EPiC: Efficient Privacy-Preserving Counting for MapReduce

Abstract. In the face of an untrusted cloud infrastructure, outsourced data needs to be protected. We present EPiC, a practical protocol for the privacy-preserving evaluation of a fundamental operation on data sets: frequency counting. In an encrypted outsourced data set, a cloud user can specify a pattern, and the cloud will count the number of occurrences of this pattern in an oblivious manner. A pattern is expressed as a Boolean formula on the fields of data records and can specify values counting, value comparison, range counting, and conjunctions/disjunctions of field values. We show how a general pattern, defined by a Boolean formula, is arithmetized into a multivariate polynomial and used in EPiC. To increase the performance of the system, we introduce a new privacy-preserving encoding with "somewhat homomorphic" properties. The encoding's security is based on previous work on the Hidden Modular Group assumption, and it is highly efficient in our particular counting scenario. Besides a formal analysis where we prove EPiC's privacy, we also present implementation and evaluation results. We specifically target Google's prominent MapReduce paradigm as offered by major cloud providers. Our evaluation performed both locally and in Amazon's public cloud with up to 1 TByte data sets shows only a modest overhead of 20% compared to non-private counting, attesting to EPiC's efficiency.

1 Introduction

Cloud computing is a promising technology for large enterprises and even governmental organizations. Major cloud computing providers such as Amazon and Google offer users to outsource their data and computation. The main advantage for users lies in the clouds' flexible cost model: users are only charged by use, e.g., total amount of storage or CPU time used. In addition, clouds "elastic" services allow the users to efficiently scale resources to satisfy dynamic load. The appeal and success of outsourcing data and operating on outsourced data is exemplified by Google's prominent MapReduce API [7]. MapReduce is offered by major public cloud providers today, such as Amazon, Google, IBM or Microsoft. MapReduce is typically used for analysis operations on huge amounts of (outsourced) data, e.g., scanning through data and finding patterns, counting occurrences of specific patterns, and other statistics [11].

While the idea of moving data and computation to a (public) cloud for cost savings is appealing, trusting the cloud to store and protect data against *adversaries* is a serious concern for users. The encryption of data is a viable privacy protection mechanism, but it renders subsequent operations on encrypted data a challenging problem. To address this problem, *Fully Homomorphic Encryption* (FHE) techniques have been investigated, cf. Gentry [9] or see Vaikuntanathan [19] for an overview. FHE guarantees that the cloud neither learns details about the stored data nor about the results. However, today's FHE schemes are still overly inefficient [6, 10, 14, 20], and a deployment in a real-world cloud would outweigh any cost advantage offered by the cloud. Furthermore, any solution running in a real-world cloud needs to be tailored to the specifics of the cloud computing paradigm, e.g., MapReduce.

This paper presents an efficient, practical, yet privacy-preserving protocol for a fundamental data analysis primitive in MapReduce: *counting occurrences* of patterns [11]. In an outsourced data set comprising a large number of encrypted data records, EPiC, "Efficient PrIvacy-preserving Counting for MapReduce", allows the cloud user to specify a pattern, and the cloud will count the number of occurrences of this pattern (and therefore histograms) in the stored ciphertexts without revealing the pattern and how often it occurs. A pattern is expressed as a Boolean formula on countable fields of data records and can specify a specific field value, a value comparison, a range of field values, and more complex forms of conjunctions/disjunctions among sub-patterns. For example, in an outsourced data set of patient health records, a pattern could be $age \in [50, 70]$ and $(diabetes = 1$ or $hypertension = 1)$. The main idea of EPiC is to transform the problem of privacy-preserving pattern counting into a summation of polynomial evaluations. Our work is inspired by Lauter et al. [13] to use *somewhat homomorphic* encryption to address specific privacy-preserving operations. In EPiC, we extend a previous work on cPIR protocols [18] to design a new "encoding" mechanisms that exhibits somewhat homomorphic properties. While we call our encoding encryption in the rest of this paper, we stress that our encryption does not provide traditional IND-CPA security, but only weaker properties suited to the context we target in this paper, i.e., the summation of polynomial evaluations. In return, our "encryption" is particularly efficient in this context. We also show how a general pattern, defined by a Boolean formula, is arithmetized into a multivariate polynomial over $GF(2)$, optimizing for efficiency. In conclusion, the contributions of this paper are:

- EPiC, a new protocol to enable privacy-preserving pattern counting in MapReduce clouds. EPiC reduces the problem of counting occurrences of a Boolean pattern to the summation of a multivariate polynomial evaluated on encrypted data.
- A new, practical "somewhat homomorphic" encoding/encryption scheme specifically addressing secure counting in a highly efficient manner.
- An implementation of EPiC and its encryption mechanism together with an extensive evaluation in a realistic setting. The source code is available for download [2].

2 Problem Statement

Overview: We will use an example application to motivate our work. Along the lines of recent reports [17], imagine a hospital scenario where patient records are managed electronically. To reduce cost and grant access to, e.g., other hospitals and external doctors, the hospital refrains from investing into an own, local data center, but plans to outsource patient records to a public cloud. Regulatory matters require the privacy-protection of sensitive medical information, so outsourced data has to be encrypted. However, besides uploading, retrieving or editing patient records performed by multiple entities (hospitals, doctors etc.), one entity eventually wants to collect some statistics on the outsourced patient records without the necessity of downloading all of them.

2.1 Cloud Counting

More specifically, we assume that each patient record R includes one or more countable fields $R.c$ containing some patterns. A user (e.g., doctor) $\mathcal{U}$ wants to extract the frequency of occurrence of pattern χ, e.g., how many patients have $R.disease = \chi$. Due to the large amount of data, downloading each patient record is prohibitive, and the counting should be performed by the cloud. While encryption of data, access control, and key management in a multi-user cloud environment are clearly important topics, we focus on the problem of a-posteriori extracting information out of the outsourced data in a privacy-preserving manner. The cloud must neither learn details about the stored data, nor any information about the counting, what is counted, the count itself, etc. Instead, the cloud processes $\mathcal{U}$'s counting queries "obliviously". We will now first specify the general setup of counting schemes for public clouds and then formally define privacy requirements. Note that throughout this paper, we will assume the countable fields to be non-negative integer fields. Besides, records may contain non-countable data, e.g., pictures or doctors' notes, that can be IND-CPA (AES-CBC) encrypted – Therewith, it is of no importance for privacy defined below.

Definition 1 (Cloud Counting). *Let $\mathcal{R}$ denote a sequence of records $\mathcal{R} := \{R_1, \ldots, R_n\}$. Besides some non-countable data, each record R_i contains m different countable fields. The k-th countable field of the i-th record, denoted as $R_{i,j}, 1 \leq k \leq m$, can take values $R_{i,k} \in \mathcal{D}_k = \{0, 1, \ldots, |\mathcal{D}_k| - 1\}$, where $\mathcal{D}_k$ denotes the domain of the k-th field with size[1] $|\mathcal{D}_k|$. For the "multi-domain" of m countable fields we write $\mathcal{D} = \mathcal{D}_1 \times \cdots \times \mathcal{D}_m$. A privacy-preserving counting scheme comprises the following probabilistic polynomial time algorithms:*

1. KEYGEN(κ)*: using a security parameter κ, outputs a secret key $\mathcal{S}$.*
2. ENCRYPT($\mathcal{S}, \mathcal{R}$)*: uses secret key $\mathcal{S}$ to encrypt the sequence of records $\mathcal{R}$. The output is a sequence of encryptions of records $\mathcal{E} := \{E_{R_1}, \ldots, E_{R_n}\}$, where E_{R_i} denotes the encryption of record R_i.*
3. UPLOAD($\mathcal{E}$)*: uploads the sequence of encryptions $\mathcal{E}$ to the cloud.*
4. PREPAREQUERY($\mathcal{S}, \chi$)*: this algorithms generates an encrypted query Q out of secret $\mathcal{S}$ and the multiple-field pattern $\chi \in \mathcal{D}$.*
5. PROCESSQUERY($Q, \mathcal{E}$)*: uses an encrypted query Q, the sequence of ciphertexts $\mathcal{E}$, and outputs a result E_Σ. This algorithm performs the actual counting.*
6. DECODE($\mathcal{S}, E_\Sigma$)*: takes secret $\mathcal{S}$ and E_Σ to output a final result, the occurrences Σ (the "count") of the specified pattern in $\mathcal{R}$.*

According to this definition, cloud user $\mathcal{U}$ encrypts the sequence of records and uploads them into the cloud. If $\mathcal{U}$ wants to know the number of occurrences of χ in the records, he prepares a query Q, which is – as we will see later – simply a fixed-length sequence of encrypted values. U then sends Q to the cloud, and the cloud processes Q. Finally, the cloud sends a result E_Σ back to U who can decrypt this result and learn the number Σ of occurrences of pattern χ, i.e., the count.

2.2 Privacy

In the face of an untrusted cloud infrastructure, cloud user $\mathcal{U}$ wants to perform counting in a privacy-preserving manner. Intuitively, the data stored at the cloud as well as the counting operations must be protected against a curious cloud. Informally, we demand

[1] Domain size $|\mathcal{D}_k|$ indicates the number of different values a field can take.

