PRISM – Privacy-Preserving Search in MapReduce

Erik-Oliver Blass[1], Roberto Di Pietro[2], Refik Molva[3], and Melek Önen[3]

[1] Northeastern University, Boston, MA
[2] Università di Roma Tre, Rome, Italy
[3] EURECOM, Sophia Antipolis, France

Abstract. We present PRISM, a privacy-preserving scheme for word search in cloud computing. In the face of a curious cloud provider, the main challenge is to design a scheme that achieves privacy while preserving the efficiency of cloud computing. Solutions from related research, like encrypted keyword search or Private Information Retrieval (PIR), fall short of meeting real-world cloud requirements and are impractical. PRISM's idea is to transform the problem of word search into a set of parallel instances of PIR on small datasets. Each PIR instance 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. Due to the linearity of PRISM, the simple aggregation of map results yields the final output of the word search operation. We have implemented PRISM on Hadoop MapReduce and evaluated its efficiency using real-world DNS logs. PRISM's overhead over non-private search is only 11%. Thus, PRISM offers privacy-preserving search that meets cloud computing efficiency requirements. Moreover, PRISM is compatible with standard MapReduce, not requiring any change to the interface or infrastructure.

1 Introduction

Today, users take advantage of public clouds operated by large companies like Google or Amazon. Instead of setting up and maintaining their own data centers, users reduce their costs by outsourcing both storage and processing to a cloud. One prominent example allowing cloud-based storage and processing is Hadoop MapReduce [3], a variant of Google's MapReduce system [17]. Hadoop MapReduce is widely used, and public MapReduce clouds are offered by companies such as Amazon [2, 25].

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, 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 [11, 12, 18]: these regulations usually aim at assuring privacy against *curious clouds* or against clouds with data centers located in "rogue" countries or with insufficient security guarantees; they also are defined to avoid data leakage in case of operational failures in the cloud. Along these lines, there is also a raising corporate concern about the privacy of sensitive business data stored in the cloud [14]. Although cloud providers thrive to meet the increased privacy demand by certifying their services [24], malicious insiders have still been identified as one of the top threats in cloud computing [15].

While encryption of outsourced data by the users seems to be a viable protection against most privacy problems, traditional data encryption does not suit the requirements of cloud computing: the cloud not only serves as high capacity memory, but is also involved in data processing such as statistical data analysis, log analysis, indexing, data mining, and searching [25]. However, data processing performed by the cloud would not be feasible or would be *inefficient* with encrypted data.

Among data processing primitives, word search, i.e., verifying, whether a certain word is part of a dataset, is not only one of the most fundamental operations, but surprisingly also one of the most demanded applications in, e.g., MapReduce cloud computing [25]. **Related work** on search in encrypted data, e.g., Boneh et al. [6], Ogata and Kurosawa [31], falls short of meeting cloud computing privacy and performance requirements. These techniques are *impractical* as they are designed for centralized execution models that are incompatible with today's highly parallel cloud architectures.

In this paper, we present PRISM, a new scheme for *privacy-preserving and efficient word search* for MapReduce clouds. PRISM pursues two specific objectives: 1.) privacy against potentially malicious cloud providers and 2.) high efficiency through the integration of security mechanisms with the operations performed in the cloud.

In order to achieve efficiency, PRISM takes advantage of the inherent parallelization akin to cloud computing: the word search problem on a very large encrypted dataset is partitioned into several instances of word search in small datasets that are executed in parallel ("Map" phase). The individual word search operations performed in the cloud yield a result amenable to straightforward *aggregation* in the ultimate phase ("Reduce" phase) of the word search operation. The word search operation builds on a Private Information Retrieval (PIR) [29] technique which is extended in order to generate intermediate search results that are still encrypted and that can be *combined* through linear operations to yield the global result of the word search over the entire dataset.

Summarizing our contributions, PRISM:

• **is suited to cloud computing:** PRISM is the first privacy-preserving search scheme suited to cloud computing; it brings together storage and search privacy with high performance by leveraging the efficiency of the MapReduce paradigm. PRISM is parallelizable and also allows efficient combination of individual results. Its efficiency has been evaluated through searching in DNS logs provided by an Internet Service Provider. Although PRISM's overhead within the core Map function is large compared to non-privacy-preserving search (factor of 9), the total system overhead is only 11%.

• **preserves privacy in the face of potentially malicious cloud providers:** PRISM allows carrying out these critical operations in the cloud without trusting the cloud.

• **is compatible to 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 Amazon.

• **provides flexible search:** In contrast to traditional 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 search for any words.

2 Problem Statement and Adversary Model

Throughout this paper, we will use an application example to motivate our work. Inspired by recent events [30], we envision a *data retention* scenario. Due to regulatory matters, a small, residential Internet Service Provider (ISP) must retain logs of client accesses. Due to the sheer amount of data to retain, the ISP outsources logfiles to the cloud. Files are encrypted as they contain sensitive data. Still, e.g., law enforcement authorities will contact the ISP to search for words (strings, text, ...) in outsourced files.

More concretely, 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 to (or is forced to) find out, whether there was an interest in a suspicious host w. So, $\mathcal{U}$ checks, at which day, i.e., in which logfiles L_i, word w occurs. $\mathcal{U}$ queries the cloud for w, and the cloud responses with an answer R telling $\mathcal{U}$ which of the L_i contain(s) w.

Note that $\mathcal{U}$ does not know in advance which word w it has to search for. This automatically disqualifies protocols for *predefined* keyword search, such as PEKS [6] and derivatives. Also, data retention regulations require outsourced data to be fully recoverable; storing only digests of data in the cloud, e.g., hash values, is insufficient.

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. This implies both, the encryption of data by $\mathcal{U}$ before outsourcing it to the cloud and "obliviously" processing queries on encrypted data by the cloud.

Before we formalize our privacy requirements, we first define the main components for a cloud word search scheme.

