# EPiC: Efficient Privacy-Preserving Counting for MapReduce

Erik-Oliver Blass, Guevara Noubir, and Triet D. Vo-Huu
Northeastern University, Boston MA 02115, USA
{blass|noubir|vohuudtr}@ccs.neu.edu

**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, 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 somewhat homomorphic encryption scheme based on a previous work on the Hidden Modular Group assumption. This scheme 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 data sets sizes of up to 1 TByte shows only modest overhead 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 allows 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 [9]. MapReduce is offered by major public cloud providers today, such as Amazon [1], Google [13], IBM [15] or Microsoft [18]. 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 [14].

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. Examples for adversaries can be hackers that break into the cloud (data center) to steal data, insiders such as data center administrative staff who can easily access data, and other cloud users hosted on the same data center ("multi tenancy"). Finally, as cloud providers place data centers abroad in foreign countries with unclear privacy laws, local authorities are threatening outsourced data. Such attacks are realistic and have already been reported in the real-world [12, 21, 22, 25, 31].

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 [10] or see Vaikuntanathan [28] 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 [8, 11, 20, 29], 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. MapReduce comprises a specific two-phase setup, where (first) the workload is parallelized in the Map-phase, and (second) individual results are aggregated during the Reduce-phase to present a combined result to the user.

This paper presents an efficient, practical, yet privacy-preserving protocol for a fundamental data analysis primitive in MapReduce: *counting occurrences* of patterns [14]. 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 (plaintext) pattern, and the cloud will count the number of occurrences of this pattern (and therefore histograms) in the stored ciphertexts without detecting which pattern is being counted or how often the pattern occurs. A pattern is expressed as a Boolean formula on the fields of data records and can therefore specify a specific field value, a range of field values, but also more complex patterns consisting of conjunctions/disjunctions of fields values. 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. [17] to use *somewhat homomorphic* encryption to address specific privacy-preserving operations. In EPiC, we extend previous work on cPIR protocols [27] to design a new somewhat homomorphic encryption scheme, which is particularly efficient in the context that we target in this paper, the summation of polynomial evaluations. 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 pattern to the summation of a multivariate polynomial evaluated on the cloud encrypted records.
- A new "somewhat homomorphic" 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 [6].

## 2 Problem Statement

**Overview:** We will use an example application to motivate our work. Along the lines of recent reports [26], 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 $R$, besides raw data such as maybe a picture or some doctors' notes, also includes one or more fields $R.c$ containing some patterns. In practice, this field could denote the category or type of disease a patient is suffering from, e.g., "diabetes" or "hypertension". While one (or more) cloud users $\mathcal{U}$ add, remove or edit records, eventually one cloud user $\mathcal{U}$ wants to know, how many patients suffer from disease $\chi$. That is, user $\mathcal{U}$ wants to extract the frequency of occurrence of pattern $\chi$ and therewith how many records contain $R.disease = \chi$. 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 data stored, 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 the rest of this paper, we will assume the "pattern" a user $\mathcal{U}$ might look for to be "countable".

**Definition 1 (Cloud Counting).** *Let $\mathcal{R}$ denote a sequence of records $\mathcal{R} := \{R_1, \ldots, R_n\}$. Besides some random data, each record $R_i$ contains $m$ different "countable fields" $R_{i,k}, 1 \leq k \leq m$. $\mathcal{D}_k$ denotes the domain of the $k$-th field. Without loss of generality, we assume each value stored in the $k$-th field of the $i$-th record can take values $R_{i,k} \in \mathcal{D}_k = \{0, 1, \ldots, |\mathcal{D}_k| - 1\}$, $|\mathcal{D}_k|$ is the domain size[1] of $\mathcal{D}_k$. 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$(\kappa)$ : *using a security parameter $\kappa$, 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 pattern $\chi \in \mathcal{D}$.*
5. PROCESSQUERY$(Q, \mathcal{E})$ : *performs the actual counting. Uses a query $Q$, the sequence of ciphertexts $\mathcal{E}$, and outputs a result $E_\Sigma$.*

[1] Domain size $|\mathcal{D}_k|$ indicates the number of different values a field can take. We will use $\|\cdot\|$ later to denote the *size in bits* of some argument, e.g., $\|\mathcal{D}_k\| = \lceil \log_2 |\mathcal{D}_k| \rceil$, $\|x\| = \lceil \log_2 x \rceil$.