1) *storage privacy*, where the cloud does not learn anything about stored data, and 2) *counting privacy*, where the cloud does not learn anything about queries and query results. The cloud, which we now call "adversary" $\mathcal{A}$, should only learn "trivial" privacy properties like the total size of outsourced data, the total number of patient records or the number of counting operations performed for $\mathcal{U}$. We formalize privacy for counting using a game-based setup. In the following, $\epsilon(\kappa)$ denotes a negligible function in the security parameter κ.

Definition 2 (Bit mapping). *Let $\mathcal{R} = \{R_1, \ldots, R_n\}$ be a set of records, and $R_{i,k} \in \{0,1\}^*$ the k-th field of record R_i. Let $\chi, \Sigma \in \{0,1\}^*$ be bit string representations of a pattern and a count. For $X \in \{R_{i,k}, \chi, \Sigma\}$, $\mathrm{bit}(j, X)$ denotes the j-th bit of X.*

Definition 3 (Storage privacy). *A challenger generates two same-size same-field-types sets of records $\mathcal{R}, \mathcal{R}'$ and two patterns $\chi, \chi' \in \mathcal{D}$. The challenger then uses* ENCRYPT *and* PREPAREQUERY *to compute the encrypted sets of records $\mathcal{E}, \mathcal{E}'$ and two encrypted counting queries Q, Q' corresponding to two patterns χ, χ'. Using* PROCESSQUERY, *he evaluates $\mathcal{E}$ with Q, and $\mathcal{E}'$ with Q' to get encrypted results E_Σ, E'_Σ. The challenger sends $I := \{\mathcal{E}, \mathcal{E}', Q, Q', E_\Sigma, E'_\Sigma\}$ to adversary $\mathcal{A}$. For any patterns χ, χ', any X, X' such that either $X \in \{\{R_{i,k}\}\}$ and $X' \in \{\{R_{i,k}\}\}$ or $X = \chi$ and $X' = \chi'$ or $X = \Sigma$ and $X' = \Sigma'$, and for any $b = \mathrm{bit}(j, X)$ and $b' = \mathrm{bit}(j', X')$, the adversary $\mathcal{A}$ outputs 1, if she guesses $b = b'$, and 0 otherwise. A protocol preserves storage privacy,* iff *for any probabilistic polynomial time (PPT) algorithm $\mathcal{A}$, the probability of correct output is not higher than a random guess. That is,* $\left|\Pr\left[\mathcal{A}(I) = 1 | b = b'\right] - \frac{1}{2}\right| \leq \epsilon(\kappa)$ and $\left|\Pr\left[\mathcal{A}(I) = 0 | b \neq b'\right] - \frac{1}{2}\right| \leq \epsilon(\kappa)$.

Definition 4 (Counting privacy). *A challenger generates two same-size same-field-types sets of records $\mathcal{R}, \mathcal{R}'$, and two patterns χ, χ', uses* ENCRYPT, PREPAREQUERY, *and* PROCESSQUERY, *and sends encrypted $I := \{\mathcal{E}, \mathcal{E}', Q, Q', E_\Sigma, E'_\Sigma\}$, to $\mathcal{A}$. Now, $\mathcal{A}$ outputs 1, if $\chi = \chi'$, and 0 otherwise. A protocol preserves counting privacy,* iff *for any PPT algorithm $\mathcal{A}$ the probability of correct output is not better than a random guess: $\left|\Pr\left[\mathcal{A}(I) = 1 | \chi = \chi'\right] - \frac{1}{2}\right| \leq \epsilon(\kappa)$ and $\left|\Pr\left[\mathcal{A}(I) = 0 | \chi \neq \chi'\right] - \frac{1}{2}\right| \leq \epsilon(\kappa)$.*

Similar to traditional indistinguishability, *storage privacy* and *counting privacy* captures the intuition that, by storing data and counting, the cloud should not learn anything about the content it stores. In addition, the cloud should not learn anything about the counting performed, such as which pattern is counted, whether a pattern is counted twice or what the resulting count is.

2.3 MapReduce

The efficiency of counting relies on the performance of PROCESSQUERY which involves processing huge amounts of data in the cloud. Cloud computing usually processes data in parallel via multiple nodes in the cloud data center based on some computation paradigm. For efficiency, PROCESSQUERY has to take the specifics of that computation into account. One of the most widespread, frequently used framework for distributed computation that is offered by major cloud providers today is MapReduce. In the following, we will give a very compressed overview about MapReduce, only to understand EPiC. For more details, refer to Dean and Ghemawat [7].

EPiC's counting "job" runs in two phases. First, in the "mapping" phase, *Mapper* nodes scan data through *InputSplits* (data pieces split automatically by MapReduce framework) and evaluate the counting's *map* function on the data. These operations are performed by all Mappers in parallel. The outputs of each *map* function are sent to one *Reducer* node, which, in the "reducing" phase, aggregates them and produces a final output that is sent back to the user. This setup takes advantage of the parallel nature of a cloud data center and allows for scalability and elasticity.

3 EPiC Protocol

To motivate the need for a more sophisticated protocol like EPiC, we briefly discuss why possible straightforward solutions do not work in our particular application scenario.
Precomputed Counters: One could imagine that the cloud user, in the purpose of counting a value χ_k in a single countable field $\mathcal{D}_k$, simply stores encrypted counters for each possible value of χ_k in domain $\mathcal{D}_k$ in the cloud. Each time records are added, removed or updated, the cloud user updates the encrypted counters. However, this approach does not scale very well in our scenario where multiple cloud users (different "doctors") perform updates and add or modify records. An expensive user side locking mechanism would be required to ensure consistency of the encrypted counters. Moreover, in the case of complex queries involving multiple fields, all possible combinations of counters need to be updated by users involving a lot of user side computation.
Per-Record Counters ("Voting"): Alternatively and similar to a naive voting scheme, each encrypted record stored in the cloud could be augmented with an encrypted "voting" field containing $|\mathcal{D}_k|$ subsets, each of $\log_2 n$ bits. If a record's countable value in field $\mathcal{D}_k$ matches the value corresponding to a subset, then the according subset is set to 1. To find the count, the cloud sums the encrypted voting fields (using additive homomorphic encryption) for all records. Again, such an approach requires heavy locking mechanism and recomputation of counters for each operation of adding, removing, or modifying a record. In conclusion, these straightforward solutions require heavy user-side computation and do not provide efficient, practical, and flexible solutions for multi-user, multiple field data sets.

3.1 EPiC Overview

For ease of understanding, we *initially* introduce EPiC for the simpler case of counting on only a single countable field $\mathcal{D}_k$ in a multiple countable fields data set where values are in GF(q). Subsequently, we extend EPiC to support counting on Boolean combinations of multiple countable fields $\mathcal{D}_1, \ldots, \mathcal{D}_m$ over GF(q). Finally, for performance improvement, we further optimize our mechanisms by considering conversion of (generic) finite fields GF(q) into binary finite fields GF(2).

EPiC's main rationale is to perform the counting in the cloud by evaluating an *indicator polynomial* $P_\chi(\cdot)$, as query Q, specific to the pattern χ the cloud user $\mathcal{U}$ is interested in. Conceptually, the cloud evaluates $P_\chi(\cdot)$ on the countable fields' values of each record. The outcome of all individual polynomial evaluations is a (large) set of values of either "1" or "0". The cloud now adds these values and sends the sum back to $\mathcal{U}$, who learns the number of occurrences of χ in the investigated set of records.

3.2 Counting on a single field

Without loss of generality, we assume a user $\mathcal{U}$ wishes to count occurrences of χ in the first field $\mathcal{D}_1$ in an oblivious manner. The idea is to prepare a univariate indicator polynomial $P_\chi(x)$ such that $P_\chi(x) = \begin{cases} 1, \text{ if } x = \chi \\ 0, \text{ otherwise} \end{cases}$, and scan through the data set $\mathcal{R} = \{R_1, \ldots, R_n\}$ of all records to compute the sum $\sum_{i=1}^{n} P_\chi(R_{i,1})$. The result is the number of occurrences of χ in the first field in the data set. The idea for generating $P_\chi(x)$ is to construct the polynomial in the Lagrange interpolation form $P_\chi(x) := \sum_{j=0}^{|\mathcal{D}_1|-1} a_j \cdot x^j := \prod_{\alpha \in \mathcal{D}_1, \alpha \neq \chi} \frac{x-\alpha}{\chi-\alpha}$. The polynomial $P_\chi(x)$ is of degree $|\mathcal{D}_1| - 1$, and its coefficients a_j are uniquely determined.

Encrypted polynomial: In EPiC, each countable value $R_{i,k}$ is encrypted to $E_{R_{i,k}}$. The above indicator polynomial based counting method for plaintext values can be applied in a similar manner. User $\mathcal{U}$ prepares the indicator polynomial based on plaintext χ, but $\mathcal{U}$ encrypts coefficients a_j to E_{a_j} before sending them to the cloud, which now computes the encrypted sum $E_\Sigma := \sum_{i=1}^{n} P_\chi(E_{R_{i,1}}) = \sum_{i=1}^{n} \sum_{j=0}^{|\mathcal{D}_1|-1} E_{a_j} \cdot (E_{R_{i,1}})^j$. Note that the polynomial *coefficients* are encrypted (and potentially large), but the polynomial *degree* remains $|\mathcal{D}_1| - 1$. In order for the cloud to compute E_Σ and user $\mathcal{U}$ to decrypt it later, additively and multiplicatively homomorphic properties are required for the encryption, which we describe in Section 3.5. As a final step, $\mathcal{U}$ simply receives back E_Σ and only decrypts the count $\sigma := \textsc{Dec}(E_\Sigma) = P_\chi(x)$. This does not require high computational costs at the user, suiting the cloud computing paradigm well.