Definition 1 (Cloud Word Search). *Let $\mathcal{L}$ denote a sequence of files $\mathcal{L} := \{L_1, \ldots, L_m\}$ and Σ the set of possible words. Each file L_i consists of a sequence of words $L_i := \{w_{(i,1)}, w_{(i,2)}, \ldots, w_{(i,|L_i|)}\}, w_{(i,j)} \in \Sigma$.*

A cloud word search scheme comprises the following algorithms:

1. *$KeyGen(s)$: using a security parameter s, this algorithm outputs a secret $\mathcal{S}$.*
2. *$Encrypt(\mathcal{S}, \mathcal{L})$: uses the secret $\mathcal{S}$ to encrypt the content of files $L_i \in \mathcal{L}$ and outputs the set of resulting encryptions $\mathcal{E} := \{E_{L_1}, \ldots, E_{L_m}\}$. Here, E_{L_i} denotes the encryption of file L_i.*
3. *$Upload(\mathcal{E})$: uploads $\mathcal{E}$ to the cloud.*
4. *$PrepareQuery(\mathcal{S}, w)$: takes $\mathcal{S}$, the word w to search for, and produces a query Q.*
5. *$Process(\mathcal{E}, Q)$: with encryptions $\mathcal{E}$ and query Q, this algorithm produces result R.*
6. *$Decode(\mathcal{S}, R, w)$: taking result R, secret $\mathcal{S}$, and word w, this algorithm outputs the set of indices $\mathcal{I} := \{i_1, \ldots, i_r\}$ such that $\forall i \in \mathcal{I} : L_i \in \mathcal{L} \wedge w \in L_i$, if $R = Process(\mathcal{E}, Q)$ with $Q = PrepareQuery(\mathcal{S}, w)$, and $\mathcal{E} = Encrypt(\mathcal{S}, \mathcal{L})$.*

The basic interaction between user $\mathcal{U}$ and the cloud can be summarized as follows: first, user $\mathcal{U}$ encrypts and uploads files. Then, $\mathcal{U}$ prepares a search query Q for word w and sends Q to the cloud. The cloud processes this query using algorithm $Process$

and produces output R. This output is sent back to $\mathcal{U}$. Using output R and another algorithm $Decode$, $\mathcal{U}$ can compute the list of files containing w. While describing PRISM's details later in Section 4, we will show how they map to these algorithms.

Note that, in a cloud setting, $\mathcal{U}$ executes $KeyGen$, $Encrypt$, $Upload$, $PrepareQuery$, and $Decode$, while the cloud executes $Process$. The idea is that algorithms $KeyGen$, $Encrypt$, $Upload$, $PrepareQuery$, and $Decode$ are computationally very "lightweight" for $\mathcal{U}$ compared to $Process$. The main computational burden lies on the cloud side.

Privacy Requirements

Intuitively, our application demands for two main types of privacy. The cloud (now called "adversary $\mathcal{A}$") must neither be able to infer any details about stored files nor learn details about $\mathcal{U}$'s queries and results delivered back to $\mathcal{U}$. This implies not only the secrecy or confidentiality of the content, but also the inability to compute statistics on the content. Informally, in our setting:

- given $\mathcal{E}$, $\mathcal{A}$ must not learn the content of $\mathcal{L}$ and must not discover whether files L_i contain a word w, e.g., multiple times;
- given a set of queries $\{Q_i\}$, $\mathcal{A}$ must not learn the words $\{w_i\}$ $\mathcal{U}$ is looking for and must not discover whether the same word is queried multiple times;
- given the result R_i of a query Q_i, $\mathcal{U}$ must not learn which file(s) contain the word corresponding to this specific query W_i.

Instead, the adversary should only learn "trivial" properties, such as the total number of files, the file size, and the total number of queries. Along the same lines as traditional indistinguishability [23], $\mathcal{A}$ should not be able to infer any additional information from encrypted files, queries, and results. We formally define privacy for a cloud word search scheme using a game between adversary $\mathcal{A}$ (the cloud) and a challenger (user $\mathcal{U}$).

Definition 2 (Privacy). *Let $\mathcal{W}$ denote a sequence of words $\mathcal{W} := \{w_1, \ldots, w_n\}$. The game* GAME *is played as follows.*

1. *The challenger executes $KeyGen(s)$ to derive secret $\mathcal{S}$.*
2. *$\mathcal{A}$ selects a distinct pair of sequences of files and words $(\mathcal{L}_0, \mathcal{W}_0)$ and $(\mathcal{L}_1, \mathcal{W}_1)$, where $|\mathcal{L}_0| = |\mathcal{L}_1|$, $\forall (L_i^0 \in \mathcal{L}_0, L_i^1 \in \mathcal{L}_1) : |L_i^0| = |L_i^1|$, and $|\mathcal{W}_0| = |\mathcal{W}_1|$. $\mathcal{A}$ sends $(\mathcal{L}_0, \mathcal{W}_0)$ and $(\mathcal{L}_1, \mathcal{W}_1)$ to the challenger.*
3. *The challenger randomly selects $b \in \{0, 1\}$ and*
   - *executes $Encrypt(\mathcal{S}, \mathcal{L}_b)$, i.e., the challenger computes encrypted files $\mathcal{E}_b = \{E(\mathcal{S}, L_i) | L_i \in \mathcal{L}_b\}$.*
   - *executes $Upload(\mathcal{E}_b)$ to send encrypted files back to $\mathcal{A}$.*
   - *executes $PrepareQuery(\mathcal{S}, w)$ for each w in $\mathcal{W}_b$. This results in the sequence of queries $\mathcal{Q}_b := \{Q_1, \ldots, Q_{|\mathcal{W}_b|}\}$ that the challenger also sends to $\mathcal{A}$.*
4. *$\mathcal{A}$ outputs $b' \in \{0, 1\}$. The outcome of* GAME *is "1" iff $b' = b$.*

A cloud word search scheme is called privacy-preserving iff

$$Pr(\textsc{Game}(\mathcal{A}) = 1) \leq \frac{1}{2} + \epsilon(s)$$

for all probabilistic polynomial-time adversaries $\mathcal{A}$. Here, $\epsilon(s)$ is a negligible function, $\epsilon(s) < \frac{1}{P(s)}$ for every polynomial P with sufficiently large security parameter s.

The specification of the $(\mathcal{L}_i, \mathcal{W}_i)$ in step 2. of Definition 2 reflects the fact that $\mathcal{A}$ can learn the total number of files, the size of each file, and the number of queries.

Limitations: We consider semi-honest clouds. Fully malicious clouds might perform DoS-attacks or deviate from protocol execution. Similar to "reaction attacks" [26], the cloud might return garbage to $\mathcal{U}$, to observe $\mathcal{U}$'s reaction (e.g., sending the same query). Although realistic, we leave such attacks for future work. Also, our privacy definition does not capture trivial privacy properties, e.g., the size of outsourced files. Mitigation strategies (e.g., padding files) might be contradictory to cloud efficiency. We conjecture that, for many applications, losing "trivial" privacy properties is acceptable.

3 Background

3.1 MapReduce

We target a system suited for the MapReduce [3] paradigm. We will now give a condensed overview of MapReduce, focusing on aspects necessary to understand PRISM.

