processing ciphertext over cleartext. There is no difference in map time but the reduce time in ciphertext is much longer than that of cleartext. This is due to the time to connect to Z-token and process the encrypted data inside the Z-token. In this test, only one Z-token is plugged to each node. That creates the bottleneck for the ciphertext processing because Z-token is much less powerful than the node that has to wait Z-token to process the encrypted data. While the cleartext data is processed directly in the powerful node, the ciphertext has to be transferred to tokens for processing. In this way, computation on ciphertext incurs three overhead in compared with the cleartext: i) time to transfer the data from node to token (including the connection time and I/O cost), ii) time to decrypt the data and encrypt the result, iii) the constraint on the CPU and memory size of token for computation inside the token.

Fig. 9. Twenty tokens running in parallel

To alleviate this overhead, we plug multiple tokens to the same node and process the ciphertext in parallel in these tokens. Figure 9 shows the 20 tokens run in parallel and plugged to the same node. In Figure 11, when the number of tokens plugged to each node increases, the reduce time decreases gradually and approaches that of cleartext. Specifically, when the number of tokens increases from 1 to 20, the average speedup is 1.75. So, if we plug 32 tokens to each node, the reduce time will be 5.49 (seconds), which gives approximate 10% longer than cleartext.

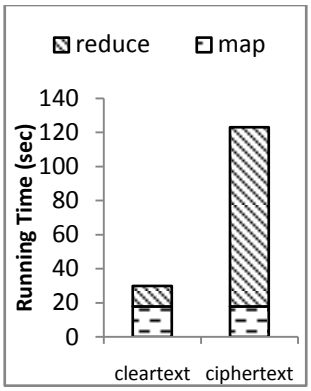


Fig. 10. Running time of cleartext & ciphertext

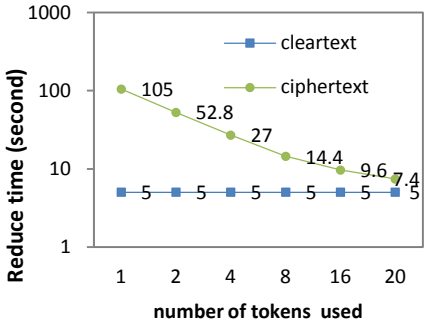


Fig. 11. Scaling depth

Apparently, the overhead is controllable by increasing the number of tokens plugged per reducer.

6.4 Scaling Depth versus Scaling Width

In traditional MapReduce, the cluster can be scaled depth by increasing number of processors per node or scaled width by increasing number of nodes. In our TrustedMR, since it depends on the tokens for cryptographic operations, we scale depth our cluster by increasing the number tokens (i.e., from 1 to 4) plugged to each node. We also scale width by increasing number of nodes (i.e., from 1 to 4), and then we compare the two ways of scaling. In this test, we also increase the size of the dataset (i.e., from 2 million tuples to 4 million tuples) to see how the running time varies.

In Figure 12 & 13, when we increase the number of nodes in the cluster and keep the same number of tokens on each node, the reduce time decreases accordingly and vice versa. Also, with the same number of tokens, plugging them to the same node or to multiple nodes gives almost no difference in term of running time (e.g., the reduce time of 4 nodes with each node having only 1 token is only few percent difference from that of 1 node having 4 tokens plugged). Furthermore, the average speedup of scaling width is 1.74 which is only 2% different from that of scaling depth (i.e., 1.71). In conclusion, scaling depth yields nearly the same performance as scaling width. The only factor that affects the overall performance of the cluster is the total number of tokens plugged to this cluster, no matter how they are distributed to each node.