Cloud computation cost: Our counting technique above requires $n \cdot |\mathcal{D}_1|$ additions, $n \cdot |\mathcal{D}_1|$ multiplications, and $n \cdot (|\mathcal{D}_1| - 1)$ exponentiations. We can improve efficiency by rearranging the order of computations:

$$E_\Sigma := \sum_{i=1}^{n} P_\chi(E_{R_{i,1}}) = \sum_{i=1}^{n} \sum_{j=0}^{|\mathcal{D}_1|-1} E_{a_j} \cdot (E_{R_{i,1}})^j = \sum_{j=0}^{|\mathcal{D}_1|-1} (E_{a_j} \cdot \sum_{i=1}^{n} (E_{R_{i,1}})^j). \tag{1}$$

Therewith, the number of multiplications is reduced to $|\mathcal{D}_1|$. We will apply this technique also to multiple field counting later, significantly improving performance. We also note that in the case of a binary domain ($|\mathcal{D}_1| = 2$), there are no exponentiations. This observation motivates our optimization described later in Section 3.4.

Oblivious counting: Our indicator polynomial based counting method is oblivious: *first*, the query is submitted to the cloud as a sequence of encrypted coefficients of the indicator polynomial; *second*, no matter what query is made, exactly $|\mathcal{D}_1|$ coefficients (including 0-coefficients) are sent, thus preventing the cloud to infer query information based on the query size.

3.3 Counting patterns defined by a Boolean formula

We now extend the indicator polynomial based counting technique towards a general solution for counting patterns defined by any Boolean combination of *multiple* fields in the data set. The key technique for defining an indicator polynomial corresponding to an arbitrary Boolean expression among multiple fields is to transform Boolean operations to arithmetic operations, which is similar to *arithmetization*, cf. Babai and Fortnow [4] or Shamir [15].

Conjunctive counting: Assume cloud user $\mathcal{U}$ is interested in counting the number of records that have their m countable fields set to the pattern $\chi = (\chi_1, \ldots, \chi_m)$. Here, χ_k, $1 \leq k \leq m$, denotes the queried value in the k-th field. Let $\varphi = (x_1 = \chi_1 \wedge \ldots \wedge x_m = \chi_m)$ be the conjunction among m fields in the data set. User $\mathcal{U}$ can now construct $P_\varphi(\mathbf{x}) = \prod_{k=1}^{m} P_{\chi_k}(x_k)$, where $\mathbf{x} = (x_1, \ldots, x_m)$ denotes the variables in the multivariate polynomial $P_\varphi(\mathbf{x})$, and $P_{\chi_k}(x_k)$ is the univariate indicator polynomial (as defined in Section 3.2) for counting χ_k in the k-th field. Therewith, $P_\varphi(\mathbf{x})$ yields 1 only when χ is matched. Note that the size of the multi-domain $\mathcal{D}$ is $|\mathcal{D}| = \prod_{k=1}^{m} |\mathcal{D}_k|$, and the degree of $P_\varphi(\mathbf{x})$ is $\sum_{k=1}^{m} (|\mathcal{D}_k| - 1)$.

Disjunctive counting: Assume the data set has 2 countable fields, and $\mathcal{U}$'s objective is to count the number of records that have value χ_1 in $\mathcal{D}_1$ or value χ_2 in $\mathcal{D}_2$. The multivariate indicator polynomial for this disjunction is $P_{\chi_1 \vee \chi_2}(\mathbf{x}) = P_{\chi_1}(x_1) + P_{\chi_2}(x_2) - P_{\chi_1 \wedge \chi_2}(\mathbf{x})$, where $P_{\chi_1}(x_1), P_{\chi_2}(x_2)$ are univariate indicator polynomials for $\mathcal{D}_1, \mathcal{D}_2$, respectively, and $P_{\chi_1 \wedge \chi_2}(\mathbf{x})$ is a multivariate indicator polynomial for conjunctive counting between $\mathcal{D}_1$ and $\mathcal{D}_2$. This method can be easily generalized to design counting query for disjunctions of m fields.

Complement counting: $\mathcal{U}$ can count records that do not satisfy a condition among fields by "flipping" the satisfying indicator polynomial: $P_{\neg\varphi}(\mathbf{x}) = 1 - P_\varphi(\mathbf{x})$.

Integer range counting: Assume $\mathcal{U}$ wants to count records having a field $\mathcal{D}_k$ lying in an integer range $[a, b]$, i.e., $\varphi = (x_k = a \vee x_k = a + 1 \vee \ldots \vee x_k = b)$. Based on disjunctive constructing method, we have $P_{[a,b]}(x_k) = P_a(x_k) + P_{a+1}(x_k) + \ldots + P_b(x_k) - P_{a \wedge a+1} - \ldots$; Since $(x_k = u)$ and $(x_k = v)$ are exclusive disjunctions for any $u \neq v \in [a, b]$, $P_{[a,b]}(x_k)$ reduces to $P_{[a,b]}(x_k) = \sum_{\chi_k = a}^{b} P_{\chi_k}(x_k)$.

Integer comparison counting: Queries involving integer comparisons can be constructed based on integer range counting, e.g., $P_{\chi_k \leq a}(x_k) = P_{[0,a]}(x_k)$, or $P_{\chi_k > a}(x_k) = P_{[a+1, |\mathcal{D}_k| - 1]}(x_k)$.

Privacy: Although the user-defined queries are different in construction, the encrypted queries Q always have exactly $|\mathcal{D}| = \prod_{k=1}^{m} |\mathcal{D}_k|$ encrypted coefficients as we include zero coefficients also. As mentioned in Section 3.2, this prevents the cloud to differentiate queries based on query sizes.

Efficiency: The user-side computation involving constructing the query's coefficients is carried on plain-text before encryption, hence it introduces much lower computation cost compared to the computation burden on the cloud. To improve the user-side performance, one could apply optimizing techniques for reducing complex expressions, but this is out of scope of our paper. To improve the cloud's performance, we apply the method of rearranging order of computations as in equation (1) for the sequence $E(R_i) = (E_{R_{i,1}}, \ldots, E_{R_{i,m}})$ of encrypted fields and coefficients $a_{\mathbf{j}}$ with $\mathbf{j} = (j_1, \ldots, j_m) \in \mathcal{D}$ corresponding to multivariate monomials $x_1^{j_1} \cdot x_2^{j_2} \cdots x_m^{j_m}$:

$$E_\Sigma = \sum_{i=1}^{n} P_\chi(E(R_i)) = \sum_{\mathbf{j} \in \mathcal{D}} (E_{a_{\mathbf{j}}} \cdot \sum_{i=1}^{n} \prod_{k=1}^{m} (E_{R_{i,k}})^{j_k}). \tag{2}$$

3.4 Optimization through arithmetization in $GF(2)$

EPiC's efficiency relies on the computations performed by the cloud. As discussed in Section 3.2, there are *no exponentiations* required for counting on a binary field. Consequently, we optimize EPiC by converting generic (non-binary) fields into multiple binary fields, thereby avoiding costly exponentiations. Note that as the conversion preserves Boolean expression output, results shown in Section 3.3 still hold, and protocol details discussed later in Section 3.6 remain unchanged.

Our idea is to store every generic field $\mathcal{D}_k$ as separate binary fields $\mathcal{D}_{k,1}, \mathcal{D}_{k,2}, \ldots, \mathcal{D}_{k,\|\mathcal{D}_k\|}$. [2] Therefore, m generic fields $\mathcal{D}_1, \ldots, \mathcal{D}_m$ become $\sum_{k=1}^{m} \|\mathcal{D}_k\|$ binary fields $\mathcal{D}_{1,1}, \ldots, \mathcal{D}_{1,\|\mathcal{D}_1\|}, \ldots, \mathcal{D}_{m,1}, \ldots, \mathcal{D}_{m,\|\mathcal{D}_m\|}$. The indicator polynomial for counting χ_k in field $\mathcal{D}_k$ is now constructed as $P_{\chi_{k,1}\wedge\ldots\wedge\chi_{k,\|\mathcal{D}_k\|}}(x_{k,1}, \ldots, x_{1,\|\mathcal{D}_k\|}) = \prod_{l=1}^{\|\mathcal{D}_k\|} P_{\chi_{k,l}}(x_{k,l})$, where $x_{k,l}$ represents the l-th bit in the generic field $\mathcal{D}_k$, and $\chi_{k,l}$ denotes the corresponding queried bit value. Applying arithmetization to "transform" from Boolean to multivariate polynomials, Boolean expressions of m generic fields can be converted into equivalent multiple binary fields. For convenience in later sections, we call the conversion to binary fields "GF(2) arithmetized" (shortly "G"), while the original is "Basic" (shortly "B"). We note that although the number of coefficients of the GF(2) arithmetized multivariate indicator polynomial corresponding to each query remains the same as in the generic case, the (multivariate) degree of the GF(2) arithmetized polynomial is much lower at $\deg(P^{(G)}) = \sum_{k=1}^{m} \|\mathcal{D}_k\| = \sum_{k=1}^{m} \lceil \log_2 |\mathcal{D}_k| \rceil \ll \sum_{k=1}^{m} (|\mathcal{D}_k| - 1) = \deg(P^{(B)})$. This implies a significant improvement for computational costs on the cloud. We refer to EPiC's evaluation in Section 4 for details.