Upload. A MapReduce cloud comprises a set of "slave" node computers and a "master" computer. While $\mathcal{U}$ uploads files into the MapReduce cloud, each file is automatically split into blocks called *InputSplits*. InputSplits have a fixed size $S_{\text{InputSplit}}$ which is a pre-configured system parameter. If S_{File} denotes the size of an uploaded file, the number of InputSplits c computes to $c = \frac{S_{\text{File}}}{S_{\text{InputSplit}}}$. For each InputSplit, a workload sharing algorithm selects a slave node and places the InputSplit on it.

In addition to data, the MapReduce also allows $\mathcal{U}$ to upload "operations", i.e., compiled Java classes. These classes 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 sent to all slave nodes storing an InputSplit.

Map Phase. After data and implementations have been uploaded, $\mathcal{U}$ specifies one uploaded file and triggers MapReduce operations on that file. The first phase of operation is the "Map" phase. Each slave node becomes a "mapper" node. Each mapper executes $\mathcal{U}$'s $Scan$ function on the InputSplit it stores locally. 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. MapReduce starts the “Reduce” phase. Slave nodes are scheduled to become “reducers”. For each of the intermediate pairs (k', v'), MapReduce selects a reducer and sends (k', v') to this reducer. MapReduce selects the *same* reducer for all pairs (k', v') having the same key. 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 $\mathcal{U}$.

3.2 Trapdoor Group Private Information Retrieval

PIR allows a user to retrieve data from a server without revealing which data is retrieved. For PRISM, we make use of a simple and efficient PIR mechanism as previously suggested by Trostle and Parrish [37]. As this mechanism is just a building block for PRISM, we will only give a 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 only interested in receiving elements of the k^{th} row in $\mathcal{M}$, but the server must not learn k. 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 $\mathbb{Z}_p$, with a prime p of m bits. $\mathcal{U}$ 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 $\boldsymbol{u}$ be the vector $\boldsymbol{u} := (\alpha_1, \ldots, \alpha_t)$. The server computes the matrix product $\boldsymbol{v}$ and sends it back to $\mathcal{U}$,

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

Result analysis: Upon receipt, in order to “un-blind” 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 $\mathcal{M}$. Therewith, $\mathcal{U}$ has retrieved the t elements of the k^{th} row of $\mathcal{M}$ in a privacy-preserving fashion. Note the linearity for two β_i and β_i' received during different PIR runs: $\beta_i \cdot b^{-1} + \beta_i' \cdot b^{-1} = (\beta_i + \beta_i') \cdot b^{-1} \mod p$. That is, the sum of two individually un-blinded vectors equals un-blinding the sum of two received vectors $\boldsymbol{v}, \boldsymbol{v'}$. We will later use this linearity during PRISM’s reduce phase.

Security Rationale: Security and privacy of this protocol are based on the trapdoor group assumption. With only knowledge of α_i, but not secret trapdoor p, 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 [37].

Discussion: Again, we stress that this particular PIR scheme is an exchangeable building block. In general, any of the "traditional" PIR techniques based on group homomorphic encryption [29, 33] is suited for use within PRISM. We have chosen Trostle and Parrish [37] only due to the straightforward way to implement it (Section 6). Other PIR schemes might reduce the (already small) overhead, but this is out of scope here.

4 PRISM Protocol

PRISM comprises three parts: *upload* of data into the cloud, the MapReduce *search*, and the *result analysis* where the user decides whether the word has been found. We will briefly give an overview about each part.

1.) Upload. During upload, $\mathcal{U}$ encrypts each word (of a logfile) using symmetric encryption. Ciphertexts are stored in a file, and this file is sent to the MapReduce cloud. The cloud automatically splits large files and distributes splits (*InputSplits*) among *mapper* nodes. We use a standard blockcipher (AES) to perform ciphering of words. However, to ensure privacy as of Definition 2, plaintext is modified before encryption using a "stateful cipher" construction. Therewith, $\mathcal{U}$ can still search for some word w, but the cloud cannot compute statistics about ciphertexts.
2.) Search. Eventually, $\mathcal{U}$ wants to search his encrypted files for some word 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 *reducer* nodes. MapReduce distributes these implementations to each mapper and reducer, respectively. PRISM's rationale is to *transform* the word search problem into a set of small PIR instances. To do so, each mapper, scanning through its locally stored 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 reducers can learn any information about which ciphertext $\mathcal{U}$ was interested in.
3.) Result analysis. Finally, $\mathcal{U}$ 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.

Initialization: Before the actual uploading, initially, and only once, $\mathcal{U}$ has to execute $KeyGen$. In PRISM, $KeyGen$ outputs secret $\mathcal{S} := \{K, N, p\}$, where $|K|$, $|N|$, $|p|$ are