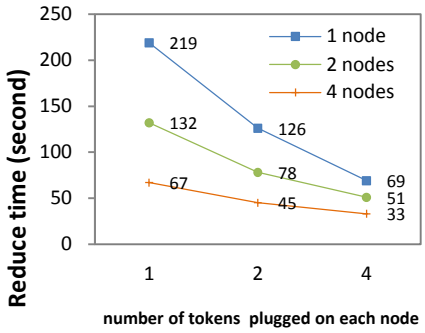


Fig. 12. Reduce time for 2 million tuples

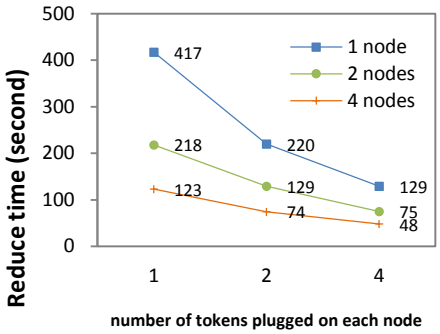


Fig. 13. Reduce time for 4 million tuples

7. CONCLUSION

In this paper, we have proposed a new approach to deal with the problem of processing big data using MapReduce while maintaining privacy guarantees. Our approach draws its novelty from the fact that (private) user data remains under the control of its owner, in a so-called Trusted Data Server. As secure hardware become available at any client device, such highly secure and decentralized architectures can no longer be ignored. The security is pushed to the edge of the network where data is produced, avoiding inherent weakness of the centralized database (single point of attack, low cost/benefit ratio). The existing MapReduce framework keeps unchanged and the types of supported operations are general. We study their efficiency in terms of running time using real secure hardware. The results show the performance overhead is acceptable (i.e., can be controlled to few percents when number of tokens plugged to each node is big enough).

The current limitation to our approach is that it only manages single dataset. In future work, we plan on tackling the problem of *joins* between several datasets, to support social network type queries (e.g. how many users have at least 10 friends that like "literature"). We also extend our thread model that we consider the breakable TDSs which can be compromised by adversaries. The other research direction is to compare this architecture with the use of IBM secure coprocessors at the server side (e.g., IBM 4765 PCIe Cryptographic Coprocessor).

8. ACKNOWLEDGMENTS

The authors wish to thank Nicolas Greneche from University of Paris Nord for his help in setting up the cluster for the experiment. This work is partially funded by project ANR-11-INSE-0005 "Keeping your Information Safe and Secure".

9. REFERENCES

[1] Allard, T., Anciaux, N., Bouganim, L., Guo, Y., Le Folgoc, L., Nguyen, B., Pucheral, P., Ray, Ij., Ray, Ik., Yin, S.: Secure Personal Data Servers: a Vision Paper. VLDB, pp. 25-35. Singapore (2010)

[2] Allard, T., Nguyen, B., Pucheral, P.: MET$_A$P: Revisiting Privacy-Preserving Data Publishing using Secure Devices. DAPD, 2013.

[3] Anciaux, N., Bonnet, P., Bouganim, L., Nguyen, B., Sandu Popa, I., Pucheral, P.: Trusted Cells: A Sea Change for Personal Data Services. CIDR, USA , 2013.

[4] Arasu, A., Kaushik, R.: Oblivious Query Processing. ICDT 2014

[5] Bajaj, S., Sion, R.: TrustedDB: a trusted hardware based database with privacy and data confidentiality. SIGMOD Conference 2011: 205-216

[6] Blass, E., Noubir, G., Huu, T.V.: EPiC: Efficient Privacy-Preserving Counting for MapReduce. In IACR Cryptology ePrint Archive (2012) 452.

[7] Blass, E. O., Pietro, R. D., Molva, R., Önen, M.: PRISM-Privacy-Preserving Search in MapReduce. In PETS, pp 180-200, 2012.

[8] Boldyreva, A., Chenette, N., Lee, Y., O'Neill, A.: Order-Preserving Symmetric Encryption. EUROCRYPT, pp 224-241, (2009).