3.5 Encryption

Since EPiC's indicator polynomial based counting technique involves additions and multiplications on ciphertexts, a homomorphic encryption scheme is needed as a building block. While there already exist various schemes [6, 9, 13, 20], their computational complexities are high, rendering their use in current clouds impractical. Although EPiC can seamlessly integrate related work, we use a novel, specific encryption scheme. As we will see, this scheme does not enjoy the same security properties, i.e., IND-CPA, as related work, but only security with respect to definitions 3 and 4 as required in the specific context of EPiC. Due to its weaker security properties, our scheme is especially practical in the settings we target. We design a somewhat homomorphic encryption scheme derived from the computational Private Information Retrieval (cPIR) technique of Trostle and Parrish [18]. Our new scheme is a secret key encryption scheme, where the cloud does not have the secret key to decrypt the data, but instead blindly performs operations on outsourced data.

Key generation – KEYGEN$(s_1, s_2, n, \mathcal{D})$: Parameters $s_1, s_2 \in \mathbb{N}$ are security parameters, $n \in \mathbb{N}$ is the upper bound for the total number of records in the data set, and $\mathcal{D} = \mathcal{D}_1 \times \ldots \times \mathcal{D}_m$ is the multi-domain of m countable fields. KEYGEN computes a random prime q such that $q > n$, a random prime p, and a random $b \in \mathbb{Z}_p$. The secret key, the output of KEYGEN, is defined as $K := \{p, b\}$. The prime p is generated based

[2] $\|X\| = \lceil \log_2 |X| \rceil$ denotes size in bits of X

on the following condition (refer to Appendix A for details).

$$\|p\| \geq s_1 + \|n\| + \|q\| + \sum_{k=1}^{m}(s_2 + \|q\|) \cdot (|\mathcal{D}_k| - 1). \tag{3}$$

Encryption – $\textsc{Enc}(\mathcal{P})$: Selects a random number r, $\|r\| \leq s_2$, and encrypts the plaintext $\mathcal{P}$ to $\mathcal{C} = \textsc{Enc}(\mathcal{P}) := b \cdot (r \cdot q + \mathcal{P}) \bmod p$.

Decryption – $\textsc{Dec}(\mathcal{C})$: Decrypts $\mathcal{C}$ to $\mathcal{P} = \textsc{Dec}(\mathcal{C}) := b^{-1} \cdot \mathcal{C} \bmod p \bmod q$.

The security of our encryption scheme is based on the Hidden Modular Group Order hardness assumption and the cPIR protocol in [18]. The rationale is that, for appropriate security parameters, more than half of the bits of p are still secret against any PPT adversary (Theorem 4.5 in Trostle and Parrish [18]); and if a PPT adversary can break the cPIR protocol, the Hidden Group Order p is also revealed, violating the assumption. Based on the proof of the cPIR protocol, we will prove EPiC's security properties in Section 3.7. The addition and multiplication operations on ciphertexts take place in the integers. There is no modulo reduction, as the cloud does not know p. One can verify that this scheme provides additively and multiplicatively homomorphic properties (see Appendix A). Note that a multiplication of ciphertexts will increase the exponent of b in the result. Therefore, in general, a decryption of a ciphertext after $(j-1)$ multiplications has to be $\mathcal{P}^j = \textsc{Dec}(\mathcal{C}^j, j) := b^{-j} \cdot \mathcal{C}^j \bmod p \bmod q$. So, contrary to related work, decryption in our scheme requires the number of multiplications performed on the ciphertext. Also, EPiC's encryption scheme does also not allow addition of two ciphertexts that have different exponents of b, i.e., are results of a different number of multiplications. In the particular context of EPiC, we can accept these limitations, and we gain high computation efficiency in return. Finally, this scheme is only somewhat homomorphic: as the cloud cannot perform modulo operations, ciphertexts increase for every multiplication and addition (see Appendix A). This requires a careful selection of q and p in advance, such that decryption remains possible.

3.6 Detailed Protocol Description

With all ingredients ready, we now describe EPiC using the notation of Section 2.1.

$\textsc{KeyGen}(\kappa)$: Based on security parameter κ, cloud user $\mathcal{U}$ chooses s_1, s_2 for the somewhat homomorphic encryption, determines an upper bound n for the total number of records that might be stored and the appropriate multi-domain $\mathcal{D}$ for the countable fields. $\mathcal{U}$ generates a secret key K from the somewhat homomorphic encryption $\textsc{KeyGen}(s_1, s_2, n, \mathcal{D})$ and a symmetric key K' for a block cipher such as AES used for non-countable data. The secret key $\mathcal{S} := \{K, K'\}$ is used throughout EPiC.

$\textsc{Encrypt}(\mathcal{S}, \mathcal{R})$: Assume $\mathcal{U}$ wants to store n records $\mathcal{R} = \{R_1, \ldots, R_n\}$. Each record R_i is encrypted separating the countable values $R_{i,k}$ from the rest of the record. $R_{i,k}$ is encrypted using the somewhat homomorphic encryption mechanism, i.e., $E_{R_{i,k}} := \textsc{Enc}(\{p, b\}, R_{i,k})$. For the rest of the record R_i, a random initialization vector IV is chosen and the record is $\text{AES}_K - \text{CBC}$ encrypted. In conclusion, a record R_i encrypts to $E_{R_i} := \{E_{R_{i,1}}, \ldots, E_{R_{i,m}}, IV, \text{AES}_K - \text{CBC}(R_{i,rest})\}$. The output of $\textsc{Encrypt}$ is the sequence of encrypted records. $\mathcal{E} := \{E_{R_1}, \ldots, E_{R_n}\}$.

Algorithm 1: ProcessQuery

```
For each Mapper $M$:
    init $s_{\mathbf{j}} := 0, \forall \mathbf{j} \in \mathcal{D}$
    forall $E_{R_i}$ in InputSplit($M$) do
        read $\{E_{R_{i,1}}, \ldots, E_{R_{i,m}}\}$
        forall $\boldsymbol{j} = (j_1, \ldots, j_k) \in \mathcal{D}$ do
            $s_{\mathbf{j}} := s_{\mathbf{j}} + \prod_{k=1}^{m} (E_{R_{i,k}})^{j_k}$
        end
    end
    emit $\{\mathbf{j}, s_{\mathbf{j}}\}, \forall \mathbf{j} \in \mathcal{D}$

Reducer $R$:
    init $E_{\Sigma} := 0, S_{\mathbf{j}} := 0, \forall \mathbf{j} \in \mathcal{D}$
    forall $\{\boldsymbol{j}, s_{\boldsymbol{j}}\}$ in MappersOutput do
        $S_{\mathbf{j}} := S_{\mathbf{j}} + s_{\mathbf{j}}$
    end
    forall $\boldsymbol{j}$ in $\mathcal{D}$ do
        $E_{\Sigma} := E_{\Sigma} + E_{a_{\mathbf{j}}} \cdot S_{\mathbf{j}}$
    end
    write $\{E_{\Sigma}\}$
```

Upload($\mathcal{E}$): Upload simply sends all records as one large file to the MapReduce cloud where the file is automatically split into *InputSplits*.

PrepareQuery($\mathcal{S}, \chi$): To prepare a query for χ, $\mathcal{U}$ computes the $|\mathcal{D}|$ coefficients $a_{\mathbf{j}}$, $\mathbf{j} \in \mathcal{D}$, of the indicator polynomial $P_{\chi}(\mathbf{x})$ as described in Section 3.3. Coefficients $a_{\mathbf{j}}$ are encrypted and sent to the cloud. The cloud will be using these coefficients to perform the evaluation of $P_{\chi}(\mathbf{x})$. Consequently in EPiC, the output Q of PrepareQuery sent to the cloud is $Q := \{E_{a_{\mathbf{j}}}, \mathbf{j} \in \mathcal{D}\}$.

ProcessQuery($Q, \mathcal{E}$): Based on the data set size and the cloud configuration, the MapReduce framework selects M Mapper nodes and 1 Reducer node.