Input: words w_i
Output: ciphertexts C_i uploaded to cloud

```
Initialize all $\gamma$ to 0;
foreach word $w_i$ 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" example and upload to MapReduce

specified by security parameter s. K is a symmetric key, and N and p are Trapdoor Group PIR parameters as presented in Section 3.2.

4.1 Upload

Overview. In our scenario, cloud user $\mathcal{U}$ continuously logs customer access and sends logfiles to the cloud. Each day, $\mathcal{U}$ starts using a new logfile. For simplicity, we assume that entries logged by $\mathcal{U}$ are simple words. Each logfile is encrypted word by word using a "stateful cipher" E_K, and resulting ciphertexts are written to a file, respectively. The encrypted files are sent to the cloud.

Definition 3 (Stateful Cipher). *Given standard symmetric encryption E_K with key K, e.g., AES, we extend E to a* stateful cipher *by adding "counters" γ_{w_i} that count the history of inputs w_i. Each time E encrypts w_i, counter γ_{w_i} is increased by one.*

In conclusion, a stateful cipher is a cipher that knows how often it has encrypted a specific plaintext. The following presents one trivial stateful cipher construction used in PRISM to encrypt before uploading.

Stateful Cipher Example (see Algorithm 1): For simplicity, user $\mathcal{U}$ uses a secret key K to derive a different key for each day d, e.g., $K_d := \mathrm{HMAC}_K(d)$.

For each day, $\mathcal{U}$ maintains a hash table containing the list of counters γ_{w_i} in $\mathcal{U}$'s local storage. At the beginning of each day, $\mathcal{U}$ initializes all counters to 0, i.e., $\gamma_{w_i} = 0$. Now, for each logentry w_i that should be stored in the cloud, $\mathcal{U}$ computes γ_{w_i} and increases γ_{w_i} by 1. Then, $\mathcal{U}$ computes ciphertext $C_i := E_{K_d}(w_i, \gamma_{w_i})$. User $\mathcal{U}$ sends ciphertext C_i to the cloud that stores it in this day's file. For the (AES) encryption $E_{K_d}(w_i, \gamma_{w_i})$, "," denotes an unambiguous pairing of inputs. We discuss the reason for using a "stateful cipher" over using, e.g., a CBC mode of encryption in Section 5.1.

Summarizing, with respect to Definition 1, $Encrypt$ in PRISM takes K to derive a separate key K_d for each file to be encrypted. Actual encryption of each file is performed word by word using the stateful cipher and key K_d, so $\mathcal{E} := \{E_{L_1}, \ldots, E_{L_m}\}$ where $E_{L_i} := \{E_{K_i}(w_1, \gamma_{w_1}), \ldots, E_{K_i}(w_{|L_i|}, \gamma_{w_{|L_i|}})\}$. $Upload$ in PRISM can be regarded as simply sending the encrypted files $\mathcal{E}$ to MapReduce.

4.2 Search

User $\mathcal{U}$ wants to search a set of files for word w within a period 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. With multiple files, all files will be separately (but in parallel) processed with PRISM exactly like with 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 14 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 local InputSplit, a mapper node creates a matrix with all elements initialized to "0". PRISM's main idea is that while the mapper scans the ciphertexts in its InputSplit, each ciphertext is assigned to one position, a certain element in the matrix by computing a hash of the ciphertext. Additionally, for each ciphertext, the mapper computes a single bit hash, and if the hash output bit is "1", the mapper puts a "1" in the matrix at the assigned position. The idea is that user $\mathcal{U}$ can also compute the position in the matrix and the one bit hash output for a word w he is looking for. Roughly speaking, $\mathcal{U}$ now queries the mapper for the value of that bit in the matrix using private information retrieval. If the bit retrieved from the mapper differs from the bit computed by $\mathcal{U}$, then $\mathcal{U}$ can decide, e.g., that w is not in this InputSplit.

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 bit retrieved by $\mathcal{U}$ can therefore be unrelated to w. This problem is amplified by the fact that $\mathcal{U}$ does not only receive a single bit for a single InputSplit, but a combination (the *sum*) of all bits from all mappers working on InputSplits. To mitigate this problem, 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 file.

Definition 4 (PIR Matrix). *A binary $t \times t$ matrix $\mathcal{M}$ with $t = 2^i, i \in \mathbb{N}$ is called a PIR matrix. The mapper uses $\mathcal{M}$ to implicitly perform the privacy-preserving word search.*

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 $(\mathcal{X}_i || \mathcal{Y}_i) := \lfloor C_i \rfloor_{2 \cdot \log_2(t)}$. Here, $\lfloor \ldots \rfloor_{2 \cdot \log_2(t)}$ denotes truncation after $2 \cdot \log_2(t)$ bits. So, the first $\log_2 t$ bits of C_i determine $\mathcal{X}_i$, and the second $\log_2 t$ bits 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 are chosen as for the Trapdoor Group PIR scheme presented in Section 3.2.*

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

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

where $\mathcal{M}_{1 \leq j \leq t, i} = 1$ 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 [37]. 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 $(\mathcal{X}||\mathcal{Y}) := \lfloor E(w, 1) \rfloor_{2 \cdot \log_2(t)}$. If w has been uploaded into a particular InputSplit at least once, then this InputSplit contains at least $E(w, 1)$ (maybe also $E(w, 2)$, $E(w, 3), \ldots$). Therefore, it is sufficient for $\mathcal{U}$ to search for $E(w, 1)$. We will now give detailed descriptions of $PRISM$'s Map and Reduce algorithms.

Query preparation – User. To start, $\mathcal{U}$ chooses parameters $t, q \in \mathbb{N}$, where t determines the size of the PIR matrix and q the number of rounds. For day d that $\mathcal{U}$ wants to search for w, he determines key $K_d := \mathrm{HMAC}_K(d)$ and the target candidate position $(\hat{\mathcal{X}}||\hat{\mathcal{Y}}) := \lfloor E_{K_d}(w, 1) \rfloor_{2 \cdot \log_2(2)}$. To prepare PIR, $\mathcal{U}$ computes t PIR Inputs $\{\alpha_1, \alpha_2, \ldots, \alpha_t\}$ as described above. $\mathcal{U}$ sends all α as part of the following map algorithm implementation to the cloud.

The above preparation of PIR Input depending on w represents $PrepareQuery$ of Definition 1 in PRISM. The algorithm's output Q is the PIR Input. PRISM's implementation of $Process$, i.e., the cloud's operation on the encrypted file using a query Q comprises the following cloud-side Map as well as the whole cloud-side reduce below.

Map Details – Cloud. On the cloud side, all mappers process PRISM in parallel, each of them on its own, locally stored InputSplit of the current file. 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 an element $(\mathcal{X}, \mathcal{Y})$ in matrix $\mathcal{M}_l$.

The mapper node *scans* its local 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 $\mathcal{M}_l, 1 \leq l \leq q$. 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 the bit $bit_j := \lfloor h(C_i, j) \rfloor_1 = 1$. Here, h denotes a cryptographic hash function and "," again an unambiguous pairing of inputs. If $bit_j = 0$, 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* $\sigma_{1 \leq j \leq t, 1 \leq l \leq q}$ based on $\mathcal{U}$'s input $\{\alpha_1, \ldots, \alpha_t\}$: values α_k with corresponding element $\mathrm{M}_{l,k,j}$ set to "1" are simply added. Finally, the mapper outputs intermediate key-value pairs (k, v). The *key* comprises the name of the file of the InputSplit this mapper was working on, e.g., the file name could be 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.

```
Input: pairs $(i, C_i)$, values $\{\alpha_1, \ldots, \alpha_t\}$
Output: intermediate key-value pairs $(k, v)$
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
        $\sigma_{j,l} := \sum_{\mathcal{M}_{l, 1 \leq k \leq t, j} = 1} \alpha_k$;
    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 matrices $\mathcal{M}$

```
Input: reducers' files FILE
Output: decision whether $w \in$ FILE
foreach file FILE do
    for $i := 1$ to $q$ do
        if $\lfloor h(C, i) \rfloor_1 = 1$ then
            $\mathcal{U}$ reads $s_{\text{FILE}, \mathcal{Y}, i}$;
            $s_i := (s_{\text{FILE}, \mathcal{Y}, i} \cdot b^{-1} \bmod p) \bmod N$
            //$s_i = bit_j$, see Alg. 2
            if $s_i = 0$ then
                OUTPUT $w \notin$ FILE; //Contradiction
                break;
            end
        end
    end
    OUTPUT $w \in$ FILE;
end
```

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

Reduce Phase – Overview. Recall that there are c InputSplits and therefore c mappers. 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 $\mathcal{U}$.

