PRISM —
Privacy-Preserving Searches in MapReduce

Erik-Oliver Blass[1] Roberto Di Pietro[2] Refik Molva[1] Melek Önen[1]
[1]EURECOM, Sophia Antipolis, France
[2]Università di Roma Tre, Roma, Italy

ABSTRACT

This paper presents PRISM, a scheme for keyword search in cloud computing that is privacy-preserving against a curious cloud provider. The main challenge in the particular context of cloud computing is to come up with a scheme that achieves privacy while preserving the efficiency of cloud computing. Main approaches like simple encryption, private information retrieval (PIR) or encrypted keyword search fall short of meeting these requirements. PRISM assures privacy against a curious cloud provider by leveraging an innovative combination of a sound PIR technique with the MapReduce paradigm akin to cloud computing. The keyword search problem in a large database is transformed into a set of parallel instances of PIR on small datasets. Each instance of PIR on a small dataset is efficiently solved by a node in the cloud during the "Map" phase of MapReduce. Outcomes of map computations are then aggregated during the "Reduce" phase and yield the final output of the keyword search operation. Besides formalization and thorough analysis, PRISM has been implemented on Hadoop MapReduce, and its efficiency has been evaluated using DNS logs. PRISM's overhead over baseline search operations was 11% on the average. To the best of our knowledge, PRISM is the first privacy-preserving search scheme for cloud computing that introduces sustainable overhead.

1. INTRODUCTION

Outsourcing services to clouds has become a major trend in today's IT landscape. Instead of setting up and maintaining their own data centers, cloud users take advantage of *public* clouds, operated by large companies like Google or Amazon. Such public clouds are especially appealing, as they help their customers lower the cost of ownership while taking advantage of increased scalability, performance, and flexibility. Having started as the most economically viable alternative for small and medium-size entreprises, cloud computing solutions are also being adopted by governmental organizations [11].

Main services offered by clouds are data storage and data processing through user-defined or publicly available programs. Both storage and processing services provided by the cloud are characterized by high performance and quality of service in that most cloud computing services offer reliable storage and management for very large amounts of data as well as highly scalable and parallel data processing facilities. Cloud computing services often rely on specific systems such as Hadoop MapReduce [2], an open source implementation of Google's MapReduce system [19]. Hadoop MapReduce is widely used and public MapReduce clouds are offered to users by companies such as Amazon [1, 28].

The advantages of cloud computing unfortunately come with a high cost in terms of new security and privacy exposures. Apart from classical security challenges of shared services raised by third party intruders or malicious users, such as access control, clustering, service abuse, key-management, and denial-of-service, outsourcing of data storage and processing raises new challenges in the face of potentially malicious cloud providers. *Privacy* of outsourced data appears to be a major requirement in this context. Some regulations are already provisioned as to the privacy protection of outsourced governmental documents [12, 13, 20], e.g., in order to assure privacy against "curious" clouds, clouds with data centers located in "rogue" countries or with insufficient security guarantees, and to avoid data leakage in case of operational failures in the cloud. Along these lines, there is also raising corporate concern about the privacy of sensitive business data stored in the cloud [16]. Although cloud providers thrive to meet the increased privacy demand by certifying their services, cf., Google Apps for Government [34], malicious insiders have still been identified as one of the top threats in cloud computing [17], and users want additional privacy guarantees provided by independent parties.

While encryption of outsourced data by the users seems to be a viable protection mechanism against most privacy problems, classical data encryption mechanisms do not suit the requirements of cloud computing: Unlike passive storage devices, the cloud does not only serve as high capacity memory, but also is involved in data processing such as statistical data analysis, log analysis, indexing, data mining, and searching for expressions [28]. Although these computations are simple, they have to be carried out in the cloud due to the large amounts of data. However, data processing performed by the cloud would not be feasible with encrypted data. Straightforward download of encrypted data, its decryption and processing in the user environment would also be totally counterproductive by eliminating all performance advantages of cloud computing as highlighted by Chen and Sion [10]. Data has to be processed within the cloud to capitalize on the high performance offered by the cloud.

More advanced data confidentiality and privacy approaches such as Private Information Retrieval (PIR) [37] or keyword search (KS) on encrypted data [5, 35] also fall short of meeting the privacy requirements of cloud computing. Like simple encryption, PIR only offers basic data storage and lookup operations and does not allow the cloud provider to perform any operation on the stored data. KS that aims at carrying out some operation on encrypted data is designed based on a centralized execution model that is not compatible with highly parallel cloud computing architectures.

Among data processing primitives, keyword search, i.e. verifying, whether a certain keyword is part of a dataset, is not only one of the most fundamental operations, but surprisingly also one of the most demanded applications in cloud computing [28]. In this paper, we present PRISM, a new scheme for privacy-preserving and efficient keyword search in cloud storage. PRISM pursues two specific objectives: 1.) privacy against potentially malicious cloud providers and 2.) high efficiency through the integration of security mechanisms with cloud operations.

PRISM assures privacy against the curious cloud provider by relying on well-known PIR techniques whereby the privacy of both the stored data and the lookup requests are assured against the storage provider. Efficiency is assured through the MapReduce paradigm that allows for parallel execution of a keyword search on very large amounts of data.

The main idea of PRISM is thus the integration of the PIR scheme into MapReduce in order to achieve high performance while capitalizing on the privacy protection of PIR. PRISM thus transforms the problem of verifying the existence of a keyword in the entire database into a set of numerous parallel instances of PIR on small datasets. Each instance of PIR on a small dataset is efficiently solved by a node in the cloud during the "Map" phase of MapReduce. Outcomes of map computations are then aggregated during the "Reduce" phase. Thanks to the linearity of the PIR technique that is chosen, the simple aggregation of the map results during the "Reduce" phase yields the final output of the keyword search operation.

PRISM has been implemented on a cloud computing prototype using a standard MapReduce system. Efficiency of PRISM has been evaluated on that implementation through search operations in Domain Name System (DNS) logs provided by an Internet Service Provider. The overhead of PRISM over baseline search operations (without the privacy support) was found to be 11% on the average.

The main contributions of PRISM are as follows:

- **Suited to cloud computing:** PRISM is the first privacy-preserving search scheme that is suited for cloud computing, that is, it provides storage and lookup privacy with very high performance by leveraging the efficiency of the MapReduce paradigm with the privacy guarantees of a PIR scheme.

- **Privacy in the face of potentially malicious cloud provider:** PRISM allows the user to carry out some critical operation in the cloud without having to trust the cloud provider.

- **Compatible with standard MapReduce:** PRISM only requires a standard MapReduce interface without modifications in the underlying system. PRISM can thus be integrated on any cloud that provides a standard MapReduce interface such as, e.g., Amazon.

- **Flexible search:** In contrast to classical encrypted keyword search techniques, PRISM is not limited to searching for a fixed set of predetermined keywords to be known in advance, but offers flexible searching for any expression.

2. PROBLEM STATEMENT AND ADVERSARY MODEL

Throughout this paper, we will use a toy application example to motivate our work. Motivated by recent events [33], we envision a *data retention* scenario. As of today, telecommunication service providers such as Internet service providers or telephone providers, must log and retain (for some time) specific details about clients accessing services. This data retention is required due to regulatory matters in many countries to enable, for example, law enforcement or to combat terrorism. As the sheer amount of data to retain will raise, we expect service providers to outsource their logfiles to clouds. Still, e.g., law enforcement authorities will contact providers and want them to search for *expressions* (words, strings, . . .) in outsourced files.

Assume service provider $\mathcal{U}$ (the cloud "user") providing DNS services to clients. $\mathcal{U}$ logs each client's access, i.e., $\mathcal{U}$ logs the tuple (timestamp, client ID, hostname queried). Due to the large amount of log data and cost reasons, $\mathcal{U}$ outsources its logfiles into a cloud. Regularly, say each day i, $\mathcal{U}$ creates a new logfile L_i. At the end of a longer period, $\mathcal{U}$ wants (or it is forced to) to find out, whether a client was interested in a suspicious host w. So, $\mathcal{U}$ checks, at which day, in which logfiles L_i, expression w occurs. $\mathcal{U}$ queries the cloud for w, and the cloud responses with an answer R that tells $\mathcal{U}$ which of the L_i contains w. Note that $\mathcal{U}$ does not know in advance which expression w it has to search for.

However, the cloud is assumed to be untrusted, more precisely semi-honest ("honest-but-curious"). Regulatory matters imply that the cloud must not learn any information about the content it hosts and search queries performed. We will now formalize privacy requirements for our application.

2.1 Privacy Requirements

Our application demands for two main types of privacy. *1.) Storage privacy.* The cloud must not be able to infer any details about stored data. *2.) Query privacy.* The cloud must not learn any details about $\mathcal{U}$'s queries and results delivered back to the cloud.