Algorithm 1 depicts the specification of EPiC's *map* and *reduce* functions that will be executed by the cloud. In the mapping phase, for each input record R_i in their locally stored InputSplits, the Mappers compute in parallel all monomials $\prod_{k=1}^{m} (E_{R_{i,k}})^{j_k}$ of the countable fields and add the same-degree monomials together. After the Mappers finish scanning over all records, the sums $s_{\mathbf{j}}$ of monomials are output as key-value pairs. These pairs contain the multi-degree $\mathbf{j}$ as key, and the computed sum $s_{\mathbf{j}}$ as value. In MapReduce, output of the Mappers is then automatically sent ("emit") to the Reducer. The sums $s_{\mathbf{j}}$ emitted by each Mapper are taken over records in the InputSplit corresponding to that Mapper only. The Reducer, therefore, based on the sums received from all Mappers, combines them together to obtain the *global* sums $S_{\mathbf{j}}$, i.e., the sums over all records in the data set. In a last step, the Reducer uses the coefficients $E_{a_{\mathbf{j}}}$ received from $\mathcal{U}$ to evaluate the polynomial by computing the inner product with the global sums. The result E_{Σ} is sent back to $\mathcal{U}$ and can be decrypted to obtain the count value.

Decode($\mathcal{S}, E_{\Sigma}$): $\mathcal{U}$ receives E_{Σ} and computes the counting result $\sigma = \text{Dec}(E_{\Sigma})$.

3.7 Privacy Analysis

We now formally prove Storage and Counting privacy for EPiC and its underlying encryption. We stress that, below, we neither target nor prove that our novel "encryption" provides traditional IND-CPA security. Instead, we show that, in combination with other details of our protocol, it provides security according to definitions 3 and 4.

Lemma 1 (Storage privacy). *Based on the security of the cPIR scheme by Trostle and Parrish [18], EPiC preserves storage privacy.*

Proof. cPIR-security, e.g., by Trostle and Parrish [18], can be summarized as follows. With a $u \times u$ bit database, a user wants to retrieve an y-th row and sends an encrypted PIR request to the cloud: $P = \{E_{v_1}, \ldots, E_{v_u}\}$, where $E_{v_k} = \textsc{Enc}(v_k)$, cf. Section 3.5, and $v_k = 1$, if $k = y$, and $v_k = 0$ otherwise. This cPIR protocol is secure *iff* for all PPT adversaries $\mathcal{A}^*$, the probability of finding y is negligible more than guessing, i.e., $\Pr[\mathcal{A}^*(P) = y] \leq 1/u + \epsilon^*(\kappa)$.

We now prove our lemma by reduction from cPIR security. We show that, for security parameter κ, any PPT $(t(\kappa), \epsilon(\kappa))$-adversary $\mathcal{A}$ breaking EPiC's storage privacy (Definition 3) in $t(\kappa)$ steps with non-negligible advantage $\epsilon(\kappa)$ can be used to construct a $(t^*(\kappa), \epsilon^*(\kappa))$-adversary $\mathcal{A}^*$ as a subroutine breaking the cPIR protocol in [18]. We construct $\mathcal{A}^*$ based on the *parity* of u.

1. u **is odd.** First, $\mathcal{A}^*$ receives as input the PIR request P and splits P into two halves $\mathcal{E} = \{E_{v_1}, \ldots, E_{v_{\lfloor u/2 \rfloor}}\}$, $\mathcal{E}' = \{E_{v_{\lfloor u/2 \rfloor + 1}}, \ldots, E_{u-1}\}$, i.e., treating the PIR request as two EPiC data sets of the same size ($\lfloor u/2 \rfloor$ records). Since E_{v_k} are either encryptions of 0 or 1, $\mathcal{E}$ and $\mathcal{E}'$ are now viewed as single-binary-field data sets, where each record contains only 1 countable binary field. $\mathcal{A}^*$ randomly selects $l_1, l_2, l'_1, l'_2 \in [1, u]$ and creates two EPiC counting queries $Q = \{E_{v_{l_1}}, E_{v_{l_2}}\}, Q' = \{E_{v_{l'_1}}, E_{v_{l'_2}}\}$. These are two valid queries, because for single-binary-field data sets $\mathcal{E}, \mathcal{E}'$, any EPiC query contains exactly 2 encrypted coefficients of 0 or 1, cf. Section 3.3. Then $\mathcal{A}^*$ runs ProcessQuery on $\mathcal{E}$ with Q, and $\mathcal{E}'$ with Q', thereby obtaining E_Σ and E'_Σ. $\mathcal{A}^*$ forwards $I = \{\mathcal{E}, \mathcal{E}', Q, Q', E_\Sigma, E'_\Sigma\}$ to $\mathcal{A}$. $\mathcal{A}^*$'s output depends on $\mathcal{A}$'s output as follows.

If $\mathcal{A}$ outputs 0, $\mathcal{A}^*$ outputs u. The intuition is that, since $\mathcal{A}$ "believes" the two halves $\mathcal{E}$ and $\mathcal{E}'$ are the same, $\mathcal{A}'$ concludes that the requested element must not belong to either $\mathcal{E}$ or $\mathcal{E}'$, i.e., $v_u = 1$.

If $\mathcal{A}$ outputs 1, $\mathcal{A}^*$ randomly selects $k \in [1, u-1]$ and outputs k. The intuition is that "$\mathcal{A}$ outputs 1" indicates the requested row index is between 1 and $u - 1$, and $\mathcal{A}^*$ simply makes a random guess for it. The probability for $\mathcal{A}^*$ to output correctly is $\Pr[\mathcal{A}^*(P) = y] = \Pr[\mathcal{A} = 0 | y = u] \cdot \Pr[y = u] + \Pr[\mathcal{A} = 1, k = y | y < u] \cdot \Pr[y < u] = \left(\frac{1}{2} + \epsilon(\kappa)\right) \cdot \frac{1}{u} + \left(\frac{1}{2} + \epsilon(\kappa)\right) \cdot \frac{1}{u-1} \cdot \frac{u-1}{u} = \frac{1}{u} + \frac{2\epsilon(\kappa)}{u}$. Therewith, $\mathcal{A}^*$ has a non-negligible advantage of $\epsilon^*(\kappa) = 2\epsilon(\kappa)/u$ in finding y.

2. u **is even.** $\mathcal{A}^*$ makes a new PIR request P' by removing the last element v_u from P, that is $P' = \{E_{v_1}, \ldots, E_{v_{u-1}}\}$. Then $\mathcal{A}^*$ uses the same approach as in the odd case for P', i.e., splitting P' into 2 halves, feeding them to $\mathcal{A}$. Now, $\mathcal{A}^*$ outputs $u - 1$, if $\mathcal{A}$ outputs 0, or outputs random $k \in [1, u-2]$ otherwise. It can be observed that $\mathcal{A}^*$ can find y with non-negligible probability, only if $y \neq u$, i.e., the requested element is not the last element discarded from P. Otherwise, $\mathcal{A}^*$ cannot find y (i.e., zero probability). More precisely, the probability of correct guess is $\Pr[\mathcal{A}^*(P) = y]$ $= \Pr[A^*(P') = y | y < u] \cdot \Pr[y < u] + \Pr[A^*(P') = y | y = u] \cdot \Pr[y = u]$ $= \left(\frac{1}{u-1} + \frac{2\epsilon(\kappa)}{u-1}\right) \cdot \frac{u-1}{u} + 0 \cdot \frac{1}{u} = \frac{1}{u} + \frac{2\epsilon(\kappa)}{u}$. Therefore, $\mathcal{A}^*$ also has a non-negligible advantage of $2\epsilon(\kappa)/u$ in finding y.

Consequently, in both cases, $\mathcal{A}^*$ has a non-negligible advantage $\epsilon^*(\kappa) = 2\epsilon(\kappa)/u$ of breaking the cPIR protocol in $t^*(\kappa) = t(\kappa)$ steps, rendering our reduction tight. □

Lemma 2 (Counting privacy). *Based on the security of the cPIR scheme by Trostle and Parrish [18], EPiC preserves counting privacy.*

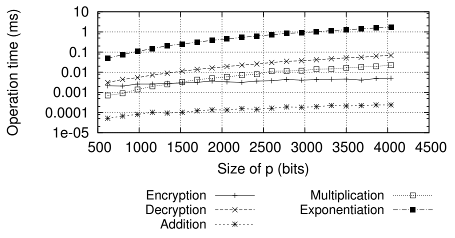

Fig. 1. Computation time on ciphertext.

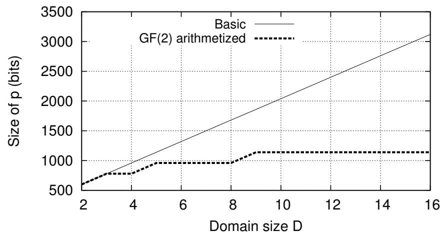

Fig. 2. Size of p depends on size of domain $\mathcal{D}$.

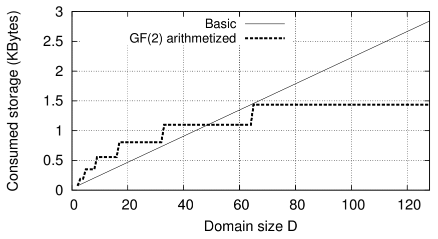

Fig. 3. Consumed storage for each field.