Reduce Phase – 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 $(\{\text{FILE}, i\}, \{\sigma_{i,1,1}, \ldots, \sigma_{i,q,1}\}), \ldots, (\{\text{FILE}, i\}, \{\sigma_{i,1,c}, \ldots, \sigma_{i,q,c}\})$.

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

Using integer addition, 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 R. To summarize, $s_{\text{FILE},i,j}$ represents the sum of column sums of all the mappers of one particular column i in PIR matrix j. This concludes the cloud's $Process$ algorithm in PRISM. The output file R is downloaded by $\mathcal{U}$.

4.3 Result Analysis

The only piece left is the $Decode$ algorithm of Definition 1 which we will describe in the following. 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 (using $\mathcal{S}$). 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.

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 $(\hat{\mathcal{X}}||\hat{\mathcal{Y}}) = \lfloor E(w, 1) \rfloor_{2 \cdot \log_2(t)}$ 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, $\mathcal{U}$ distinguishes 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 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 word $\mathcal{U}$ 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, $\mathcal{U}$ for sure knows 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 $\hat{\mathcal{Y}}$ of matrices $\mathcal{M}$, as they can refer to candidate position $(\hat{\mathcal{X}}, \hat{\mathcal{Y}})$, only. Therefore, $\mathcal{U}$ keeps values $\{s_{\text{FILE},\hat{\mathcal{Y}},1}, \ldots, s_{\text{FILE},\hat{\mathcal{Y}},q}\}$ only and discards the rest. In each round where $\lfloor h(C, i) \rfloor_1 = 1$, un-blinds $s_{\text{FILE},\hat{\mathcal{X}},i}$ to get value $s_i := \sum_{j=1}^{c} bit_j$. 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, if $\exists i, s_i = 0$ **or** output $w \in$ FILE, if $\forall i, s_i \neq 0$. We will compute $\mathcal{U}$'s error probability P_{err} for the latter case and dependencies between P_{err} and values t and q in Section 5.2.

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 this computation reduction in our evaluation in Section 6.

5 PRISM Analysis

5.1 Privacy

We will now show why PRISM is privacy-preserving. The main *rationale* behind our proof is to show that pairs of output generated by both our stateful cipher construction $E_K(w_i, \gamma_{w_i})$ (Section 4.1) and the PIR-based search mechanism (Section 4.2) are computationally indistinguishable for $\mathcal{A}$. Below, we assume a sufficiently large security parameter s and probabilistic polynomial time adversaries $\mathcal{A}$.

Theorem 1. *PRISM is a privacy-preserving cloud word search scheme assuming pseudorandom properties for E and the trapdoor group property of the PIR scheme.*

Proof (Sketch). Assume there would be an adversary $\mathcal{A}$ with $Pr(\text{GAME}(\mathcal{A}) = 1) > \frac{1}{2} + \epsilon(s)$, i.e., $\mathcal{A}$ has non-negligible advantage over guessing. As PRISM generates $\mathcal{E}_b$ and $\mathcal{Q}_b$ independently from each other, this would indicate that $\mathcal{A}$ has non-negligible advantage over guessing in determining b from either $\mathcal{E}_b$ or $\mathcal{Q}_b$ (or both).

We will now show with the following two lemmas that this is impossible.

Lemma 1. *In PRISM, any pair of sequences of ciphertexts (files E_L and $E_{L'}$) generated by a pair of sequences of words (files L and L') is computationally indistinguishable for $\mathcal{A}$, assuming E is a pseudorandom permutation and "," an unambiguous pairing of inputs.*

Proof (Sketch). First, note that our stateful-cipher uses a different random key for each file. In a learning phase, $\mathcal{A}$ makes a number of queries to two stateful-cipher oracles encrypting with two different keys K_0, K_1. Then, $\mathcal{A}$ prepares word w, submits to a challenge oracle and gets back $E_{K_b}(w, \gamma_{b,w})$, $b \in \{0, 1\}$. $\mathcal{A}$ has to output b correctly with only negligible advantage over guessing.

However, we now show by using the hybrid argument [28] that the distributions generated by $E_{K_i}(w, \gamma_{i,w})$ are computationally indistinguishable for $\mathcal{A}$. That is, pairs $E_{K_0}(w, \gamma_{0,w})$, $E_{K_1}(w, \gamma_{1,w})$ are distinguishable with only negligible advantage over guessing). As "," is an unambiguous pairing, we now write w'_i instead of $(w, \gamma_{i,w})$.

Our hybrid distributions are: (1.) $PRP_{K_0}(w'_0)$, (2.) $RP_{K_0}(w'_0)$, (3.) $RF_{K_0}(w'_0)$, (4.) $RF_{K_1}(w'_1)$, (5.) $RP_{K_1}(w'_1)$, and (6.) $PRP_{K_1}(w'_1)$. "PRP" means pseudorandom permutation, "RP" random permutation, and "RF" random function.

(1.) - (2.) and (5.) - (6.): by definition of pseudorandom permutation, the probability to distinguish $PRP_K(w'_i)$ from $RP_K(w'_i)$ is negligible.

(2.) - (3.) and (4.) - (5.): the probability to distinguish a random permutation from a random function is negligible, cf., Section 3.6.3 in Katz and Lindell [28].

(3.) - (4.): If $\mathcal{A}$ observes $RF_{K_0}(w'_0) = RF_{K_1}(w'_1)$ for a pair w'_0, w'_1, then this only indicates a collision in RF_{K_0} and RF_{K_1}. Even if $\mathcal{A}$ queries the same w multiple times, output $RF_{K_0}(w, \gamma_{0,w})$ or $RF_{K_1}(w, \gamma_{1,w})$ will always be different as counters increase. If RF_{K_0} (or RF_{K_1}) outputs the same value twice (unlikely), this only indicates a collision in RF_{K_0} (or RF_{K_1}). The advantage over guessing in distinguishing $RF_{K_0}(w'_0)$ from $RF_{K_1}(w'_1)$ is zero.

$\mathcal{A}$'s advantage over guessing in distinguishing pairs $E_L, E_{L'}$ is negligible. □

Lemma 2. *Based on the trapdoor group assumption (*"TGA"*), PRISM's PIR-search produces computationally indistinguishable pairs of queries Q_i.*

Proof (Sketch). Assume $\mathcal{A}$ submits two words w_1, w_2 to an oracle. The oracle picks $\hat{b} \in \{0,1\}$ and returns $Q_{\hat{b}} := \{\alpha_{\hat{b},1} := b \cdot e_{\hat{b},1} \mod p, \ldots, \alpha_{\hat{b},t} := b \cdot e_{\hat{b},t} \mod p\}$, with $e_{\hat{b},\mathcal{X}_{\hat{b}}} := 1 + a_{\mathcal{X}_{\hat{b}}} \cdot N, \forall i \neq \mathcal{X}_{\hat{b}} : e_{\hat{b},i} := a_i \cdot N$. Here, $\mathcal{X}_{\hat{b}} := \lfloor E_K(w_{\hat{b}}, 1) \rfloor_{\log_2(t)}$, and a_i are chosen randomly. $\mathcal{A}$ has to output $\hat{b}$ with non-negligible advantage over guessing.