2.1.1 Storage privacy

The cloud (now called "adversary $\mathcal{A}$") must not be able to learn the content or to compute statistics on the content. Example: if stored logfiles contain the same words, then this must be undetected. This automatically calls for encryption of content before uploading. One (sufficient) approach to achieve this property is to use an asymmetric probabilistic encryption cipher to encrypt and upload data. For example, using traditional Elgamal provides semantic security and ciphertext indistinguishability [26]. However, we target a slightly *weaker* indistinguishability property compared to Goldwasser and Micali [26]: the main problem we want to address with storage privacy is that adversary $\mathcal{A}$ must *not* understand whether any two ciphertexts are originating from the same plaintext. Inspired by the traditional notions of indistinguishability, we can cover storage privacy by our (informal) definition of "IND-CO" for uploaded ciphertexts.

DEFINITION 1 (CIPHERTEXT-ONLY INDISTINGUISHABLE). *An encryption cipher E is ciphertext-only indistinguishable (IND-CO)* **iff** *an adversary $\mathcal{A}$ with access to* only *a set of ciphertexts $E(w_1), \ldots, E(w_n)$ cannot decide whether $\exists i, j$ such that $w_i = w_j$.*

The difference to traditional indistinguishability is that $\mathcal{A}$ is unable to submit his own plaintexts to an encryption oracle, but only has access to ciphertexts. IND-CO automatically implies *confidentiality*, i.e., $\mathcal{A}$ does not learn any cleartext w_i: if $\mathcal{A}$ would learn cleartexts (w_i, w_j) then he would trivially be able to distinguish them.

In conclusion, storage privacy requires encryption of content before uploading to the cloud using an IND-CO encryption mechanism.

2.1.2 *Query privacy*

Interaction between $\mathcal{U}$ and the cloud consists of two steps. First, $\mathcal{U}$ sends $Q(w)$, a search query for expression w to the cloud. The cloud processes this query using a protocol $\mathcal{P}$ to produce output $R := \mathcal{P}(Q(x))$. This output is sent back to $\mathcal{U}$. Using this output R and another algorithm $\mathcal{D}$, $\mathcal{U}$ can compute the list of files $\mathcal{D}(R) = \{L_1, L_2, \ldots\}$ containing w.

In our scenario, an adversary $\mathcal{A}$ must not learn which expression w user $\mathcal{U}$ is looking for. Also, $\mathcal{A}$ must not learn anything about subsequent queries, i.e., search *patterns* should be protected: for example, $\mathcal{A}$ is oblivious of whether the same word is queried for twice. Moreover, $\mathcal{A}$ must not learn which file L_i contains w. That is, $\mathcal{A}$ must not learn for which file the search was "successful" or whether two subsequent queries have produced the same list of files. We capture the above requirements in our definition of query privacy.

Definition 2 (Query Privacy). *A protocol $\mathcal{P}$ to find expressions on encrypted data provides* query privacy, **iff** *adversary $\mathcal{A}$ with access to ciphertexts $E(w_i)$, a set of queries $Q(w_j)$, knowledge of protocol $\mathcal{P}$ and outputs R_j (but not $\mathcal{D}$) cannot decide*

1. *whether $\exists Q(w_u), Q(w_v)$ such that $w_u = w_v$.*
2. *for any pair of outputs (R_1, R_2), whether $\exists L_1 \in \mathcal{D}(R_1)$, $L_2 \in \mathcal{D}(R_2)$ such that $L_1 = L_2$.*

The first condition ensures that by looking at the queries, $\mathcal{A}$ cannot *link* any two of them. This requirement automatically implies query *confidentiality*, i.e., $\mathcal{A}$ does not learn anything about the expression w_i a query $Q(w_i)$ is about: if $\mathcal{A}$ would know how to compute w_i from $Q(w_i)$, then he would be able to link two queries $Q(w_1), Q(w_2)$.

The second condition ensures that $\mathcal{A}$ cannot *link* any two outputs, i.e., decide whether w was found in the same file on subsequent queries or not. This implies output *confidentiality*. $\mathcal{A}$ does not learn anything about the output $\{L_1, L_2, \ldots\}$ of the processing of $\mathcal{P}(Q(w_i))$: if $\mathcal{A}$ would be able to compute the L_i, then he would be able to link two outputs.

In conclusion, the definition of query privacy implies a randomized encryption of queries and a protocol $\mathcal{P}$ that can operate under $\mathcal{A}$'s control on such queries producing "randomized" output.

Finally, note that coping with a completely dishonest cloud that could deviate from properly executing $\mathcal{P}$—that is, maliciously forging output or just sending back garbage to $\mathcal{U}$—is out of the scope of this paper. While this is a clearly important real-world issue, PRISM only focuses on a curious semi-honest adversary.

2.2 MapReduce

In the following, we provide an overview of MapReduce, only focusing on aspects necessary to understand PRISM. For details, the reader may refer to [2].

Upload. Roughly speaking, a MapReduce cloud comprises a set of "slave" node computers and a "master" computer. While user $\mathcal{U}$ uploads files into the MapReduce cloud, each file is automatically split into blocks, so called *InputSplits*. InputSplits have a fixed size which is a pre-configured system parameter. For each InputSplit, a workload sharing algorithm running on the master selects a slave node and places the InputSplit on it. Also, for fault tolerance, each InputSplit is replicated a number of times and placed on additional slaves.

In addition to data, the MapReduce framework also allows $\mathcal{U}$ to upload "operations", i.e., compiled Java classes. These classes essentially represent the implementation of three functions.

1. $Scan(\textsc{InputSplit}) \rightarrow [(k, v)])$, a functions that takes an InputSplit as an input, parses it, i.e., scans it and generates a set of key-value pairs $[(k, v)]$ out of it.
2. $Map(k, v) \rightarrow [(k', v')]$, a function that takes as an input a single key-value pair (k, v) and outputs a set of "intermediate" key-value pairs $[(k', v')]$.
3. $Reduce([k', v']) \rightarrow \textsc{File}$, a function that takes as an input a set of intermediate key-value pairs $[(k', v')]$ and writes arbitrary output into a file.

Uploaded Java classes are replicated and sent to all slave nodes storing an InputSplit.

Map Phase. After data and implementations have been uploaded, $\mathcal{U}$ can specify a single uploaded file (now split into InputSplits) and trigger MapReduce operations on that file. As the name suggests, the first phase of operation is the "Map" phase. Each slave node becomes a "mapper" node. First, each mapper executes $\mathcal{U}$'s *Scan* function on the InputSplit it stores. This generates a set of key-value pairs on each mapper. Furthermore, the mapper node executes $\mathcal{U}$'s Map function on this generated key-value pairs to produce a set of intermediate key-value pairs.

Reduce Phase. Eventually, MapReduce starts the "Reduce" phase. Slave nodes in the MapReduce cloud are scheduled to become "reducers". For each of the generated intermediate pairs (k', v'), MapReduce selects a reducer and sends (k', v') to this reducer. More precisely, MapReduce selects the *same* reducer for all intermediate key-value pairs (k', v') having the same key. So, each mapper receives a set of intermediate key-value pairs (k'_1, v'_1), $(k'_2, v'_2), \ldots$ with equal keys $k'_1 = k'_2 = \ldots$. Each reducer executes $\mathcal{U}$'s reduce function on its set of intermediate key-value pairs and writes the output to a file. This file is sent to $\mathcal{U}$.

After completion of such a MapReduce job on a single uploaded file, $\mathcal{U}$ could invoke MapReduce on another uploaded file. However, for increased efficiency, instead of sequentially processing files, $\mathcal{U}$ can also specify a *set* of uploaded files. MapReduce will process the set of files completely in parallel. Still, Scan/Map/Reduce operations on InputSplits belonging to different files are separated to keep the same semantics as files were processed sequentially.

3. PRISM PROTOCOL

Before going into PRISM's technical details, we will first highlight its main design rationales. PRISM comprises two parts: a *upload of data into the cloud phase* and then the actual MapReduce *search*.

1.) Upload. During upload, user $\mathcal{U}$, encrypts expressions using symmetric encryption. Ciphertexts are stored in a file, and this file is sent to the MapReduce cloud. In PRISM, we use standard encryption, e.g., a blockcipher like AES, to perform ciphering of data. However, to ensure IND-CO, plaintext is modified before encryption. Therewith, $\mathcal{U}$ can still search for some expression w, but the cloud cannot compute statistics about ciphertexts.

2.) Search. Eventually, $\mathcal{U}$ wants to search his encrypted files for some expression w. Therefore, $\mathcal{U}$ sends implementations of "algorithms" for the map and reduce phases to the MapReduce cloud, and the cloud executes these on uploaded data. For example, $\mathcal{U}$ sends Java ".class" files for the mappers and Java ".class" files for the reducers. MapReduce distributes these implementations to each mapper and reducer, respectively. PRISM's main idea is that each mapper, scanning through its InputSplit, creates a binary matrix. Ciphertexts in the InputSplit are assigned to individual elements in that matrix. If a ciphertext is present in an InputSplit, its corresponding element in the matrix is set to either "0" or "1". Using private information retrieval techniques, PRISM can extract the value of a single element in the matrix with the mapper being totally oblivious to which element is extracted. Consequently, $\mathcal{U}$ can specify which element to extract in a privacy-preserving way. All mappers send their obliviously extracted elements as key-value pairs to reducers. Reducers simply sum up received values and return sums to $\mathcal{U}$. Therewith, neither mappers nor receivers can learn any information about which ciphertext $\mathcal{U}$ was interested in.