6. DECODE($\mathcal{S}, E_\Sigma$) : *takes secret $\mathcal{S}$ and $E_\Sigma$ to output a final result, the occurrences 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 $\chi$ in the records, he prepares a query $Q$, which is – as we will see later – simply a fixed-length sequence of encrypted values generated from the user-defined plaintext query. $U$ then sends $Q$ to the cloud, and the cloud processes $Q$. Finally, the cloud sends a result $E_\Sigma$ back to $U$ who can decrypt this result and learn the number of occurrences of $\chi$. The idea of a performing the counting in the cloud is to put the main computational burden on the cloud side. Both storage and computational overhead for KEYGEN, ENCRYT, UPLOAD, PREPAREQUERY, and DECODE should be lightweight compared to PROCESSQUERY. The definition's non-countable *random data*, e.g., an image, can be IND-CPA (AES-CBC) encrypted. Therewith, it is of no importance for privacy below.

### 2.2 Privacy

In the face of a 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.) *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 counts 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 $\kappa$.

**Definition 2 (Storage privacy).** *Let $\mathrm{bit}(j, R_{i,k})$ be the $j$-th bit of the $k$-th field of a record $R_i$. A challenger generates two same-size same-field-types sets of records $\{R_i^{(0)}\}$, $\{R_i^{(1)}\}$ and two patterns $\{\chi_0, \chi_1\}$. The challenger then uses* ENCRYPT *and* PREPAREQUERY *to compute the encrypted sets of records $\mathcal{E}_0, \mathcal{E}_1$ and two encrypted counting queries $Q_{\chi_0}, Q_{\chi_1}$. Using* PROCESSQUERY*, he evaluates $\mathcal{E}_0$ with $Q_{\chi_0}$ and $\mathcal{E}_1$ with $Q_{\chi_1}$ to get encrypted results $E_{\Sigma_{\chi_0}}, E_{\Sigma_{\chi_1}}$. The challenger sends $I := \{\mathcal{E}_0, \mathcal{E}_1, Q_{\chi_0}, Q_{\chi_1}, E_{\Sigma_{\chi_0}}, E_{\Sigma_{\chi_1}}\}$ to adversary $\mathcal{A}$. A protocol preserves storage privacy,* iff *for any probabilistic polynomial time (PPT) algorithm $\mathcal{A}$, the probability of correctly determining whether any bit of any $R_i$ in any data set is equal to any other bit is not higher than a random guess. That is, $\forall \chi_0, \chi_1, 0 \leq i_0, i_1 \leq n, 1 \leq k_0, k_1 \leq m, 0 \leq j_0 \leq \|\mathcal{D}_{k_0}\|, 0 \leq j_1 \leq \|\mathcal{D}_{k_1}\|, b_0, b_1 \in \{0,1\}$:*

$$\left| Pr[\mathcal{A}(I) \text{ determines } \text{``}\mathrm{bit}(j_0, R^{(b_0)}_{i_0,k_0}) = \mathrm{bit}(j_1, R^{(b_1)}_{i_1,k_1})\text{''}] - \frac{1}{2} \right| \leq \epsilon(\kappa).$$

**Definition 3 (Counting privacy).** *A challenger generates two same-size same-field-types sets of records, and two patterns $\chi_0, \chi_1, Pr[\chi_0 = \chi_1] = \frac{1}{2}$, uses* ENCRYPT*,* PREPAREQUERY*, and* PROCESSQUERY*, and sends encrypted $I := \{\mathcal{E}_0, \mathcal{E}_1, Q_{\chi_0}, Q_{\chi_1},$

$E_{\Sigma_{\chi_0}}, E_{\Sigma_{\chi_1}}\}$*, to* $\mathcal{A}$*. Now,* $\mathcal{A}$ *outputs whether* $\chi_0$ *equals* $\chi_1$*, i.e.,* $\chi_0 \stackrel{?}{=} \chi_1$*. A protocol preserves counting privacy,* iff *for any PPT algorithm* $\mathcal{A}$ *the probability of outputting correctly is not better than a random guess:*

$$\left| Pr[\mathcal{A}(I) \text{ outputs } \chi_0 \stackrel{?}{=} \chi_1 \text{ correctly }] - \frac{1}{2} \right| \leq \epsilon(k).$$

With *storage privacy*, we capture the intuition that, by storing data and counting, the cloud should not learn any information about the content it stores. In addition, *counting privacy* captures the problem that, again by storing data and counting, the cloud should not learn any details about the counting performed, e.g., which value is counted, whether a value 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 [9].