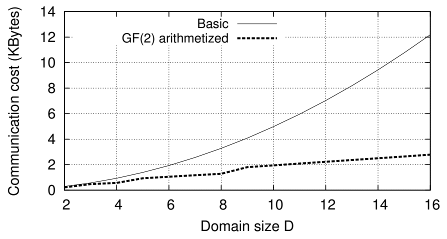

Fig. 4. Communication cost

Proof. We prove our lemma by reduction from cPIR security. Recall the cPIR-security definition as in Lemma 1's proof. We assume the existence of a PPT $(t(\kappa), \epsilon(\kappa))$-EPiC-adversary $\mathcal{A}$ breaking EPiC's counting privacy (Definition 4) in $t(\kappa)$ steps with non-negligible advantage $\epsilon(\kappa)$. In the following, we construct a new $(t^*(\kappa), \epsilon^*(\kappa))$-PIR-adversary $\mathcal{A}^*$ that breaks this cPIR security.

$\mathcal{A}^*$ receives as input the PIR request $P = \{E_{v_1}, \ldots, E_{v_u}\}$, where $v_y = 1$ and $v_k = 0, \forall k \neq y$. The goal of $\mathcal{A}^*$ is to guess y. First, $\mathcal{A}^*$ sets $\mathcal{E} = \mathcal{E}' = P$ and randomly picks 4 elements $E_{l_1}, E_{l_2}, E_{l'_1}, E_{l'_2}$ from P to make two EPiC queries $Q = \{E_{l_1}, E_{l_2}\}$, $Q' = \{E_{l'_1}, E_{l'_2}\}$. Note that $\mathcal{E}, \mathcal{E}'$ can be viewed as EPiC's two identical single-binary-field data sets, and Q, Q' are valid queries (corresponding to some patterns χ, χ') for $\mathcal{E}, \mathcal{E}'$. Then $\mathcal{A}^*$ runs PROCESSQUERY on $\mathcal{E}$ with Q and on $\mathcal{E}'$ with Q' to obtain E_Σ, E'_Σ. Now, $\mathcal{A}^*$ forwards $I = \{\mathcal{E}, \mathcal{E}', Q, Q', E_\Sigma, E'_\Sigma\}$ to $\mathcal{A}$ and observes $\mathcal{A}$'s output.

Let $U = \{1, \ldots, u\}, L = \{l_1, l_2, l'_1, l'_2\}$. If $\mathcal{A}$ returns 1, $\mathcal{A}^*$ concludes that the two queries Q and Q' are identical, implying that $E_{v_y} \notin Q \cup Q'$, i.e., $y \notin L$. Therewith, $\mathcal{A}^*$ makes a guess for y by selecting a random $k \in U \setminus L$ and outputs k. Otherwise, if $\mathcal{A}$ returns 0, $\mathcal{A}^*$ concludes that v_y is in either Q or Q', thus $\mathcal{A}^*$ outputs a random $k \in L$. The probability of the correct guess is $\Pr[\mathcal{A}^*(P) = y] = \Pr[\mathcal{A}(I) = 1, k = y | y \in U \setminus L] \cdot \Pr[y \in U \setminus L] + \Pr[\mathcal{A}(I) = 0, k = y | y \in L] \cdot \Pr[y \in L] = \left(\frac{1}{2} + \epsilon(\kappa)\right) \cdot \frac{1}{u-4} \cdot \frac{u-4}{u} + \left(\frac{1}{2} + \epsilon(\kappa)\right) \cdot \frac{1}{4} \cdot \frac{4}{u} = \frac{1}{u} + \frac{2\epsilon(\kappa)}{u}$. That is A^* has a non-negligible advantage $\epsilon'(\kappa) = 2\epsilon(\kappa)/u$ of breaking the cPIR protocol in $t^*(\kappa) = t(\kappa)$ steps. □

4 Evaluation

To show its real-world applicability, we have implemented and evaluated EPiC in Hadoop's MapReduce framework v1.0.3 [3]. The source code is available for download [2]. We have evaluated EPiC on Amazon's public MapReduce cloud [1]. Our EPiC implementation is written in Java, and all cryptographic operations are *unoptimized*, relying on

Java's standard BigInteger data type. Still, exponentiation, e.g. $\mathcal{C}^j$, with $j = 15$ and $|\mathcal{C}| \approx 4000$ takes < 2ms on a 1.8GHz Intel Core i7 laptop, a single addition is not measurable with $< 1\mu$s. Figure 1 shows a benchmark of various operations on the ciphertexts using our encryption scheme. In our evaluation, we use security parameters $s_1 = 400$ bits as suggested by Trostle and Parrish [18] for good security, and $s_2 = |r| = 160$ bits. We have implemented a data generator program to randomly generate patient records with m countable fields with size between 4 and 10 bits.

We have evaluated the performance of EPiC by comparing our "Basic" and "GF(2) arithmetized" solutions with a "non-privacy-preserving" solution. Unless otherwise stated, the single/multi-domain size in both "Basic" and "GF(2) arithmetized" solutions is always set to the same value $|\mathcal{D}|$ for comparison. For shorter presentation, we use "B" as index of the cost in Basic approach, and "G" in GF(2) arithmetized approach, e.g., $\|p_B\|, \|p_G\|$ indicate the size in bits of p in Basic, GF(2) arithmetized approach respectively. We also set $u = s_1 + \|n\| + \|q\|$, $v = s_2 + \|q\|$ and use them as fixed parameters (with respect to $|\mathcal{D}|$) when evaluating the cost.

Size of prime p As discussed in Section 3.5, prime q depends only on the number of records n, while prime p also depends on $|\mathcal{D}|$. Derived from Equation (3), we show the benefit of the GF(2) arithmetized approach ($m = \|\mathcal{D}\|, |\mathcal{D}_k| = 2$) by demonstrating that a conversion to multiple binary fields reduces $\|p\|$ significantly to $\|p_G\| = u + \|\mathcal{D}\| \cdot v$, while the Basic approach ($m = 1, |\mathcal{D}_1| = |\mathcal{D}|$) requires that $\|p_B\| = u + (|\mathcal{D}| - 1) \cdot v$. Figure 2 shows the logarithmic increase of size of p with GF(2) arithmetized approach and linear increase with Basic approach.

Storage cost The storage cost depends on the size of the data stored on the cloud. In the following, we only focus on the size of the countable fields, which is determined by the size of prime p. In Basic approach, a generic field of domain $\mathcal{D}$ requires a storage of $S_B = \|p_B\| = u + (|\mathcal{D}| - 1) \cdot v$ bits. In GF(2) arithmetized approach, the equivalent generic field converted to binary fields requires a storage of $S_G = \|\mathcal{D}\| \cdot \|p_G\| = \|\mathcal{D}\| \cdot (u + \|\mathcal{D}\| \cdot v)$ bits. Again, in Figure 3, we see a linear increase of storage in terms of the domain size $|\mathcal{D}|$ in Basic approach, while the GF(2) arithmetized approach consumption increase only logarithmically. This demonstrates a significant improvement on reducing the data size when "splitting" them into binary fields.

User computation cost $\mathcal{U}$ prepares the query in plaintext, which incurs very low computation cost compared to the cloud computation performed on ciphertexts. The encryption of one coefficient takes roughly 1 ms (Figure 1), resulting in approximately $|\mathcal{D}|$ ms for encrypting $|\mathcal{D}|$ coefficients in the query, regardless of using Basic or GF(2) arithmetized. The count is obtained by decrypting only one final sum, therefore the total user computation cost is negligible compared to the cloud cost (shown below).

User communication cost Due to oblivious counting, user $\mathcal{U}$ prepares and sends all $|\mathcal{D}|$ coefficients corresponding to *all* monomials to the cloud. The total size of the encrypted coefficients is $|\mathcal{D}| \cdot \|p\|$. In Basic approach, the query size is $Q_B = |\mathcal{D}| \cdot \|p_B\| = |\mathcal{D}| \cdot (u + (|\mathcal{D}| - 1) \cdot v)$. In contrast, the GF(2) arithmetized approach reduces to $Q_G = |\mathcal{D}| \cdot \|p_G\| = |\mathcal{D}| \cdot (u + \|\mathcal{D}\| \cdot v)$. For example of a data set containing $n = 10^6$ records with a countable

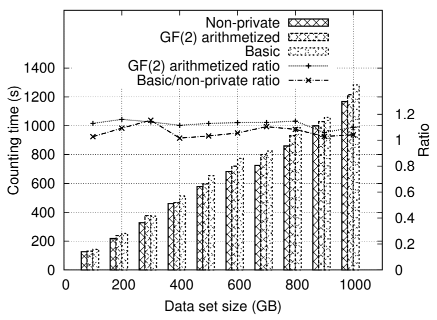

Fig. 5. Counting time vs. data set size. $|\mathcal{D}| = 16$.

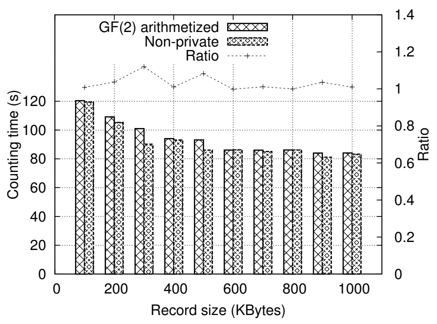

Fig. 6. 50GB, varying record size. $|\mathcal{D}| = 16$.