The final step of the search part is called **result analysis**: $\mathcal{U}$ receives an encrypted sum for each of the originally uploaded files from reducers. $\mathcal{U}$ can decrypt them and decide which of the files contain w.

However, due to the probability of "collisions" in matrices, i.e., two different ciphertexts can be assigned to the same element, and due to ambiguities of received sums, $\mathcal{U}$'s decision whether w is inside some file might be wrong. Therefore, PRISM repeats the above process in a total of q so called "rounds". In each of the rounds, a new matrix is generated, elements are set to "1" or "0" depending on the round number, and results are returned as described. This reduces the probability of $\mathcal{U}$ making incorrect decisions.

3.1 Preliminaries

We begin with an overview about variables used throughout this paper:

w – expression user $\mathcal{U}$ is looking for.

S_{File} – total size of file to be searched.

$S_{\text{InputSplit}}$ – size of one "InputSplit". Each file is split into several "InputSplits", InputSplits are distributed to Map nodes in the cloud, one node works on one InputSplit. The size of an InputSplit in MapReduce is usually between 64 MByte and 128 MByte.

c – number of InputSplits of one file, $c = \frac{S_{\text{File}}}{S_{\text{InputSplit}}}$. If the cloud provides at least c slave nodes (more precisely, at least c CPUs), then all InputSplits can be processed in parallel. Otherwise, mappers have to work on multiple InputSplits consecutively.

n – number of ciphertexts in one InputSplit,
$n := \frac{S_{\text{InputSplit}}}{\text{CipherBlockSize}}$. The total number of ciphertexts stored in the cloud is $(c \cdot n)$.

3.1.1 Trapdoor Group Private Information Retrieval

PRISM uses a simple and efficient PIR mechanism as previously suggested by Trostle and Parrish [41]. As this PIR mechanism is just a (exchangeable) building block for PRISM, we will only give a conclusive summary of its mode of operation and rationale.

Overview: Matrix $\mathcal{M}$ is a $t \times t$ matrix of elements in $\mathbb{Z}_N$ stored at a server. For example, $N = 2$ for a binary matrix. User $\mathcal{U}$ is specifically interested in the elements of the k^{th} row in $\mathcal{M}$, but the server must not determine $\mathcal{U}$'s interest.

The idea is now that $\mathcal{U}$ sends two "types" of values to the server. For each row that $\mathcal{U}$ is not interested in, he sends a value of the "first" type. For the one row that $\mathcal{U}$ is interested in, he sends a single value of the "second" type. To prevent the server from distinguishing between the two types of values, $\mathcal{U}$ blinds each value with a blinding factor b. This blinding factor can later be removed by $\mathcal{U}$. The server now performs simple additions with received values and elements stored in $\mathcal{M}$. The result is sent back to $\mathcal{U}$ who removes the blinding factor and determines the values of the row of his interest.

Preparation: Assume $\mathcal{U}$ is interested in row k. $\mathcal{U}$ chooses a group $\mathbb{Z}_p$, where p is a prime of m bits, $gcd(m,p) = 1$. $\mathcal{U}$ also chooses a random $b \in \mathbb{Z}_p$ and t random values $a_i \in \mathbb{Z}_p$. Therewith, $\mathcal{U}$ computes t values $e_i < \frac{p}{t\cdot(N-1)}$ such that $e_k := 1 + a_k \cdot N$ and $\forall i \neq k : e_i := a_i \cdot N$.

Finally, $\mathcal{U}$ computes $\alpha_i := b \cdot e_i \mod p$ and sends the α_i to the server. Other values (p, m, b, e_i, a_i) remain secret. The server treats α_i as large integers and performs the following integer operations, i.e., without any modulo.

Server computation: Let $\vec{u}$ be the vector $\vec{u} := (\alpha_1, \ldots, \alpha_t)$. The server computes the matrix product

$$\vec{v} := (\beta_1, \ldots, \beta_t) = \vec{u} \cdot \mathcal{M} = (\sum_{i:=1}^{t} \alpha_i \cdot \mathcal{M}_{i,1}, \ldots, \sum_{i:=1}^{t} \alpha_i \cdot \mathcal{M}_{i,t}).$$

The server sends $\vec{v}$ back to $\mathcal{U}$.

Result analysis: Upon receipt, $\mathcal{U}$ "un-blinds" values: $\mathcal{U}$ computes the t inverse values $z_i := \beta_i \cdot b^{-1} \mod p$. Now, U can conclude that $z_i \mod N$ equals the i^{th} element of the k^{th} row in $\mathcal{M}$. There with $\mathcal{U}$ has retrieved the t elements of the k^{th} row of $\mathcal{M}$ in a privacy-preserving fashion.

Security rationale: The security and privacy of this protocol is based on the so called "trapdoor group assumption". With only knowledge of α_i, it is computationally hard for the server to infer any information about low order bits, i.e., the modulo of z or e_i, cf., Trostle and Parrish [41].

3.2 Upload

In our toy scenario, the webserver continuously logs access to the cloud. Each day the webserver starts using a new logfile. For simplicity, we assume that entries logged by webservers are simple expressions.

Overview. One privacy requirement as stated earlier in Section 2 is that the cloud must not be able derive any details about stored cleartext. We will cover this by using a *stateful cipher* and show its IND-CO property later in Section 4.1.

DEFINITION 3 (STATEFUL CIPHER). *Given a standard symmetric encryption cipher E with secret key k, e.g., AES, we extend E to a* stateful cipher*, by adding "counters" γ_{w_i} that count the history of different inputs w_i. Each time E encrypts w_i, counter γ_{w_i} is increased by one.*

So, a stateful cipher is a cipher that knows how often it has encrypted as specific plaintext. We will know present a trivial stateful cipher construction and use this in PRISM to encrypt before uploading.

Upload details: User $\mathcal{U}$ and webserver W share a secret key K. For each day d, webserver W computes his key-of-the-day $K_d := \text{HMAC}_K(d)$.

W executes Algorithm 1. For each day, W maintains a hash table containing the list of counters γ_{w_i} in W's local storage. At the beginning of each day, W initializes all counters to 0, i.e., $\gamma_{w_i} = 0$.

Now, for each logentry w_i that should be stored in the cloud, W computes γ_{w_i} and increases γ_{w_i}. Then, W computes ciphertext C_i. Webserver W sends ciphertext C_i to the cloud that stores it in this day's file.

```
Initialize all $\gamma$ to 0;
**foreach** *logentry* $w_i$ **do**
    $\gamma_{w_i}$ := get ($w_i$); //from hash table
    $\gamma_{w_i} := \gamma_{w_i} + 1$;
    insert ($w_i, \gamma_{w_i}$); //into hash table
    $C_i := E_{K_d}(w_i || \gamma_{w_i})$;
    **upload** $C_i$ ;
**end**
```

Algorithm 1: "Stateful Cipher" encryption and upload to MapReduce. Key-of-the-day is K_d.

We discuss the reason for using a "stateful cipher" over using, e.g., a CBC mode of encryption in Section 4.1.

3.3 Search: Map Phase

User $\mathcal{U}$ wants to search a set of files for expression w within a period (a set of days) of time. For ease of understanding, we will restrict our description below to PRISM working on a single file specified by the user, i.e., the file of day d. In practice, the user can specify a set of files, and all files will be separately (but in parallel) processed with PRISM exactly like a single file.

$\mathcal{U}$ sends map and reduce implementations of PRISM to MapReduce, and the map phase starts. In the following, we describe the PRISM algorithms for, first, the mappers and in Section 3.4 the reducers. We would like to stress that the PRISM algorithms, e.g., Java ".class" files, are not encrypted and not specially protected against a curious cloud. Even though mappers and reducers know what operations they perform, they cannot deduce any private information about stored data or details about the search.

Overview. Before scanning through its InputSplit, a mapper node creates a matrix where all elements are initialized to "0". PRISM's main idea is that while the mapper scans the ciphertexts in its InputSplit, each ciphertext is then assigned to one position, a certain element, in a matrix using a hash function. For each ciphertext, the mapper can put a "1" in the matrix at the assigned position. Roughly speaking, $\mathcal{U}$ can now query the mapper for the value of a specific element in the matrix using private information retrieval. This guarantees privacy.

Problem is that due to the limited size of the matrix and the properties of the hash function, there might be collisions in the assignment process. That is, by chance there can be two different ciphertexts being assigned with the same position in the matrix. By chance, the information retrieved by $\mathcal{U}$ can therefore be unrelated to w. To mitigate this effect, PRISM repeats generation and filling of matrices a total of q rounds. Also, setting an element in a matrix to "1" depends on the round number. After q rounds, the probability that the information $\mathcal{U}$ retrieved from this mapper is unrelated to w therefore decreases, and $\mathcal{U}$ can finally decide whether w is inside this particular InputSplit.