[9] Ceselli, A., Damiani, E., De Capitani di Vimercati, S., Jajodia, S., Paraboschi, S., Samarati, P.: Modeling and assessing inference exposure in encrypted databases. ACM TISSEC, vol 8(1), pp. 119-152, (2005)

[10] Damiani, E., De Capitani di Vimercati, S., Jajodia, S., Paraboschi, S., Samarati, P.: Balancing confidentiality and efficiency in untrusted relational DBMSs. ACM CCS, pp. 93-102, (2003)

[11] Dean, J., and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Commun. ACM, 51(1):107–113, 2008.

[12] Gentry, C.: Fully homomorphic encryption using ideal lattices. STOC, pp. 169-178. Maryland, (2009)

[13] Herodotou, H., Babu, S.: Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. PVLDB 4(11): 1111-1122 (2011)

[14] Hore, B., Mehrotra, S., Tsudik, G.: A Privacy-Preserving Index for Range Queries. VLDB, pp. 223-235, (2004)

[15] Lam, H.Y., Fung, G.S.K., and Lee, W.K.: A Novel Method to Construct Taxonomy Electrical Appliances Based on Load Signatures. IEEE Transactions on Consumer Electronics. 53(2), 653-660.2007.

[16] Mayberry, T., Blass, E., Chan, A.H.: PIRMAP: Efficient Private Information Retrieval for MapReduce. IACR Cryptology ePrint Archive, 2012:398, 2012.

[17] de Montjoye, Y-A., Wang, S. S., Pentland, A.: On the Trusted Use of Large-Scale Personal Data. IEEE Data Eng. Bull. 35(4): 5-8 (2012)

[18] Popa, R. A., Redfield, C. M. S., Zeldovich, N., et al. CryptDB: protecting confidentiality with encrypted query processing. In SOSP, pp 85–100, 2011.

[19] Roy, I., Setty, S., Kilzer, A., Shmatikov, V., and Witchel, E.: Airavat: Security and privacy for MapReduce. USENIX NSDI, pp. 297–312, 2010.

[20] Ruan, A., and Martin, A.: TMR: Towards a trusted mapreduce infrastructure. IEEE World Congress on Services, pages 141–148, 2012.

[21] To, Q.C., Nguyen, B., Pucheral, P.: Privacy-Preserving Query Execution using a Decentralized Architecture and Tamper Resistant Hardware, EDBT, pp. 487-498, 2014.

[22] Tu, S., Kaashoek, M. F., Madden, S., Zeldovich, N.: Processing analytical queries over encrypted data. In PVLDB, pp 289-300, 2013.

[23] Wei, W., Du, J., Yu, T., and Gu, X.: SecureMR: A Service Integrity Assurance Framework for MapReduce. ACSAC, pp. 73–82, 2009.

[24] Zhang, K., Zhou, X., Chen, Y., Wang, X., Ruan, Y.: Sedic: privacy-aware data intensive computing on hybrid clouds. CCS 2011: 515-526

[25] Zhang, C., Chang, E., Yap, R.: Tagged-MapReduce: A General Framework for Secure Computing with Mixed-Sensitivity Data on Hybrid Clouds. CCGrid, pp 31-40, 2014.

[26] Zhang, X., Yang, L.T., Liu, C., Chen, J.: A Scalable Two-Phase Top-Down Specialization Approach for Data Anonymization Using MapReduce on Cloud. Parallel and Distributed Systems, IEEE Transactions on , vol.25, no.2, pp.363-373, 2014.

[27] Zhang, X., Liu, C., Nepal, S., Pandey, S., and Chen, J.: A Privacy Leakage Upper-bound Constraint based Approach for Cost-effective Privacy Preserving of Intermediate Datasets in Cloud, IEEE Transactions on Parallel and Distributed Systems, 24(6): 1192-1202, 2013

[28] Directive 95/46/EC of the European Parliament and of the Council of 24 October 1995 on the protection of individuals with regard to the processing of personal data. Official Journal of the EC, 23, 1995.

[29] The World Economic Forum. Rethinking Personal Data: Strengthening Trust. May 2012.