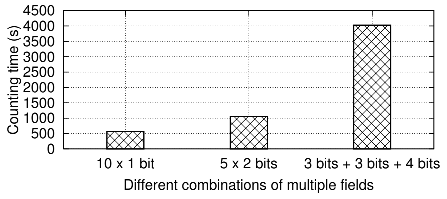

Fig. 7. Effect of different field combinations.

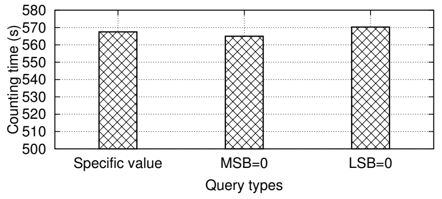

Fig. 8. Different query types on the same data.

field of domain size $|\mathcal{D}| = 1024$ (i.e., $\|\mathcal{D}\| = 10$), the corresponding query size in each approach is $Q_B = 22.5$ MBytes, and $Q_G = 280$ KBytes, respectively.

The answer size (size in bits of the received ciphertext as final sum) depends on the maximum size of the multivariate monomial. The size of a monomial is determined by the ciphertext size (i.e., $\|p\|$) and the number of performed multiplications, i.e., its multi-degree. Let d denote the maximum multi-degree of monomials, then, $d = |\mathcal{D}|$ in the Basic approach, and $d = \|\mathcal{D}\|$ in the GF(2) arithmetized approach. We have $A_B = |\mathcal{D}| \cdot \|p_B\| = |\mathcal{D}| \cdot (u + (|\mathcal{D}| - 1) \cdot v)$ and $A_G = \|\mathcal{D}\| \cdot \|p_G\| = \|\mathcal{D}\| \cdot (u + \|\mathcal{D}\| \cdot v)$. For example of a data set containing $n = 10^6$ records with a countable field of domain size $|\mathcal{D}| = 1024$, the answer size in each approach is $A_B = 22.5$ Mbytes, and $A_G = 2.7$ KBytes, respectively.

Total transfer cost: The total communication cost, $C = Q + A$, as shown in Figure 4, is much less in GF(2) arithmetized approach than in Basic approach: $C_B = Q_B + A_B = 2 \cdot |\mathcal{D}| \cdot (u + (|\mathcal{D}| - 1) \cdot v)$, $C_G = Q_G + A_G = (|\mathcal{D}| + \|\mathcal{D}\|) \cdot (u + \|\mathcal{D}\| \cdot v)$.

Cloud computation We have evaluated the cloud computation cost for large-scale data sets on Amazon's public cloud. As Amazon imposes an (initial) limit of 20 instances per job, we restrict ourselves to 20 Standard Large On-Demand instances [1]. Each instance comprises 4 2.27GHz Intel Xeon CPUs and a total of 7.5 GB RAM.

Variable data set size: First, we fix the size of each record to 1 MB. The data set size (x-axis) is varied from 100 GB to 1 TB. We query a countable field of size $|\mathcal{D}| = 16$. Figure 5 shows the average counting time for a MapReduce job on the whole data set of different sizes. The y-axis shows the total time for MapReduce to evaluate the user's query. This is the time that a user has to pay for to Amazon. To put our results into perspective, we not only show the time for both Basic and GF(2) arithmetized approaches, but as well the time a "non-privacy-preserving" counting would take, i.e., the countable field is not encrypted and directly counted. Moreover, we also show the overhead ratio between EPiC's two approaches and non-private counting. The additional overhead introduced by EPiC over non-private counting is less than 20%. We conjecture that only 20% overhead/additional cost over non-privacy-preserving counting is acceptable in many real-world situations, rendering EPiC practical.

Variable record size: To also evaluate the effect of the size of the records on the general performance, we run the system with a fixed data set size of 50 GB. The record size is changed from 100 KB to 1 MB. Figure 6 shows that, while IO time remains unchanged, a higher number of records increases counting time in EPiC. However, the overhead of EPiC is still under 20% even for small record sizes such as 100 KB compared to non-private counting. That is, EPiC is efficient even for small patient records.

Effect of multiple fields: To study the efficiency of transforming a single countable field $\mathcal{D}$ into multiple fields of different size, we conduct an experiment on a data set size of 100GB. The total domain size is set to $|\mathcal{D}| = 1024$ (10 bits). We compare three cases: (a) transform $\mathcal{D}$ into 10 single binary fields; (b) transform $\mathcal{D}$ into 5 quaternary fields each of 2 bits; (c) transform $\mathcal{D}$ into 3 fields of 3 bits, 3 bits, and 4 bits, respectively. In Figure 7, we can see that the GF(2) arithmetized approach yields the best performance.

Query types: Finally, to evaluate the effects of different query types on the performance, we run EPiC with a fixed data set of 100 GB. Total domain size is $|\mathcal{D}| = 1024$. We make 3 different queries: (a) query for a specific value; (b) query for the MSB of the field equal to 0; (c) query for the LSB of the field equal to 0. Figure 8 demonstrates that there is no significant difference in counting time between different queries.

5 Related Work

Protecting privacy of outsourced data and delegated operations in a cloud computing environment is the perfect setting for fully homomorphic encryption. While there is certainly a lot of ongoing research in fully homomorphic encryption (see Vaikuntanathan [19] for an overview), current implementations indicate high storage and computational overhead [10, 14], rendering fully homomorphic encryption impractical for the cloud.

Similar to EPiC, Lauter et al. [13] observe that often weaker "somewhat" homomorphic encryption might be sufficient. Lauter et al. [13]'s scheme is based on a protocol for lattice-based cryptography by Brakerski and Vaikuntanathan [6]. However, for the specific application scenario considered in this paper, EPiC's somewhat homomorphic encryption scheme allows for much faster exponentiation.

Superficially, our work bears similarity with the work of Kamara and Raykova [12] that protect polynomial evaluation by randomized reduction techniques. With q being the degree of a polynomial, the user splits each data record into $2 \cdot q + 1$ shares, each of size $2 \cdot q + 1$. Shares are then uploaded and evaluated in parallel, and results are aggregated. However, storage expansion, even for modest values of q, the approach quickly becomes impractical. Also, for different polynomials, the user would need to upload the data multiple times.

Searching on encrypted data has received a lot of attention recently, cf. seminal papers [5, 16]. While closely related, it is far from straightforward to adopt these schemes to perform efficient counting in a highly parallel cloud computing, e.g., MapReduce environment. Also notice that, e.g., Boneh et al. [5] rely on the computation of very expensive bilinear pairings for each element of a data set, rendering this approach impractical in a cloud setting.

Much research has been done to compute statistics in a privacy-preserving manner using *differential privacy*, see the seminal paper by Dwork [8]. Contrary to the threat model considered in this paper, the adversary in differential privacy research is not the cloud infrastructure, but a curious user querying statistics to learn information about individual entries in a data set. EPiC addresses the opposite problem, where a user does not trust the cloud infrastructure.

6 Conclusion

In this paper, we present EPiC to address a fundamental problem of statistics computation on outsourced data: privacy-preserving pattern counting. EPiC's main idea is to count occurrences of patterns in outsourced data through a privacy-preserving summation of the pattern's indicator-polynomial evaluations over the encrypted dataset records. Using a "somewhat homomorphic" encryption mechanism, the cloud neither learns any information about outsourced data nor about the queries performed. Our implementation and evaluation results for MapReduce running on Amazon's cloud with up to 1 TByte of data show only modest overhead compared to non-privacy-preserving counting. Contrary to related work, this makes EPiC practical in a real-world cloud computing setting today.

Bibliography

[1] Amazon. Elastic MapReduce, 2012. `http://aws.amazon.com/elasticmapreduce/`.

[2] Anonymized for submission. EPiC Source Code, 2013. Download from `https://www.dropbox.com/s/abr7tuck1akv1ly/epic.zip`.

[3] Apache. Hadoop, 2010. `http://hadoop.apache.org/`.

[4] L. Babai and L. Fortnow. Arithmetization: A New Method In Structural Complexity Theory. *Computational Complexity*, pages 41–66, 1991. ISSN 1016-3328.

[5] D. Boneh, G. DiCrescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Proceedings of Eurocrypt*, pages 506–522, Barcelona, Spain, 2004.

[6] Z. Brakerski and V. Vaikuntanathan. Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages. In *Proceedings of Annual Cryptology Conference*, pages 505–524, Santa Barbara, USA, 2011. ISBN 978-3-642-22791-2.

[7] 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.

[8] C. Dwork. Differential Privacy. In *Proceedings of Colloquium Automata, Languages and Programming*, pages 1–12, Venice, Italy, 2006. ISBN 3-540-35907-9.

[9] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of ACM Symposium on Theory of Computing*, pages 169–178, Bethesda, USA, 2009. ISBN 978-1-60558-506-2.

[10] C. Gentry and S. Halevi. Implementing Gentry's fully-homomorphic encryption scheme. In *Proceedings of International Conference on Theory and Applications of Cryptographic Techniques*, pages 129–148, Tallinn, Estonia, 2011. ISBN 78-3-642-20464-7.

[11] Hadoop. Powered by Hadoop, list of applications using Hadoop MapReduce, 2011. `http://wiki.apache.org/hadoop/PoweredBy`.

[12] S. Kamara and M. Raykova. Parallel Homomorphic Encryption. Technical Report, ePrint Report 2011/596, 2011. `http://eprint.iacr.org/2011/596`.