Preparation:

DEFINITION 4 (PIR MATRIX). *A binary matrix $\mathcal{M}$ of size $t \times t$ is called a PIR matrix. This is the matrix used by the mapper to implicitly perform the keyword search in a privacy-preserving way.*

DEFINITION 5 (CANDIDATE POSITION). *For each ciphertext C_i in an InputSplit, the candidate position $(\mathcal{X}_i, \mathcal{Y}_i)$ of C_i in $\mathcal{M}$ is computed by $(\mathcal{X}_i || \mathcal{Y}_i) := \lfloor C_i \rfloor_{2 \cdot \log_2(t)}$. That is, the first t bits of C_i determine $\mathcal{X}_i$, and the second t bits of C_i determine $\mathcal{Y}_i$.*

DEFINITION 6 (PIR INPUT). *If $\mathcal{U}$ is interested in a specific element $(\mathcal{X}, \mathcal{Y})$ in $\mathcal{M}$. He computes PIR input $\{\alpha_1, \alpha_2, \ldots, \alpha_t\}$, where $\alpha_{\mathcal{X}} := b \cdot (1 + a_{\mathcal{X}} \cdot N) \mod p$, and $\forall i \neq \mathcal{X}, \alpha_i := b \cdot (a_i \cdot N) \mod p$. Random values b and a_i as well as parameters N and p are chosen as for the Trapdoor Group PIR scheme presented in Section 3.1.1.*

DEFINITION 7 (COLUMN SUM). *The column sum σ_i of the i^{th} column of PIR matrix $\mathcal{M}$ is defined as*

$$\sigma_i := \sum_{\mathcal{M}_{1 \le j \le t, i} = 1} \alpha_j,$$

where $\mathcal{M}_{1 \le j \le t, i} = 1$ denotes the entries in the i^{th} column of $\mathcal{M}$ that are set to 1.

Note that additions in this definition are integer additions.

The above computation of column sums is simply a digest of the PIR technique by Trostle and Parrish [41]. In short, if a mapper computes such a column sum on a given PIR matrix $\mathcal{M}$ and given PIR inputs α_i, it is impossible for the mapper to derive $(\mathcal{X}, \mathcal{Y})$. $\mathcal{U}$, however, can compute whether $\mathcal{M}_{\mathcal{X},\mathcal{Y}} = 1$, because $\mathcal{M}_{\mathcal{X},\mathcal{Y}} = 1$ **iff** $(\sigma_{\mathcal{Y}} \cdot b^{-1} \mod p) \mod 2 = 1$ holds.

It is important to point out that not only a mapper can compute a candidate position for some ciphertext in its InputSplit, but also $\mathcal{U}$ can compute candidate positions. More precisely, as $\mathcal{U}$ is looking for w, he can compute $E(w||1)$ and candidate position $(\mathcal{X}_i || \mathcal{Y}_i) := \lfloor E(w||1) \rfloor_{2 \cdot \log_2(t)}$. If w has been uploaded into a particular InputSplit at least once, then this InputSplit contains at least $E(w||1)$ (maybe also $E(w||2)$, $E(w||1), \ldots$). Therefore, it is sufficient for $\mathcal{U}$ to search for $E(w||1)$. We will now give detailed descriptions of $\mathcal{U}$'s Map and Reduce algorithms.

3.3.1 PRISM Map Description

To start, $\mathcal{U}$ chooses two system parameters $t, q \in \mathbb{N}$ for PRISM. Parameter t is a security parameter, and q denotes the number of rounds.

For day d that $\mathcal{U}$ wants to search for w, he determines the key-of-the-day $K_d := \text{HMAC}_K(d)$ and the target candidate position $(\mathcal{X}_d || \mathcal{Y}_d) := \lfloor E_{K_d}(w||1) \rfloor_{2 \cdot \log_2(2)}$. To prepare PIR, $\mathcal{U}$ computes for day d the t PIR Inputs $\{\alpha_{d,1}, \alpha_{d,2}, \ldots, \alpha_{d,t}\}$ as described above. $\mathcal{U}$ sends all α as part of the following map algorithm implementation to the cloud.

$$\begin{array}{c|cccc}
\mathcal{X}\backslash\mathcal{Y} & 1 & 2 & \cdots & t \\
\hline
1 & 1 & 0 & \cdots & 1 \\
2 & 0 & \textcircled{1} & \cdots & 0 \\
\vdots & \cdots & \cdots & \cdots & \cdots \\
t & 1 & 0 & \cdots & 0
\end{array}$$

Figure 1: Sample matrix $\mathcal{M}_j$. Candidate position (2,2) is for one ciphertext, e.g., $(2,2) = \lfloor C_i \rfloor_{2\cdot\log_2(t)}$**. Value "1" stored in (2,2) is output of $\lfloor h(C_i, j) \rfloor_1$.**

```
for $l := 1$ to $q$ do
    INITIALIZE $\mathcal{M}_l$;
end
SCANTHROUGHINPUTSPLIT;
foreach pair $(i, C_i)$ do //Fill matrices
    $(\mathcal{X}_i||\mathcal{Y}_i) := \lfloor C_i \rfloor_{2\cdot\log_2(t)}$;
    for $j := 1$ to $q$ do
        $bit_j := \lfloor h(C_i, j) \rfloor_1$;
        if $bit_j = 1$ then
            $\mathcal{M}_{j,\mathcal{X}_i,\mathcal{Y}_i} := 1$;
        end
    end
end
for $l := 1$ to $q$ do //q rounds
    for $j := 1$ to $t$ do //Compute column sums
        $\sigma_{j,l} = \sum_{\mathcal{M}_{l,1\leq k\leq t,j}=1} \alpha_{d,k}$
    end
end
for $j := 1$ to $t$ do //Intermediate (k,v) pairs
    $(k, v) :=(\{\text{FILE},\ j\},\{\sigma_{j,1}, \ldots, \sigma_{j,q}\})$;
    OUTPUT $(k, v)$;
end
```

Algorithm 2: Computation of PIR matrices $\mathcal{M}$.

All mappers process PRISM in parallel, each of them on its own InputSplit of the file corresponding to day d. More precisely, a mapper executes Algorithm 2. Initially, the mapper generates q PIR matrices M_l, where each element is initially set to 0. We will now write $\mathcal{M}_{l,\mathcal{X},\mathcal{Y}}$ to denote element $(\mathcal{X}, \mathcal{Y})$ in matrix $\mathcal{M}_l$.

The mapper node *scans* its complete InputSplit consisting of ciphertexts $\{C_1, \ldots, C_n\}$. For each ciphertext C_i, the mapper creates a key-value pair (i, C_i).

Then, the mapper fills matrices $\mathcal{M}_l$. For pair (i, C_i),

- the mapper computes candidate position $(\mathcal{X}_i||\mathcal{Y}_i) := \lfloor C_i \rfloor_{2\cdot\log_2(t)}$.
- the mapper puts in PIR matrix $\mathcal{M}_j$, in element $\mathcal{M}_{j,\mathcal{X}_i,\mathcal{Y}_i}$, a "1", if $\lfloor h(C_i, j) \rfloor_1 = 1$. See Figure 1 for an example. Otherwise, element $\mathcal{M}_{j,\mathcal{X}_i,\mathcal{Y}_i}$ remains untouched. This means that entries in $\mathcal{M}_j$ can flip from 0 to 1, but never from 1 back to 0.

After all q PIR matrices are filled, the mapper computes for each matrix the t column sums $\sigma_{1\leq i\leq t,1\leq j\leq q}$ on $\mathcal{U}$'s input $\{\alpha_{d,1}, \ldots, \alpha_{d,t}\}$. Finally, the mapper outputs intermediate key-values pairs (k, v). The *key* comprises the name of the file of the InputSplit this mapper was working on, e.g., the file name is the day d, and the number of the column sum of $\mathcal{M}_l$. The *value* consists of a list of the q column sums.

These intermediate key-value pairs will now be input for the reducers during the Reduce phase.

3.4 Search: Reduce Phase

Overview: A single reducer receives from all the c mappers working on the same file all their q column sums for the *same* column. The reducer simply adds these received sums and writes the result into a file which is sent back to $\mathcal{U}$.

Details: For all key-value pairs $(\{\text{FILE},\ i\},\{\sigma_{i,1}, \ldots, \sigma_{i,q}\})$ using the same $\{\text{FILE},\ i\}$ as key, the MapReduce framework designates the same reducer. This reducer receives from all c different mappers working on the same file all intermediate key-value pairs with the same key. That is, a reducer receives c pairs which we rewrite as

$$\begin{gathered}
(\{\text{FILE}, i\}, \{\sigma_{i,1,1}, \ldots, \sigma_{i,q,1}\}) \\
(\{\text{FILE}, i\}, \{\sigma_{i,1,2}, \ldots, \sigma_{i,q,2}\}) \\
\ldots \\
(\{\text{FILE}, i\}, \{\sigma_{i,1,c}, \ldots, \sigma_{i,q,c}\}).
\end{gathered}$$

Here, for a given $\sigma_{i,j,k}$, value i, $1 \leq i \leq t$ denotes the column, value j, $1 \leq j \leq q$ denotes the round, and value k, $1 \leq k \leq c$ the InputSplit.