However, we will now show that any pair of sequences of α values is computationally indistinguishable for $\mathcal{A}$. The proof is a direct implication of the security of the PIR protocol, based on *TGA*: for all adversaries $\mathcal{A}$, $Pr[\mathcal{A}(b \cdot e_1, \ldots, b \cdot e_t) = LSB(e_1, \ldots, e_t)] = \epsilon(s)$. That is, given $b \cdot e_i$, the probability that $\mathcal{A}$ computes low order bits of $e_i \mod N$ ("LSB") is negligible [37].

Assume that $\mathcal{A}$ can distinguish sequences $Q_0 = \{\alpha_{0,i}\}$ and $Q_1 = \{\alpha_{1,i}\}$ with non-negligible advantage. This would violate *TGA* as follows. First, note that besides $\alpha_{0,\mathcal{X}_0}, \alpha_{1,\mathcal{X}_1}$ all elements in both sequences $\{\alpha_{0,i}\}$ and $\{\alpha_{1,i}\}$ are created in the same way (multiplication of b with a random number). Therefore, besides $\alpha_{0,\mathcal{X}_0}, \alpha_{1,\mathcal{X}_1}$, any pair $(\alpha, \alpha') \in \{\alpha_{0,i}\} \cup \{\alpha_{1,i}\}$ is computationally indistinguishable for $\mathcal{A}$. If $\mathcal{A}$ can still distinguish between sequences $\{\alpha_{0,i}\}$ and $\{\alpha_{1,i}\}$, then $\mathcal{A}$ can determine with non-negligible probability $\mathcal{X}_0$ or $\mathcal{X}_1$ and thus value i with $e_i = 1 \mod N$, violating *TGA*. □

5.2 Statistical Analysis

We now discuss how $\mathcal{U}$ chooses parameters t and q to get a certain error probability P_{err}. This probability describes the chance that, despite $w \notin$ FILE, $\mathcal{U}$ wrongly outputs $w \in$ FILE after q rounds without a contradiction, cf., Algorithm 3. Let n be the number of ciphertexts in one InputSplit, $n := \frac{S_{\text{InputSplit}}}{\text{CipherBlockSize}}$. The total number of ciphertexts stored in the cloud is $(c \cdot n)$. We consider for simplicity only rounds where $\lfloor h(C,i) \rfloor_1 = 1$, cf., Algorithm 3. With h a cryptographic hash, $\lfloor h(C,i) \rfloor_1 = 1$ in $q' \approx \frac{q}{2}$ rounds.

While inserting any ciphertext, the collision probability is $P_{\text{collision}} := \frac{1}{t^2}$. The probability for a one-collision is $P_{\text{one-collision}} := \frac{P_{\text{collision}}}{2}$. If w is *not* inside an InputSplit, the probability that, after inserting the n ciphertexts of that InputSplit into $\mathcal{M}$, the candidate position is *not* set to 1 is $P_{\mathrm{InputSplit,no-one-collision}} := (1 - P_{\mathrm{one-collision}})^n$.

If $w \notin$ FILE, i.e., in *none* of the InputSplits, the probability that the candidate position is not set to 1 in *any* InputSplit is $P_{\mathrm{contradiction}} := (P_{\mathrm{InputSplit,no-one-collision}})^c$. This is the probability that a contradiction occurs in a single round. If $w \notin$ FILE, the probability that a contradiction occurs in *at least one* round is $P_{\mathrm{contradiction,q-rounds}} := 1 - (1 - P_{\mathrm{contradiction}})^q$.

After q rounds without a contradiction, $\mathcal{U}$ automatically decides that w is in FILE. In case that $w \notin$ FILE, and *no* contradiction occurs in q rounds, $\mathcal{U}$ is therefore wrong with $P_{\mathrm{err}} := 1 - P_{\mathrm{contradiction,q-rounds}} = (1 - (1 - \frac{1}{2 \cdot t^2})^{cn})^q$.

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 6.

6 Evaluation

To show its real-world feasibility, we have implemented and evaluated PRISM with the scenario described in the introduction. The source code is available for public download [1]. 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 (anonymized by provider for regulatory matters), hostname. The scenario for our evaluation is to use PRISM to upload this data encrypted to MapReduce and perform a search 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. [5]. 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.

6.1 Setup

For the 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 2.5 GHz Pentium Dual Core and 4 GByte of RAM, 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 $S_{\mathrm{InputSplit}} = 96$ MByte (InputSplits must be dividable by $3 \cdot 32$ Byte, since log entries are 3-tuples).

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	1.78	1.93	2.00	2.08	2.09	2.14	2.21	2.25
t	2^{10}	2^{11}							2^{12}							
q	100	60					80		20							

In addition to the evaluation with 96 MByte InputSplits, we also performed a second measurement with 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 costly (re-)scheduling takes places, and mappers do not have to process 2 InputSplits sequentially.

Finally, to put timing results into perspective, we implemented and measured a trivial, non-privacy-preserving MapReduce search called *Baseline*. Baseline search consists of an empty map phase, where mappers simply scan over InputSplits and compare each word of the InputSplit with a predetermined one, but do not generate any key-value pairs. Only at the end of the map phase, a single intermediate key-value pair per mapper (e.g., “found”) is sent to reducers. 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. [17, 34] into account. Note that linear scanning through the entire InputSplit is mandatory, as we assume our data to be unordered and unsorted.

For the private information retrieval algorithm, we set $m = 400$ as suggested by Trostle and Parrish [37] for good security. 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 [20]. As individual DNS entries occurred way less than 2^{16} times per day, we reserved $|\gamma| = 2$ Bytes and truncated entries longer than 30 Byte down to the last 30 Byte. Because the size of input $|w_i|+|\gamma_{w_i}|$ is less than E’s blocksize (using standard padding for w_i), *concatenation* provides an unambiguous pairing.

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 $P_{\text{err}} < 0.01$, we derived t and q. Table 1 summarizes parameters (t, q) computed for each file individually. Compared to q, we observed that parameter t has a much higher impact on P_{err}, 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).