In the MapReduce framework, a user uploads his data into the cloud. During upload, data is automatically split into pieces (*InputSplits*) and distributed among the nodes in the cloud's data center. If the user wants the cloud to perform an operation on the outsourced data, he uploads an implementation of his operation, e.g., Java .class files, to the cloud. More precisely, the user has to provide implementations of two functions, the so called *map* function and the *reduce* function – these two functions will be executed by the cloud on the user's data.

A MapReduce "job" runs in two phases. Nodes in the data center storing an InputSplit (*Mapper* nodes) scan through their InputSplit and evaluate the user's map function on data. This operation is performed by all Mappers in parallel. The output of each Map function evaluation is a set of key-value pairs. All key values pairs are sent to so called *Reducer* nodes. The Reducer nodes collect key-value pairs emitted by Mappers and aggregate them using the user provided reduce function. Reducers produce 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

Before presenting EPiC, we briefly discuss why possible straightforward solutions do not work in our particular application scenario. This motivates the need for more sophisticated solutions such as EPiC.

*Precomputed Counters:* One could imagine that the cloud user, in the purpose of counting a value $\chi_k$ in a single countable field $\mathcal{D}_k$, simply stores encrypted counters

for each possible value of $\chi_k$ 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 counter values. Moreover, in the case of more complex queries involving multiple countable fields, all possible combinations of counters would 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, to enable counting for a single countable field $\mathcal{D}_k$, 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 LSB of the according subset is set to 1. The cloud only sums the encrypted voting fields (using additive homomorphic encryption) for all records. Again, such an approach does not allow flexible queries. To support counting on $m$ fields, the user would need to compute $2^m$ counters for each record ($2^m$ combinations for $m$ fields) when adding, removing, or modifying a record. This puts a high burden on the user. In conclusion, these straightforward solutions require heavy user-side computation, bad for light-weight client such as mobile devices. Moreover, they do not provide efficient, practical, and flexible counting solutions for multi-user, multiple field data sets.

### 3.1 EPiC Overview

For ease of exposition, we first introduce EPiC for the simpler case of counting on only one field ($m = 1, \mathcal{D} = \mathcal{D}_1$) using univariate polynomial evaluation. Later, we extend the technique to support counting on Boolean combinations of multiple fields $\mathcal{D}_1, \ldots, \mathcal{D}_m$ based on multivariate polynomials.

EPiC's main rationale is to perform the counting in the cloud by evaluating an indicator polynomial, as query $Q$, specific to the value $\chi$ the cloud user $\mathcal{U}$ is interested in. Conceptually, the cloud evaluates $P_\chi(x)$ on the value stored in the single countable field 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 $\chi$ in the investigated set of records.

### 3.2 Counting on a single field

If $\mathcal{U}$ wishes to count occurrences of $\chi$ in the single-field data set in an oblivious manner, the idea is to prepare a univariate *indicator polynomial* $P_\chi(x) = \begin{cases} 1, \text{ if } x = \chi \\ 0, \text{ otherwise} \end{cases}$ and scan through the data set $\mathcal{R} = \{R_1, \ldots, R_n\}$ of all patient records to compute the sum $\sum_x P_\chi(x)$. The result is the number of occurrences of $\chi$ in the data set. One way to generate a polynomial $P_\chi(x)$ is to construct the interpolation polynomial in the Lagrange form $P_\chi(x) := \sum_{j=0}^{|\mathcal{D}|-1} a_j \cdot x^j := \prod_{\alpha \in \mathcal{D}, \alpha \neq \chi} \frac{x-\alpha}{\chi-\alpha}$, where $\alpha$ are all possible values in $\mathcal{D}$ except $\chi$. The polynomial $P_\chi(x)$ is of degree $|\mathcal{D}| - 1$, and its coefficients $a_j$ are uniquely determined based on $\chi$.

In EPiC's settings, each countable value $R_{i,k}$ is encrypted to $E_{R_{i,k}}$ (in single-field case, $k = 1$). User $\mathcal{U}$ prepares the indicator polynomial based on $\chi$ and encrypts coefficients $a_j$ to $E_{a_j}$ then sends them to the cloud, which now computes the encrypted sum $E_\Sigma := \sum_{i=1}^{n} P_\chi(E_{R_{i,1}}) = \sum_{i=1}^{n} \sum_{j=0}^{|\mathcal{D}|-1} E_{a_j} \cdot (E_{R_{i,1}})^j$. In order for the cloud to compute $E_\Sigma$, additively and multiplicatively homomorphic properties are required for the encryption, which we describe in Section 3.5. Finally, $\mathcal{U}$ simply receives back $E_\Sigma$ 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 $n \cdot |\mathcal{D}|$ additions, $n \cdot |\mathcal{D}|$ multiplications of coefficients $E_{a_j}$ and monomials $E^j_{R_{i,1}}$, and $n \cdot |\mathcal{D}|$ exponentiations. Note that in the case of a small domain, i.e., $|\mathcal{D}| = 2$, there are no exponentiations, and multiplications dominate the computational costs of this approach. We can furthermore improve efficiency by rearranging the order of computations as follows:

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

Therewith, only $|\mathcal{D}|$ multiplications are required. We will apply this technique also to multiple field counting later, significantly improving performance.

**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 $|\mathcal{D}|$ 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. That is, each (patient) record can contain multiple countable fields, such as $R_{i,1}, R_{i,2}, \ldots, R_{i,m}$. 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 [5] or Shamir [23].

**Conjunctive counting:** Assume cloud user $\mathcal{U}$ is interested in counting the number of records that have their countable fields set to the pattern $\chi = (\chi_1, \ldots, \chi_m)$. Here, $\chi_k$, $1 \leq k \leq m$, 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 $P_\varphi(\hat{x}) = \prod_{k=1}^{m} P_{\chi_k}(x_k)$, where $\hat{x} = (x_1, \ldots, x_m)$ denotes the variables in the multivariate polynomial $P_\varphi(\hat{x})$, and $P_{\chi_k}(x_k)$ is the univariate indicator polynomial of the single $k$-th field for counting the single value $\chi_k$ as defined in Section 3.2. Therewith, $P_\varphi(\hat{x})$ is the multivariate indicator polynomial for the desired pattern $\chi$. Note that the size of the multi-domain $\mathcal{D}$ is $|\mathcal{D}| = \prod_{k=1}^{m} |\mathcal{D}_k|$, and the degree of $P_\varphi(\hat{x})$ 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 $\chi_1$ in $\mathcal{D}_1$ **or** value $\chi_2$ in $\mathcal{D}_2$. The multivariate indicator polynomial for this disjunction is $P_{\chi_1 \vee \chi_2}(\hat{x}) = P_{\chi_1}(x_1) + P_{\chi_2}(x_2) - P_{\chi_1 \wedge \chi_2}(\hat{x})$, where $P_{\chi_1}(x_1), P_{\chi_2}(x_2)$ are univariate indicator polynomials for $\mathcal{D}_1, \mathcal{D}_2$, respectively, and $P_{\chi_1 \wedge \chi_2}(\hat{x})$ 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.

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

**Integer range counting:** Assume $U$ wants to count records having a single 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 $P_{[a,b]}(x_k) = \sum_{\chi_k=a}^{b} P_{\chi_k}(x_k)$.

In summary, EPiC's oblivious counting technique based on indicator polynomial allows counting with respect to any Boolean formula on multiple fields. Users can construct complex queries such as integer comparisons, e.g., $P_{\chi_k \leq a}(x_k) = P_{[0,a]}(x_k)$, or $P_{\chi_k > a}(x_k) = P_{[a+1,|\mathcal{D}_k|-1]}(x_k)$. Our initial motivating example of Section 1 can be expressed as: $P_{age \in [50,70] \ and \ (dia=1 \ or \ hyp=1)}(\hat{x}) = \left(\sum_{age=50}^{70} P_{age}(\hat{x})\right) \cdot (P_{dia=1}(\hat{x}) + P_{hyp=1}(\hat{x}) - P_{dia=1}(\hat{x}) \cdot P_{hyp=1}(\hat{x}))$.

Although the user-defined queries are different in construction and length, the encrypted queries $Q$ always have exactly $|\mathcal{D}| = \prod_{k=1}^{m} |\mathcal{D}_k|$ encrypted coefficients (including 0-coefficients). As in the case of single-fields, this prevents the cloud to differentiate queries based on query sizes. To improve performance, we also apply the computation method of (1) for the sequence $\hat{E}(R_i) = (E_{R_{i,1}}, \ldots, E_{R_{i,m}})$ of encrypted fields and coefficients $a_{\hat{j}}$ corresponding to monomials $x_1^{j_1} \cdot x_2^{j_2} \cdots x_m^{j_m}$:

$$E_{\Sigma} = \sum_{i=1}^{n} P_{\chi}(\hat{E}(R_i)) = \sum_{\hat{j} \in \mathcal{D}} (E_{a_{\hat{j}}} \cdot \sum_{i=1}^{n} \prod_{k=1}^{m} (E_{R_{i,k}})^{j_k}), \hat{j} = (j_1, \ldots, j_m) \in \mathcal{D}, \quad (2)$$