Using integer addition, the reducer computes q "final PIR sums" $s_{\text{FILE},i,j} := \sum_{k=1}^{c} \sigma_{i,j,k}, 1 \leq j \leq q$, and stores values $\{s_{\text{FILE},i,1}, \ldots, s_{\text{FILE},i,q}\}$ into an output file. In conclusion, $s_{\text{FILE},i,j}$ represents the sum of column sums of all the mappers of one particular column i in PIR matrix j.

This output file is downloaded by $\mathcal{U}$.

3.5 Result Analysis

For each outsourced file (day d), user $\mathcal{U}$ retrieves an output file generated by reducers. Now, $\mathcal{U}$ analyzes retrieved files' content to finally conclude which of the outsourced files contain w. Again for ease of understanding, we restrict our description to the analysis of the result generated from PRISM on a single outsourced file called FILE. $\mathcal{U}$ repeats this process with all other results from the other files accordingly.

3.5.1 Result Analysis Protocol

DEFINITION 8 (COLLISION). *Assume $\mathcal{U}$ is looking for w, so $C := E_{K_d}(w, 1)$. Similar to hash functions, a* collision *in PIR matrix $\mathcal{M}$ denotes the case of an event where the candidate position $(\mathcal{X}', \mathcal{Y}')$ of another ciphertext $C' \neq C$ matches the candidate position $(\mathcal{X}||\mathcal{Y}) = \lfloor E(w, 1) \rfloor_{2\cdot\log_2(t)}$ of w in $\mathcal{M}$. That is, $\lfloor C \rfloor_{2\cdot\log_2(t)} = \lfloor C' \rfloor_{2\cdot\log_2(t)}$.*

DEFINITION 9 (ONE-COLLISION). *A* one-collision *is the event where in an InputSplit a ciphertext $C' \neq E_{K_d}(w, 1)$ puts a 1 into the same candidate position in $\mathcal{M}$ as $E_{K_d}(w, 1)$.*

Overview: The rationale for the result analysis protocol of PRISM is to observe the candidate position of C over q rounds to mitigate the effect of one-collisions. Of particular interest will be rounds where $\lfloor h(C, i) \rfloor_1 = 1$.

First, $\mathcal{U}$ un-blinds all values received from reducers. Based on the result, $\mathcal{U}$ distinguished two cases.

Case 1.) If a reducer, reducing for a specific file FILE, has returned the value 0 for C's candidate position, then U knows for sure that all mappers have output 0 for this candidate position. Consequently, the candidate position in

```
forall files FILE do
    for $i := 1$ to $q$ do
        if $\lfloor h(C,i) \rfloor_1 = 1$ then
            $\mathcal{U}$ reads $s_{\text{FILE},\mathcal{Y},i}$;
            $s_i := (s_{\text{FILE},\mathcal{Y},i} \cdot b^{-1} \mod p) \mod N$
            $= \sum_{j:=1}^{c} bit_j$;
            if $s_i = 0$ then
                OUTPUT $w \notin$ FILE; //Contradiction
                break;
            end
        end
    end
    OUTPUT $w \in$ FILE;
end
```

Algorithm 3: $\mathcal{U}$ decides whether $w \in$ FILE

matrix $\mathcal{M}$ of each mapper is 0. Therefore, C has not been in any of the InputSplits of FILE, and $\mathcal{U}$ reasons $w \notin$ FILE. If C would have been in one InputSplit, then at least the mapper working on this InputSplit would have returned a 1 in this round.

DEFINITION 10 (CONTRADICTION). *Let w be the expression $\mathcal{U}$ is looking for, and C its ciphertext. If in some round i, $\lfloor h(C,i) \rfloor_1 = 1$ holds and the reducer for file* FILE *sends $\mathcal{U}$ a value of* 0 *then this is called a* contradiction.

In case of such a contradiction, $\mathcal{U}$ knows for sure that w is not in file FILE.

Case 2.) If, however, this reducer returns a value > 0, then w was in at least one InputSplit *or* a one-collision has occurred in at least one InputSplit. User $\mathcal{U}$ can neither decide $w \notin$ FILE nor $w \in$ FILE with absolute certainty.

U's strategy is to keep the probability for one-collisions low and run multiple rounds q, such that eventually a contradiction occurs ($\Rightarrow \mathcal{U}$ decides $w \notin$ FILE), or, if no contradiction occurs, $\mathcal{U}$ decides $w \in$ FILE with only a small error probability P_{err}.

Details: $\mathcal{U}$ executes Algorithm 3. For each file, $\mathcal{U}$ is only interested in row $\mathcal{Y}$ of matrices $\mathcal{M}$, as only they can refer to candidate position $(\mathcal{X}, \mathcal{Y})$. Therefore, $\mathcal{U}$ keeps only values $\{s_{\text{FILE},\mathcal{Y},1}, \ldots, s_{\text{FILE},\mathcal{Y},q}\}$ and discards the rest. In each round where $\lfloor h(C,i) \rfloor_1 = 1$, un-blinds $s_{\text{FILE},\mathcal{X},i}$ to get value $s_i := \sum_{i=1}^{c} bit_i$. If $s_i = 0$, then we have a contradiction, and $\mathcal{U}$ can infer $w \notin$ FILE. If none of the s_i values has been 0 after all the q rounds, then $\mathcal{U}$ will decide $w \in$ FILE. $\mathcal{U}$ will be wrong with P_{err}.

Note that, although PIR matrices are binary matrices, $\mathcal{U}$ sets $N > c$ to cope with the larger possible values that sums might take due to collisions.

In conclusion, $\mathcal{U}$'s strategy can be summarized by

- output $w \notin$ FILE, **iff** $\exists i, s_i = 0$.
- output $w \in$ FILE, **iff** $\forall i, s_i \neq 0$

We will compute $\mathcal{U}$'s error probability P_{err} for the latter case and dependencies of P_{err} and choices of t and q in Section 4.2.

3.6 Extensions

Saving computation: To save some computation in PRISM, we can modify the hash-based mechanism that determines whether to put a "1" or a "0" in a certain element in $\mathcal{M}$. Recall that the first $2 \cdot log_2(t)$ bit of a ciphertext C are used to determine its position (element) in $\mathcal{M}$. However, instead of computing an expensive hash function $\lfloor h(C_i, j) \rfloor_1$ to get a single bit in round j, we can simply replace the hash and take C's bit on position $(2 \cdot log_2(t) + j)$. Assuming that cipher E has good security properties (each bit of C is "1" with probability $\frac{1}{2}$), this results in the same property as using the hash: eventually two different ciphertexts that collide in $\mathcal{M}$ will differ and lead to a contradiction. We use of this computation reduction in our evaluation in Section 5.

Long expressions: As in related work, e.g., Song et al. [40], we assume for simplicity that expressions are not longer than our symmetric cipher blocksize. This might not be realistic in some scenarios. To cope with longer expressions and varying numbers of ciphertext blocks, PRISM could be extended to use (stateful) CBC encryption and to add a hash of each ciphertext to the file. PRISM would then search within hashes and not ciphertexts.

Fuzzy search: PRISM only allows searching for precise expressions. Using techniques similar to Li et al. [31], i.e., by adding encryptions of variations of expressions, PRISM could also support *fuzzy* searching.

Counting: Finally, PRISM's use of counters γ can be used for another data mining application: counting. As with a binary search algorithm, $\mathcal{U}$ can launch a set of queries to eventually determine the exact number of occurrences of an expression. The number of queries scales logarithmically with the total number of occurrences possible.

4. PRISM ANALYSIS

4.1 Privacy

While we target no formal proofs, we evaluate PRISM's privacy analytically and give short proof sketches.

First, we show that PRISM's upload of ciphertexts into the cloud does not reveal any information about plaintexts as in storage privacy. We do this by explaining why our stateful cipher encryption is IND-CO. Second, we show that a set of PRISM queries does neither reveal any information about which expression $\mathcal{U}$ is looking for, nor whether this expression is present (in encrypted form) in the cloud. We do this by explaining why PRISM is query private

4.1.1 Storage Privacy

LEMMA 1. *Stateful cipher $E_k(w_i || \gamma_{w_i})$ is IND-CO, where γ_{w_i} is the counter for the current number of occurrence of plaintext w_i, and "$||$" is an unambiguous pairing of inputs.*

PROOF (SKETCH). For different expressions $w_i \neq w_j$, as k is unknown, output of E is undistinguishable from random data for adversary $\mathcal{A}$. The stateful cipher construction modifies input plaintexts for the cipher, such that even for twice the same plaintexts w_i input to the cipher becomes different, because γ_i will increase. Therewith E creates different output even for the same w_i, undistinguishable from random data. $\square$

Discussion: The remaining question is how to implement such an unambiguous pairing "$||$". If the size of the *concatenated* input $|w_i| + |\gamma_{w_i}|$ is less than the cipher's blocksize (using standard padding), then concatenation provides an unambiguous pairing.

We use a stateful cipher construction to achieve IND-CO, because we require individual ciphertexts to be "findable in a deterministic fashion. While other encryption modes, e.g., CBC, might also provide IND-CO, they do not allow searching for an individual ciphertext deterministically. So for example in CBC, the position of the plaintext in the file changes the ciphertext. As $\mathcal{U}$ cannot know the position of an expression in advance, this renders classical encryption modes useless.