[13] K. Lauter, N. Naehrig, and V. Vaikuntanathan. Can Homomorphic Encryption be Practical? In *Proceedings of ACM Workshop on Cloud Computing Security*, Chicago, USA, 2011. ISBN 978-1-4503-1004-8.

[14] H. Perl, M Brenner, and M. Smith. An Implementation of the Fully Homomorphic Smart-Vercauteren Crypto-System. In *Proceedings of Conference on Computer and Communications Security*, pages 837–840, Chicago, USA, 2011. ISBN 978-1-4503-0948-6.

[15] A. Shamir. IP = PSPACE. *Journal of the ACM*, 39(4):869–877, 1992. ISSN 0004-5411.

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

[17] The Telegraph. Patient records go online in data cloud, 2011. `http://www.telegraph.co.uk/health/healthnews/8600080/Patient-records-go-online-in-data-cloud.html`.

[18] J. Trostle and A. Parrish. Efficient computationally private information retrieval from anonymity or trapdoor groups. In *Proceedings of Conference on Information Security*, pages 114–128, Boca Raton, USA, 2010.

[19] Vinod Vaikuntanathan. Computing blindfolded: New developments in fully homomorphic encryption. In *Proceedings of the 2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, FOCS '11, pages 5–16, 2011.

[20] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In *Proceedings of International Conference on Theory and Applications of Cryptographic Techniques*, pages 24–43, Monaco, 2010. ISBN 3-642-13189-1.

A Properties of EPiC's somewhat homomorphic encryption

Along with EPiC's encryption scheme specified in Section 3.5, for ease of presentation, we introduce $e(\mathcal{P})$ as a shorthand for $r \cdot q + x$. Therewith, an encryption of a plaintext $\mathcal{P}$ can be rewritten as $C = \textsc{Enc}(\mathcal{P}) = b \cdot e(\mathcal{P}) \bmod p$.

Somewhat homomorphism With the appropriate selection of prime p, cf. Equation (3), EPiC's encryption scheme supports up to n additions and $|\mathcal{D}|$ multiplications on encrypted data. The homomorphism of the encryption scheme is shown below.
Additively homomorphic: $\textsc{Dec}(\mathcal{C}_1^j + \mathcal{C}_2^j, j)$

$$\begin{aligned}
&= b^{-j} \cdot [(b \cdot e(\mathcal{P}_1) \bmod p)^j + (b \cdot e(\mathcal{P}_2) \bmod p)^j] \bmod p \bmod q \\
&= [b^{-j} \cdot (b \cdot e(\mathcal{P}_1) \bmod p)^j \bmod p \bmod q] + [b^{-j} \cdot (b \cdot e(\mathcal{P}_2) \bmod p)^j \bmod p \bmod q] \\
&= \textsc{Dec}(\mathcal{C}_1^j, j) + \textsc{Dec}(\mathcal{C}_2^j, j)
\end{aligned}$$

Multiplicatively homomorphic: $\textsc{Dec}(\mathcal{C}_1^j \cdot \mathcal{C}_2^k, j+k)$

$$\begin{aligned}
&= b^{-(j+k)} \cdot [(b \cdot e(\mathcal{P}_1) \bmod p)^j \cdot (b \cdot e(\mathcal{P}_2) \bmod p)^k] \bmod p \bmod q \\
&= [b^{-j} \cdot (b \cdot e(\mathcal{P}_1) \bmod p)^j \bmod p \bmod q] \cdot [b^{-k} \cdot (b \cdot e(\mathcal{P}_2) \bmod p)^k \bmod p \bmod q] \\
&= \textsc{Dec}(\mathcal{C}_1^j, j) \cdot \textsc{Dec}(\mathcal{C}_2^k, k)
\end{aligned}$$

Ciphertext size: As multiplications and additions are performed on integers without modulo operations, the size of the result increases after each operation.

– Addition: $\|\mathcal{C}_1 + \mathcal{C}_2\| = \begin{cases} \|\mathcal{C}_1\| + 1, & \text{if } \|\mathcal{C}_1\| = \|\mathcal{C}_2\|. \\ \max(\|\mathcal{C}_1\|, \|\mathcal{C}_2\|), & \text{otherwise.} \end{cases}$.
– Multiplication: $\|\mathcal{C}_1 \cdot \mathcal{C}_2\| = \|C_1\| + \|\mathcal{C}_2\| - 1$.
– Scalar multiplication: $\|n \cdot \mathcal{C}\| = \|n\| + \|\mathcal{C}\| - 1$.
– Exponentiation: $\|\mathcal{C}^j\| = j \cdot \|\mathcal{C}\| - (j-1)$.

Encrypting data and query We employ this encryption scheme to encrypt both countable fields and query. Using the secret key $K = \{p, b\}$ returned from KEYGEN, the user encrypts each countable field $R_{i,k}$ to $E_{R_{i,k}} = \text{ENC}(R_{i,k})$ before uploading his encrypted data to the cloud. The encryption of a counting query Q, as a sequence of encrypted coefficients $E_{a_\mathbf{j}}$, however, is more difficult. A simple method of encrypting $a_\mathbf{j}$ to $E_{a_\mathbf{j}} = \text{ENC}(a_\mathbf{j})$ does not allow the decryption of the final sum E_Σ of Equation 2 due to different exponents of b in the multivariate monomials, explained below.

Consider the cloud computation in Equation (2). For every $\mathbf{j} = (j_1, \ldots, j_m) \in \mathcal{D}$, $j_k \in \mathcal{D}_k$, the corresponding multivariate monomial $\prod_{k=1}^m (E_{R_{i,k}})^{j_k}$ has a multi-degree dependent on $\mathbf{j}$. More precisely, let $d_1(\mathbf{j}) = \deg \prod_{k=1}^m (E_{R_{i,k}})^{j_k}$ denote the multi-degree of the multivariate monomial $\prod_{k=1}^m (E_{R_{i,k}})^{j_k}$, then $d_1(\mathbf{j}) = \sum_{k=1}^m j_k$. If $a_\mathbf{j}$ was simply encrypted similarly to the encryption of countable fields, i.e., $E_{a_\mathbf{j}} = \text{ENC}(a_\mathbf{j})$, each product $E_{a_\mathbf{j}} \cdot \sum_{i=1}^n \prod_{k=1}^m (E_{R_{i,k}})^{j_k}$ would contain b of different degree equal to $1 + d_1(\mathbf{j})$. This would prohibit additions among them to obtain the *decryptable* final sum E_Σ, as our encryption scheme only allows adding ciphertexts containing the same exponents of b. Our approach is to "augment" the exponents of b in the coefficients. Specifically, we encrypt $a_\mathbf{j}$ to $E_{a_\mathbf{j}} := \text{ENC}(a_\mathbf{j}) \cdot (\text{ENC}(1))^{d_2(\mathbf{j})} = b^{d_2(\mathbf{j})+1} \cdot e(a_\mathbf{j}) \bmod p$, where $d_2(\mathbf{j}) = \sum_{k=1}^m (|\mathcal{D}_k| - j_k - 1)$. Therewith, E_Σ will contain b of degree $d = 1 + d_1(\mathbf{j}) + d_2(\mathbf{j}) = 1 + \sum_{k=1}^m (|\mathcal{D}_k| - 1)$, which is independent of $\mathbf{j}$. All multivariate monomials now have the same exponents of b, allowing successful decryption of E_Σ.

Selection of q and p Consider a ciphertext $\mathcal{C} = \text{ENC}(\mathcal{P}) = b \cdot e(\mathcal{P}) \bmod p$ and its decryption procedure: $\text{DEC}(\mathcal{C}) = b^{-1} \cdot \mathcal{C} \bmod p \bmod q = e(\mathcal{P}) \bmod p \bmod q \overset{(a)}{=} e(\mathcal{P}) \bmod q \overset{(b)}{=} \mathcal{P}$. For the equalities (a) and (b) to hold, the plaintext $\mathcal{P}$ needs to satisfy: 1) $\mathcal{P} \in GF(q)$ for (b) to hold; 2) $e(\mathcal{P}) \in GF(p)$ for (a) to hold.

In EPiC, user $\mathcal{U}$ receives the encrypted final sum E_Σ from the cloud. In this context, $\mathcal{P} = \sigma = \text{DEC}(E_\Sigma)$ is the plaintext count to be decrypted from E_Σ. The first condition requires that $\sigma \in GF(q)$. Since σ is the count value, it is at most the number of records in the data set, i.e., $\sigma \leq n$. This implies $q > n$.

Since $e(\sigma) = \sum_{\mathbf{j} \in \mathcal{D}} (e(a_\mathbf{j}) \cdot \sum_{i=1}^n \prod_{k=1}^m (e(R_{i,k}))^{j_k})$ and the second condition requires $e(\sigma) \in GF(p)$. Therewith, we establish the lower bound on the size of p as

$$\|p\| \geq \|n\| + \|q\| + \sum_{k=1}^m (s_2 + \|q\|) \cdot (|\mathcal{D}_k| - 1).$$

As our scheme relies on the Hidden Modular Group Order assumption, a security parameter s_1 has to be added to the size of p [18], therewith yielding the requirement for p as in Equation (3).