6.2 Results

Computational overhead at the cloud is low as indicated by Figure 1 (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. 1 shows the average. For each file, Fig. 1 shows two stacked boxes, respectively: the first one for 96 MByte and the second one for 120 MByte

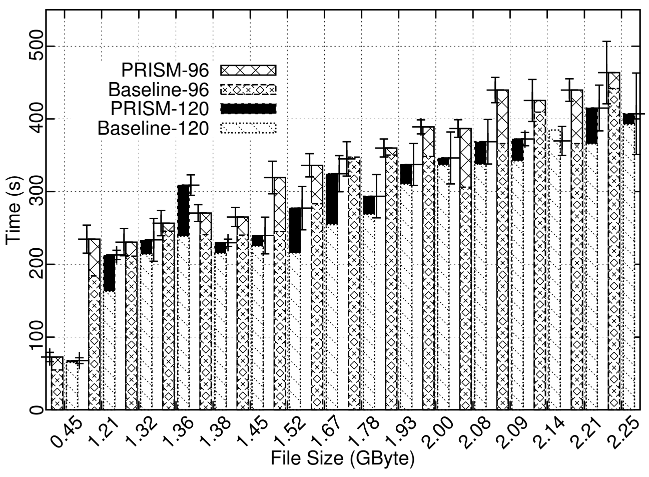

Fig. 1. Wall clock timings for PRISM and Baseline, with $S_{\text{InputSplit}} = 96$ and 120 MByte

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. 1 also shows 95% confidence intervals drawn right next to each box.

Timings shown in Fig. 1 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 the real-world, wall clock time reflects the time a cloud, e.g., Amazon [2], would charge a user $\mathcal{U}$. 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. This overhead is mostly computational overhead, as there is no difference in disk access between Baseline and PRISM and network volume increases only little by sending slightly larger values during "Reduce". These results do not only show the feasibility of PRISM in practice, but also demonstrate the low overhead implied by PRISM over the non-privacy preserving MapReduce job. We claim that a performance optimized (not based on Java BigInteger) implementation improves performance significantly and furthermore reduces overhead.

The simple increase from 96 MByte InputSplit size to 120 MByte InputSplit size has reduced wall clock times for MapReduce jobs 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 indicates 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. [34], so a trade-off has to be found. MapReduce configuration optimizations are, however, out of scope.

To better understand the cloud's computational overhead, we also measured the **computation time for a PRISM mapper**. On a single CPU, execution of an isolated PRISM map function on a single InputSplit is ca. 9 times slower than Baseline (9.3 for 96 MByte and 9.1 for 120 MByte, 95% confidence interval of ± 0.1). While this seems to be a lot, we remark that 1.) this map overhead is constant for an InputSplit and does not depend on or scale with the total size of the data, 2.) there is a lot of potential to improve our map implementation, 3.) this overhead is obviously amortized by other MapReduce aspects such as the Reduce phase and also disk latency, network overhead etc., and 4.) a user is charged for the total system time, i.e., the wall clock time.

Computational overhead at the User is also low in PRISM: per file, the preparation of, e.g., 2^{12} α values for the underlying PIR scheme is barely measurable (≈ 200 ms) on a PC with 2.5 GHz CPU. 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 Java BigInteger multiplications with modulo have to be performed. For our examples with $q \leq 100$, this was not measurable at less than 1 ms.

Memory consumption for $\mathcal{U}$ is, on the one hand, constant; $\mathcal{U}$ only stores the 256 bit AES key K. On the other hand however, the cloud user $\mathcal{U}$'s memory consumption scales linearly with $O(\Sigma)$, i.e., the number of *different* words. This is due to the construction of our stateful cipher that stores counters γ in a hashtable. In our straightforward implementation with Java's standard Hashtable, memory consumption of this hashtable was 548 MByte for the largest log file. While this is certainly a lot of RAM, we conjecture this to be available on PC hardware – moreover, as there is a large potential for performance tuning with such data structures.

Communication overhead for PRISM is dominated by the underlying PIR scheme. $\mathcal{U}$ sends, besides .class files once, only the t α values per file to the cloud. For example, with $t = 2^{12}$ and $m = 400$ Bit, this computes to 200 KByte per file. The response from the cloud is, for each round, t values of size m. The most expensive configuration in terms of communication in our experiments has been $t = 2^{10}, q = 100$; this results in ≈ 5 MByte communication overhead. Note that communication complexity in the underlying PIR scheme by Trostle and Parrish [37] 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 $\epsilon > 0$ [29]. Those optimizations as well as amortization techniques discussed by Ostrovsky and Skeith [33] are out of scope.

In conclusion, PRISM is very lightweight for a user using standard PC hardware.

Discussion: On a larger cluster in a more professional environment (hundreds or even thousands of CPUs [25]), all files will be processed in parallel. As shown in Fig. 1, total time for the 2 GByte file is ≈ 350 s. However, already ≈ 340 s are required by MapReduce just to "scan" through the various InputSplits, see Baseline. Such inefficiency with non-optimal configurations has been observed before, and our results are along the lines of Pavlo et al. [34]. Here, a "grep"-like MapReduce job on 1 TByte of data took $\approx 1,500$ s on 50 CPUs which would be ≈ 20 times faster than our Baseline. However, Pavlo et al. [34] 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 [27].

7 Related Work

Private Information Retrieval: Private Information Retrieval (and similarly oblivious transfer and oblivious RAM) has received a lot of attention [9, 13, 19, 22, 29, 32, 33, 35]. In PIR, a user retrieves a specific data from a database. The only "privacy" goal in PIR is access privacy whereby the server should not discover which data a user is interested in. Note that PIR does not ensure privacy of data in the database. PRISM, however, focuses on *searching* for a word and uses PIR only as a tool.

Searchable Encryption: With searching on encrypted data techniques [6], 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 [4, 6–8, 10, 16, 21, 31, 36], the result ("found" or "not found") originating from a query is known to the adversary; therefore as opposed to PRISM, standard searchable encryption techniques do not ensure *query privacy*. Moreover, existing mechanisms *cannot* be easily 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, today's solution do not support the *combination* (*aggregation*) of results (as in a reduce phase). To conclude, PRISM not only ensures both *storage privacy* and *query privacy*, but also enables the aggregation of results originating from intermediate parallelized operations.

8 Conclusion

PRISM is the first privacy-preserving search scheme suited for cloud computing. That is, PRISM provides storage and query privacy while introducing only limited overhead. PRISM is specifically designed to leverage parallelism and efficiency of the MapReduce paradigm. Moreover, PRISM is compatible with any standard MapReduce-based cloud infrastructure (such as Amazon's), and does not require modifications to the underlying system. Thanks to this compatibility, PRISM has been efficiently implemented on an experiemental cloud computing environment using Hadoop MapReduce. Besides a throughout analysis, performance of PRISM has been evaluated on that environment through search operations in DNS logs provided by an ISP. PRISM's overhead over non-privacy-preserving search is only 11% on average, acertaining its efficiency.