4.1.2 Query Privacy

We simply inherit the strong privacy guarantees and constructions of the PIR mechanism of Section 3.1.1 to prove the two properties of query privacy.

First, note that in PRISM query $Q(w)$ is split into a number of "sub"-queries, namely one query $Q_d(w)$ per file on date d. These queries are processed in parallel by MapReduce, and $\mathcal{U}$ receives output for them separately. Instead of one response R that can be "un-blinded" into a list of files $\mathcal{D}(R) = \{L_1, L_2, \dots\}$ containing w, PRISM provides $\mathcal{U}$ with a set of values $\{s_{\textsc{File},i,j}\}$, one set per file, that are analyzed by $\mathcal{U}$ separately. Therefore, we again restrict our explanations to the case of just one outsourced file.

Lemma 2 explains PRISM's query unlinkability.

Lemma 2. *Given a total of z PRISM-queries $Q_d(w_i) = \{\alpha_{\{d,j\},i}\}$ for expressions w_i, results $R_{\text{File},i} = \{s_{\{\textsc{File},j,k\},i}\}$, $1 \le i \le z, 1 \le j \le t, 1 \le k \le q$, together with PRISM's map and reduce implementation, an adversary $\mathcal{A}$ cannot decide whether $\exists u, v$ such that $w_u = w_v$.*

Proof (Sketch). For each query $Q_d(w_i)$, values $\{\alpha_{\{d,j\},i}\}$ are the product of (fresh) random variable b_i and (fresh) random values e_j of the PIR preparation phase. Deciding, whether $\exists u, v$ such that $w_u = w_v$ is equivalent to deciding whether in any of the z sets of α values, there is one value $\alpha_{\{d,u\},v}$ from the first set and another value $\alpha_{\{d,u'\},v'}$ from the second set, such that either $e_u = e_{u'}$ or $e_{u'} \ne e_{u'}$.

As it is computationally impossible for $\mathcal{A}$ to deduce any information about the mod 2 value of any e_j value. Moreover, e_j values are *randomly* blinded for each individual query, and, therefore, results $\{s_{\{\textsc{File},j,k\},i}\}$ are randomly blinded, too. $\mathcal{A}$ cannot decide whether $e_u = e_{u'}$. □

Finally, Lemma 3 shows PRISM's response unlinkability.

Lemma 3. *Given a total of z PRISM-queries $Q_d(w_i) = \{\alpha_{\{d,j\},i}\}$ for expressions w_i, results $R_{\text{File},i} = \{s_{\{\textsc{File},j,k\},i}\}$, and PRISM's map and reduce implementation, an adversary $\mathcal{A}$ cannot decide for any pair of outputs $(R_{\text{File},i}, R_{\text{File},i'}) = (\{s_{\{\textsc{File},j,k\},i}\}, \{s_{\{\textsc{File},j,k\},i'}\})$ whether $\exists j_1, j_2, k_1, k_2$ such that $s_{\{\textsc{File},j_1,k_1\},i} = s_{\{\textsc{File},j_2,k_2\},i'} \mod 2$.*

Proof (Sketch). This proof is along the same lines as Lemma 2. As α values are randomized, respectively, any pair of $(s_{\{\textsc{File},j_1,k_1\},i}, s_{\{\textsc{File},j_1,k_1\},i'})$ values will look random to $\mathcal{A}$, too. Values $(s_{\{\textsc{File},j_1,k_1\},i}, s_{\{\textsc{File},j_1,k_1\},i'})$ will be different, even if $(s_{\{\textsc{File},j_1,k_1\},i}, s_{\{\textsc{File},j_1,k_1\},i'})$ originate both from queries for the same expression w. □

4.2 Statistical Analysis

The remaining question is how $\mathcal{U}$ can chose parameters t and q to get a certain error probability P_{err}. This probability describes the chance that, despite $w \notin \textsc{File}$, $\mathcal{U}$ wrongly outputs $w \in \textsc{File}$ after q rounds without a contradiction, cf., Algorithm 3.

We consider for simplicity only rounds where $\lfloor h(C,i) \rfloor_1 = 1$, cf., Algorithm 3. If h is a cryptographic hash function, then $\lfloor h(C,i) \rfloor_1 = 1$ in $q' \approx \frac{q}{2}$ of the q total rounds.

First, we recall some basic probabilities. In one InputSplit, the probability for a collision is

$$P_{\text{collision}} := \frac{1}{t^2}.$$

In one InputSplit, the probability for a one-collision is

$$P_{\text{one-collision}} := \frac{P_{\text{collision}}}{2}.$$

If w is *not* inside a particular InputSplit, the probability that, after inserting the n ciphertexts of that InputSplit into $\mathcal{M}$, the candidate position is *not* set to 1 by collision computes to

$$P_{\text{InputSplit,no-one-collision}} := (1 - P_{\text{one-collision}})^n.$$

Now, if $w \notin \textsc{File}$, i.e., in *none* of the InputSplits, the probability that the candidate position is not set to 1 by collision in *any* InputSplits of $\textsc{File}$ is

$$P_{\text{contradiction}} := (P_{\text{InputSplit,no-one-collision}})^c.$$

This is the probability that a contradiction occurs in a single round. If $w \notin \textsc{File}$, the probability that a contradiction occurs in *at least one* of the q rounds is

$$P_{\text{contradiction,q-rounds}} := 1 - (1 - P_{\text{contradiction}})^q.$$

After q rounds without a contradiction, $\mathcal{U}$ automatically decides that w is in $\textsc{File}$. In case that $w \notin \textsc{File}$, and *no* contradiction occurs in q rounds, $\mathcal{U}$ is therefore wrong with probability

$$P_{\text{err}} := 1 - P_{\text{contradiction,q-rounds}} = (1 - (1 - \frac{1}{2 \cdot t^2})^{cn})^q.$$

Given a certain file size, the size of InputSplits, and the blocksize of the symmetric cipher, $\mathcal{U}$ computes c and n. Therewith, $\mathcal{U}$ can target a false-positive probability by appropriately selecting t and q. We evaluate this using a real-world scenario in Section 5.

5. EVALUATION

To show its real-world feasibility, we have implemented and evaluated PRISM. The source code is available for public download [21]. We received 16 days of log data from May 2010 from a small local Internet provider. This provider logs and retains all customers' DNS resolve requests for possible forensic analysis and intrusion detection. Log data is split into files on a daily basis. Each file contains one day of logged 3-tuples (timestamp, customer IP, hostname). The scenario for our evaluation is to use PRISM to upload this data (encrypted) to MapReduce and perform searches for specific hostnames in a privacy-preserving manner. This is useful for, e.g., "passive DNS analysis" to determine at which day certain command-and-control centers of botnets have been accessed by customer machines, cf., Bilge et al. [4]. The goal of our experiments was to analyze the computational overhead induced by PRISM's privacy mechanism, i.e., the additional time consumed by PRISM over non-privacy-preserving MapReduce.

5.1 Setup

For the period of 16 days, the log data contains $\approx 3 \cdot 10^8$ log entries, i.e., $\approx 2 \cdot 10^7$ per file/day. The total space required by all files uploaded into MapReduce using PRISM is 27 GByte, on average 1.7 GByte per file.

Our experiments have been performed on a small "cloud" comprising 1 master computer and 9 slaves. Computers featured a Pentium Dual Core with 2.5 GHz clock frequency and 4 GByte of RAM and running a standard desktop installation of Fedora 11. With this hardware configuration, a total of 18 CPUs were available for maps and reduces. We installed Hadoop version 0.20.2 on our cloud. Being aware that tailoring MapReduce's configuration parameters can have a huge impact on performance, we use the standard, out-of-the-box configuration of Hadoop 0.20.2 without any configuration tweaks. Performance tuning is out of scope of this paper. Similarly, as the InputSplit size is recommended to be between 64 MByte and 128 MByte, we chose $S_{\text{InputSplit}} = 96$ MByte (our InputSplits have to be dividable by $3 \cdot 32$ Byte).

In addition to the evaluation with 96 MByte InputSplits, we also performed a second measurement with slightly larger InputSplits of 120 MByte. We expected a slightly improved performance of PRISM, due to the fact that for the larger files the total number of InputSplits c reduced to less than our 18 available CPUs. Therewith, no additionally costly (re-)scheduling takes places and mappers do not have to process 2 InputSplits sequentially.

Finally, to put timing results into perspective, we also implemented and measured a trivial, non-privacy-preserving MapReduce job called *Baseline*. Baseline consists of an empty map phase, were each mapper simply scans over its InputSplit, but does not generate any key-value pairs out of the input. Only at the end of the map phase, a single intermediate key-value pair per mapper is sent to reducers. Upon receipt, reducers discard this key-value pair and write empty files to disk. This trivial baseline only serves in deducing the overhead implied by PRISM, not taking MapReduce specific delays due to rescheduling, speculative execution of backup tasks etc. [19, 38] into account.

For the trapdoor group private information retrieval algorithm, we set $m = 400$ as suggested by Trostle and Parrish [41] for good security and privacy. Our Java implementation is a naive, straightforward implementation using Java's BigInteger without any performance optimizations. As symmetric encryption cipher, we used AES with 256 Bit blocksize from the GNU Crypto Library V2.0.1 [23]. For our IND-CO extension we reserved $|\gamma| = 2$ Byte and truncated DNS hostnames longer than 30 Byte down to the last 30 Byte.

Table 1: Parameters t, q to achieve $P_{\text{err}} < 0.01$.

	File size (GByte)							
	0.45	1.21	1.32	1.36	1.38	1.45	1.52	1.67
t	2^{10}	2^{11}						
q	100	60					80	

	File size (GByte)							
	1.78	1.93	2	2.08	2.09	2.14	2.21	2.25
t	2^{12}							
q	20							

Simulating $\mathcal{U}$, we computed n and c using blocksize, InputSplit size $S_{\text{InputSplit}}$,, and individual file size S_{File}. Assuming that $\mathcal{U}$ targets an error probability of $P_{\text{err}} < 0.01$, Table 1 sums up parameters (t, q) computed for each file individually. Compared to q, we observed that parameter t has a much higher impact on $P_{\text{contradiction}}$, but a comparatively lower impact on computations. Therefore, we increased preferably t than q. Higher values for (t, q) will achieve even smaller values for P_{err}, but Table 1 shows the computationally "cheapest" combination of (t, q).

5.2 Results

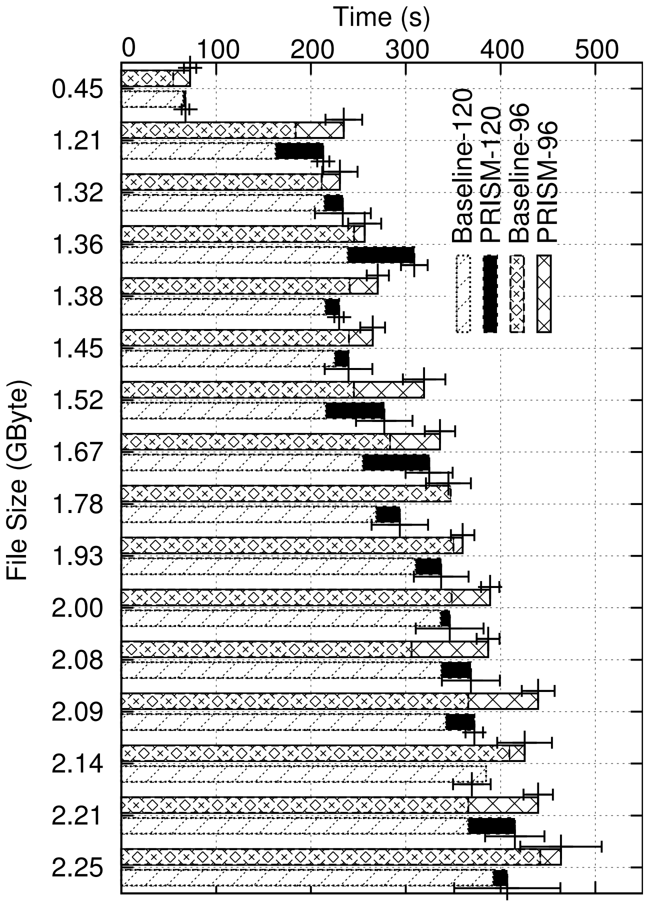

Figure 2: Wall clock timings for PRISM and Baseline protocol runs with $S_{\text{InputSplit}} = 96$ MByte and $S_{\text{InputSplit}} = 120$ MByte.

Figure 2 presents PRISM's timing results. We have sorted the 16 files based on their size in an increasing order, i.e, the size of the smallest log file we received from the Internet provider was 0.45 GByte, the largest one was 2.25 GByte. PRISM's execution time was clocked on each file 6 times, respectively, and Fig. 2 shows the average. For each file, Fig. 2 shows two stacked boxes, respectively: the first one for 96 MByte and the second one for 120 MByte InputSplit size. Each of the stacked boxes comprises, first, the baseline timing, and, second, the additional time required to run PRISM. To give trust into the evaluation, Fig. 2 also shows 95% confidence intervals drawn right next to each box.

Timings shown in Fig. 2 are "wall clock" timings. This captures the complete time elapsed from submitting the

PRISM map and reduce classes and starting the job until the end of the reduce phase.

In conclusion the additional overhead over the trivial Baseline MapReduce jobs was on average 11% with a 95% confidence interval of ± 3. The largest overhead seen was 24% over Baseline. These results do not only show the feasibility of PRISM in practice, but also demonstrate the low overhead implied by PRISM over any non-privacy preserving MapReduce job. We claim that a performance optimized implementation, not based on Java BigInteger, improves performance significantly and furthermore reduces PRISM's overhead.

The simple increase from 96 MByte InputSplit size to 120 MByte InputSplit size has reduced wall clock times for MapReduce jobs on by 9% on average (95% confidence interval of ± 4). Files of size smaller than 2 GByte are split into ≤ 18 InputSplits, and both jobs, PRISM and Baseline, are processed completely in parallel. This demonstrates that a careful configuration of Hadoop MapReduce's many system parameters, handcrafted and specific to the scenario and jobs to be executed, will lead to substantial performance improvements. This also indicates that in a cloud with more CPUs than in our small setup, the increased number of CPUs will enable to configure way *smaller* InputSplits being processed in parallel. Substantially smaller InputSplits will be beneficial for the overall performance of PRISM or any MapReduce job. However, increasing the number of InputSplits also implies a performance penalty due to (re-)scheduling and coordination activities of the central job tracker, cf., Pavlo et al. [38], so a trade-off has to be found. MapReduce configuration optimization are, however, out of scope of this paper.

Computational overhead for $\mathcal{U}$ is extremely low in PRISM: per file, the preparation of, e.g., $2^{12} = 4096$ α values on a desktop computer with a 2.5 GHz CPU is barely measurable (≈ 200 ms). During result analysis, $\mathcal{U}$ automatically discards all received values that he is not interested in, i.e., all besides $s_{\text{FILE},\mathcal{Y},1\leq i\leq q}$. For these q values, a total of q BigInteger multiplications with modulo. For our examples with $q \leq 100$, this was not measurable at < 1 ms. Communication overhead from $\mathcal{U}$ to MapReduce is, besides .class files, only the t α values. With, e.g., $t = 2^{12}$ and $m = 400$ Bit, this computes to 200 KByte. Response from the cloud is, for each round, t values of size m. The most expensive configuration in terms of communication in our experiments was $t = 2^{10}, q = 100$; this results in ≈ 5 MByte communication overhead. Note that communication complexity in the PIR scheme by Trostle and Parrish [41] is linear in the square root of the total table size, i.e., $O(t)$. This can be further reduced by using recursive PIR queries to $O(t^{\epsilon})$, for any $\epsilon > 0$ [30]. Those optimizations, as well as, e.g., amortization techniques discussed by Ostrovsky and Skeith [37] are out of scope. In conclusion, PRISM is very lightweight for a client using standard PC hardware.

Discussion: While gross timings for queries are high, we point out that most of the time is required by the core MapReduce framework, and PRISM adds only a small computational overhead. Note that on a larger cluster in a more professional environment (hundreds or even thousands of CPUs [28]), all files will be processed in parallel. As you can see in Fig. 2, total time for the 2 GByte file is ≈ 350 s (≈ 6 min). However, already ≈ 340 s are required by MapReduce just to "scan" through the various InputSplits, see Baseline. This has been reported previously, and our results are along the lines of Pavlo et al. [38]. Here, a "grep"-like MapReduce job on 1 TByte of data was clocked at $\approx 1,500$ s on 50 CPUs which would be ≈ 20 times faster than our Baseline. However, Pavlo et al. [38] use a slightly tuned configuration and moreover a more efficient scanning through InputSplits (100 Byte text values instead of 32 Byte binary values in our case) which is known to lead to significant performance increases [29].

6. RELATED WORK

Cloud Security: Current security research for cloud computing tackles rather traditional issues such as data confidentiality [43], unauthorized access [42] or Denial of Service attacks [27, 32]. Most solutions focus on the protection of the cloud from malicious users. Roy et al. [39] propose a security solution dedicated to MapReduce. Unlike PRISM, the cloud user is the adversary and the cloud is protected against a curious users. In contrast, PRISM considers the *cloud* as a privacy threat.

Private Information Retrieval: Private Information Retrieval (and similarly oblivious transfer and oblivious RAM) has received a lot of attention [8, 14, 15, 22, 25, 30, 36, 37, 37]. In PIR, a user retrieves a specific date from a database. The only "privacy" goal in these schemes is access privacy whereby the server should not discover which date a user is interested in. Note that PIR does not ensure privacy of data in the database. PRISM, however, focuses on searching for an expression and uses PIR as a tool.