References

[1] PRISM source code (2012), `http://www.ccs.neu.edu/~blass/prism.tgz`
[2] Amazon. Elastic mapreduce (2010), `http://aws.amazon.com/elasticmapreduce/`
[3] Apache. Hadoop (2010), `http://hadoop.apache.org/`
[4] Bellovin, S.M., Cheswick, W.R.: Privacy-enhanced searches using encrypted Bloom filters (2007), `http://mice.cs.columbia.edu/getTechreport.php?techreportID=483`
[5] Bilge, L., Kirda, E., Krügel, C., Balduzzi, M.: Exposure: Finding malicious domains using passive dns analysis. In: Proceedings of 18th Annual Network and Distributed System Security Symposium, San Diego, USA, pp. 195–211 (2011) ISBN 1891562320

[6] Boneh, D., Di Crescenzo, G., Ostrovsky, R., Persiano, G.: Public Key Encryption with Keyword Search. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 506–522. Springer, Heidelberg (2004)

[7] Boneh, D., Kushilevitz, E., Ostrovsky, R., Skeith III, W.E.: Public Key Encryption That Allows PIR Queries. In: Menezes, A. (ed.) CRYPTO 2007. LNCS, vol. 4622, pp. 50–67. Springer, Heidelberg (2007)

[8] Brassard, G., Crépeau, C., Robert, J.M.: All-or-Nothing Disclosure of Secrets. In: Odlyzko, A.M. (ed.) CRYPTO 1986. LNCS, vol. 263, pp. 234–238. Springer, Heidelberg (1987)

[9] Cachin, C., Micali, S., Stadler, M.A.: Computationally Private Information Retrieval with Polylogarithmic Communication. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, pp. 402–412. Springer, Heidelberg (1999)

[10] Chang, Y.-C., Mitzenmacher, M.: Privacy Preserving Keyword Searches on Remote Encrypted Data. In: Ioannidis, J., Keromytis, A.D., Yung, M. (eds.) ACNS 2005. LNCS, vol. 3531, pp. 442–455. Springer, Heidelberg (2005)

[11] Chief Information Officer's Council. Proposed security assessment & authorization for U.S. government cloud computing (2010), `http://www.digitalgovernment.com/media/Knowledge-Centers/asset_upload_file652_2491.pdf`

[12] Chief Information Officer's Council. Privacy recommendations for the use of cloud computing by federal departments and agencies (2010), `http://www.cio.gov/`

[13] Chor, B., Goldreich, O., Kushilevitz, E., Sudan, M.: Private information retrieval. In: Proceedings of Symposium on Foundations of Computer Science, Milwaukee, USA, pp. 41–51 (1995)

[14] Cloud Security Alliance. Security guidance for critical areas of focus in cloud computing (2009), `https://cloudsecurityalliance.org/guidance/csaguide.v2.1.pdf`

[15] Cloud Security Alliance. Top cloud computing threats (2010), `https://cloudsecurityalliance.org/topthreats/csathreats.v1.0.pdf`

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

[17] Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: Proceedings of OSDI, San Francisco, USA, pp. 137–150 (2004)

[18] EU, Eu information management instruments (2010), `http://europa.eu/`

[19] Gertner, Y., Ishai, Y., Kushilevitz, E.: Protecting data privacy in private information retrieval. In: Proceedings of Symposium on Theory of Computing, Dallas, USA, pp. 151–160 (1998) ISBN 0-89791-962-9

[20] GNU, The gnu crypto project (2011), `http://www.gnu.org/software/`

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

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

[23] Goldwasser, S., Micali, S.: Probabilistic encryption. Journal of Computer and System Sciences 28(2), 270–299 (1984) ISSN 0022-0000

[24] Google. Google apps for government (2010), `http://googleenterprise.blogspot.com/2010/07/google-apps-for-government.html`

[25] Hadoop. Powered by hadoop, list of applications using hadoop mapreduce (2011), `http://wiki.apache.org/hadoop/PoweredBy`

[26] Hall, C., Goldberg, I., Schneier, B.: Reaction Attacks against Several Public-Key Cryptosystem. In: Varadharajan, V., Mu, Y. (eds.) ICICS 1999. LNCS, vol. 1726, pp. 2–12. Springer, Heidelberg (1999)

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

[28] Katz, J., Lindell, Y.: Introduction to modern cryptography. Chapman & Hall/CRC (2008) ISBN 978-1-58488-551-1

[29] Kushilevitz, E., Ostrovsky, R.: Replication is not needed: single database, computationally-private information retrieval. In: Proceedings of Symposium on Foundations of Computer Science, Miami Beach, USA, pp. 364–373 (1997)

[30] McCullagh, D.: Fbi wants records kept of web sites visited (2010), `http://news.cnet.com/8301-13578_3-10448060-38.html`

[31] Ogata, W., Kurosawa, K.: Oblivious keyword search. Journal of Complexity – Special Issue on Coding and Cryptography 20, 356–371 (2004) ISSN 0885-064X

[32] Ostrovsky, R., Skeith III, W.E.: Private Searching on Streaming Data. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 223–240. Springer, Heidelberg (2005)

[33] Ostrovsky, R., Skeith III, W.E.: A Survey of Single-Database Private Information Retrieval: Techniques and Applications. In: Okamoto, T., Wang, X. (eds.) PKC 2007. LNCS, vol. 4450, pp. 393–411. Springer, Heidelberg (2007)

[34] Pavlo, A., Paulson, E., Rasin, A., Abadi, D.J., DeWitt, D.J., Madden, S., Stonebraker, M.: A comparison of approaches to large-scale data analysis. In: Proceedings of International Conference on Management of Data, Rhode Island, USA, pp. 165–178 (2009)

[35] Sion, R., Carbunar, B.: On the computational practicality of private information retrieval. In: Proceedings of Network and Distributed Systems Security Symposium, San Diego, USA, pp. 1–10 (2007)

[36] Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: Proceedings of Symposium on Security and Privacy, Berkeley, USA, pp. 44–55 (2000)

[37] Trostle, J., Parrish, A.: Efficient Computationally Private Information Retrieval from Anonymity or Trapdoor Groups. In: Burmester, M., Tsudik, G., Magliveras, S., Ilić, I. (eds.) ISC 2010. LNCS, vol. 6531, pp. 114–128. Springer, Heidelberg (2011)