Searchable Encryption: With searching on encrypted data techniques [5], user privacy is guaranteed thanks to the encryption of the queries and the stored data. However, PRISM offers *higher* privacy guarantees since in existing searchable encryption solutions [3, 5–7, 9, 18, 24, 35, 40], the result ("found" or "not found") originating from a query is known to the adversary. So, the server can obtain some knowledge on users' queries. In PRISM, the adversary does not learn anything about queries or results. Moreover, existing searchable encryption mechanisms *cannot* be extended to leverage from a parallelized cloud setup: while in theory the search on encrypted data itself could be run in parallel on subsets of data, the *combination* of results (as in a reduce phase) is impossible in today's solutions.

7. CONCLUSION

PRISM is the first privacy-preserving search scheme suited for cloud computing. That is, PRISM provides storage and lookup privacy with very high performance by leveraging the efficiency of the MapReduce paradigm with the privacy guarantees of a PIR scheme. Moreover, the scheme is compatible with any MapReduce-based cloud infrastructure since it only requires a standard MapReduce interface (such as Amazon's) without any modification in the underlying system. Thanks to this compatibility, PRISM has been efficiently implemented on an experiemental cloud computing environment using Hadoop MapReduce. Performance of PRISM has been evaluated on that environment through search operations in DNS logs provided by a telco operator. The overhead of PRISM over baseline search operations (without the privacy support) was found to be 11% on the average, acertaining the efficiency of PRISM.

References

[1] Amazon. Elastic mapreduce, 2010. `http://aws.amazon.com/elasticmapreduce/`.

[2] Apache Hadoop. Welcome to apache hadoop, 2010. `http://hadoop.apache.org/`.

[3] Steven M. Bellovin and William R. Cheswick. Privacy-enhanced searches using encrypted Bloom filters. Technical Report CUCS-034-07, Department of Computer Science, Columbia University, September 2007. `http://tinyurl.com/3v5g9rv`.

[4] L. Bilge, E. Kirda, C. Krügel, and M. Balduzzi. Exposure: Finding malicious domains using passive dns analysis. In *Proceedings of 18th Annual Network and Distributed System Security Symposium (NDSS)*, pages 195–211, San Diego, USA, 2011. ISBN 1891562320.

[5] D. Boneh, G. DiCrescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Proceedings of Eurocrypt 2004*, pages 506–522, Barcelona, Spain, 2004. LNCS 3027. ISBN 978-3-540-72539-8.

[6] D. Boneh, E. Kushilevitz, and R. Ostrovsky. Publickey encryption that allows pir queries. In *Proceedings of Crypto 2007*, pages 50–67, Santa Barbara, USA, 2007. LNCS 4622. ISBN 978-3-540-74142-8.

[7] G. Brassard, C. Crépeau, and J.-M. Robert. All-or-nothing disclosure of secrets. In *Proceedings on Advances in Cryptology, CRYPTO*, pages 234–238, Santa Barbara, USA, 1986.

[8] C. Cachin, S. Micali, and M. Stadler. Computationally private information retrieval with polylogarithmic communication. In *Proceedings of Advances in Cryptology, EUROCRYPT*, pages 402–414, Prague, Czech Republic, 1999. ISBN ISBN 3-540-65889-0.

[9] Y.-C. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *Proceedings of Applied Cryptography and Network Security (ACNS)*, volume 3531, pages 442–455. LNCS, 2005. ISBN 3-540-26223-7.

[10] Y. Chen and R. Sion. On securing untrusted clouds with cryptography. In *Workshop on Privacy in the Electronic Society*, Chicago, USA, 2010.

[11] Chief Information Officer's Council (CIO). Cloud computing: Benefits and risks of moving federal it into the cloud, 2010. `http://tinyurl.com/6hbzx5y`.

[12] Chief Information Officer's Council (CIO). Proposed security assessment & authorization for u.s. government cloud computing, 2010. `http://tinyurl.com/28yk9fe`.

[13] Chief Information Officer's Council (CIO). Privacy recommendations for the use of cloud computing by federal departments and agencies, 2010. `http://tinyurl.com/2czrhn3`.

[14] B. Chor and M. Naor N. Gilbao. Private information retrieval by keywords. Technical Report TR CS0917, Department of Computer Science, Technion, 1998. `http://eprint.iacr.org/1998/003.ps.gz`.

[15] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Proceedings of the 36th Annual IEEE Symposium on Foundations of Computer Science*, pages 41–51, Milwaukee, USA, 1995.

[16] Cloud Security Alliance. Security guidance for critical areas of focus in cloud computing, 2009. `http://tinyurl.com/ycrchqj`.

[17] Cloud Security Alliance. Top threats to cloud computing, 2010. `http://tinyurl.com/yer9tvs`.

[18] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of ACM Conference on Computer and Communications Security, CCS*, pages 79–88, Alexandria, USA, 2006.

[19] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, USA, 2004.

[20] EU. Eu information management instruments, 2010. `http://tinyurl.com/3289efv`.

[21] EURECOM. PRISM source code, 2011. `http://www.eurecom.fr/~blass/prism.tgz`.

[22] Y. Gertner, Y. Ishai, and E. Kushilevitz. Protecting data privacy in private information retrieval. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 151–160, Dallas, USA, 1998. ACM. ISBN 0-89791-962-9.

[23] GNU. The gnu crypto project, 2011. `http://www.gnu.org/software/gnu-crypto/`.

[24] E.-J. Goh. Secure indexes. Cryptology ePrint Archive Report 2003/216, 2003. `http://eprint.iacr.org/2003/216`.

[25] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious ram. *Journal of the ACM*, 45:431–473, May 1996. ISSN 0004-5411.

[26] S. Goldwasser and S. Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *Proceedings of 14th ACM symposium on Theory of Computing*, pages 365–377, 1982. ISBN 0-89791-070-2.

[27] B. Grobauer and T. Schreck. Towards incident handling in the cloud: Challenges and approaches. In *Proceedngs of the ACM Cloud Computing Security Workshop, CCSW*, Chicago, USA, 2010. ISBN 978-1-4503-0089-6.

[28] Hadoop. Powered by hadoop, 2010. `http://wiki.apache.org/hadoop/PoweredBy`.

[29] D. Jian, B.C. Ooi, L. Shi, and S. Wu. The performance of mapreduce: An in-depth study. *Proceedings of the VLDB Endowment*, 3(1):472–483, 2010.

[30] E. Kushilevitz and R. Ostrovsky. Replication is not needed: single database, computationally-private information retrieval. In *Proceedings of 38th Annual Symposium on Foundations of Computer Science*, pages 364–373, Miami Beach, USA, 1997. ISBN 0-8186-8197-7.

[31] J. Li, A. Wang, C. Wang, and W. Lou N. Cao an K. Ren. Fuzzy keyword search over encrypted data in cloud computing. In *Proceedings of INFOCOM Mini-Conference*, pages 441–445, San Diego, USA, 2010. ISBN 978-1-4244-5838-7.

[32] H. Liu. A new form of DOS attack in a cloud and its prevention mechanism. In *Proceedngs of the ACM Cloud Computing Security Workshop, CCSW*, Chicago, USA, 2010. ISBN 978-1-4503-0089-6.

[33] D. McCullagh. Fbi wants records kept of web sites visited, 2010.
`http://tinyurl.com/yh5omkg`.

[34] Official Google Enterprise Blog. Google apps for government, 2010.
`http://tinyurl.com/3pnabxz`.

[35] W. Ogata and K. Kurosawa. Oblivious keyword search. *Journal of Complexity – Special issue on coding and cryptography*, 20:356–371, April 2004. ISSN 0885-064X.

[36] R. Ostrovsky and W. Skeith. Private searching on streaming data. In *Proceedings of Advances in Cryptology, CRYPTO*, pages 223–240, Santa Barbara, USA, 2005. ISBN 3-540-28114-2.

[37] R. Ostrovsky and W.E. Skeith. A survey of single-database private information retrieval: techniques and applications. In *Proceedings of the 10th international conference on Practice and theory in public-key cryptography*, pages 393–411, Beijing, China, 2007. ISBN 978-3-540-71676-1.

[38] A. Pavlo, E. Paulson, A. Rasin, D.J. Abadi, D.J. De-Witt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 165–178, Rhode Island, USA, 2009. ISBN 978-1-60558-551-2.

[39] I. Roy, S. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for mapreduce. In *Proceedings of NSDI*, pages 297–312, San Jose, USA, 2010. ISBN 978-931971-73-7.

[40] D.X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 44–55, Berkeley, California, 2000.

[41] J. Trostle and A. Parrish. Efficient computationally private information retrieval from anonymity or trapdoor groups. In *Proceedings of 13th International Conference on Information Security*, pages 114–128, Boca Raton, USA, 2010. ISBN 978-3-642-18177-1.

[42] W.Wang, Z. Li, R. Owens, B. Bhargava, and M. Linderman. Secure and efficient access to outsourced data. In *Proceedings of the ACM Cloud Computing Security Workshop, CCSW*, Chicago, USA, 2009. ISBN 978-1-60558-784-4.

[43] A. Yun, C. Shi, and Y. Kim. On protecting integrity and confidentiality of cryptographic file system for outsourced storage. In *Proceedings of the ACM Cloud Computing Security Workshop, CCSW*, Chicago, USA, 2009. ISBN 978-1-60558-784-4.