### 3.4 Optimization through arithmetization in $GF(2)$

EPiC's efficiency relies on the computations performed by the cloud. As discussed above, there are *no exponentiations* required for counting on a binary field. Consequently, we convert "generic" fields (of large domain size) 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 $\mathcal{D}_{k,1}$, $\mathcal{D}_{k,2}$, ..., $\mathcal{D}_{k,\|\mathcal{D}_k\|}$. 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 $\chi_k = (\chi_{k,1}, \ldots, \chi_{k,\|\mathcal{D}_k\|})$ in field $D_k$ 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 $x_{k,l}$ indicates the $l$-th bit in $\mathcal{D}_k$, 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, 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 (denoted as $\#$) of the GF(2) arithmetized multivariate indicator polynomial corresponding to each query remains the same as in the generic case, i.e., $\#_G = |\mathcal{D}^{(G)}| = 2^{\|\mathcal{D}^{(G)}\|} = 2^{\sum_{k=1}^{m} \|\mathcal{D}_k\|} = \prod_{k=1}^{m} |\mathcal{D}_k| = |\mathcal{D}^{(B)}| = \#_B$, the (multivariate) degree of the GF(2) arithmetized polynomial is much lower at $\deg(P_G) = \sum_{k=1}^{m} \|\mathcal{D}_k\| < \sum_{k=1}^{m}(|\mathcal{D}_k|-1) = \deg(P_B)$. Our "transformation" into multiple binary fields and resulting lower degree polynomials reduces computational costs on the cloud significantly. 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 [8, 10, 17, 29], 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. This scheme does not enjoy the same properties as related work, but is especially practical in the setting we target.

**EPiC's somewhat homomorphic encryption:** We design a somewhat homomorphic encryption scheme derived from the computational Private Information Retrieval (cPIR) technique of Trostle and Parrish [27]. 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. EPiC's somewhat homomorphic encryption is defined by the following set of operations.

- $\textsc{KeyGen}(s_1, s_2, n, |\mathcal{D}|)$: 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 $|\mathcal{D}|$ is the multi-domain size of $m$ countable fields. $\textsc{KeyGen}$ computes 1.) a random prime $q$ such that $q > n$, 2.) a random prime $p$ where $\|p\| \geq s_1 + \|n\| + \|q\| + \sum_{k=1}^{m}(s_2 + \|q\|) \cdot (|\mathcal{D}_k| - 1)$, and 3.) a random $b \in \mathbb{Z}_p$. We discuss the selection of $q$ and $p$ further below. The secret key, the output of $\textsc{KeyGen}$, is defined as $\mathcal{S} := \{p, b\}$.
- $\textsc{Enc}(\mathcal{P})$: Selects random number $r$, $\|r\| \leq s_2$, and encrypts the plaintext $\mathcal{P}$ to $\mathcal{C} = \textsc{Enc}(\mathcal{P}) := b \cdot (r \cdot q + \mathcal{P}) \bmod p$. In the rest of the paper, we instead use the following shorthand: $\mathcal{C} = \textsc{Enc}(\mathcal{P}) := b \cdot e(\mathcal{P}) \bmod p$, where $e(x) := r \cdot q + x$.
- $\textsc{Dec}(\mathcal{C})$: Decrypts $\mathcal{C}$ by computing $\mathcal{P} = \textsc{Dec}(\mathcal{C}) := b^{-1} \cdot \mathcal{C} \bmod p \bmod q$.

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 requires knowing 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.

**Encrypting data and query:** We employ this encryption scheme to encrypt both countable fields and query. For the privacy requirements (discussed later), we use different secret keys for data fields and query. Each countable field is encrypted using the secret key $\mathcal{S} = \{p, b\}$ mentioned above: $E_{R_{i,k}} = \textsc{Enc}(\mathcal{S}, R_{i,k})$. The encryption of a counting query $Q$, as a sequence of encrypted coefficients $E_{a_{\hat{j}}}$, however, is more complicated.

For each new counting query $Q$, the user $\mathcal{U}$ generates a random number $b' \in GF(p)$ to create a new secret key $\mathcal{S}' = \{p, b'\}$. A simple method of encrypting $a_{\hat{j}}$ to $E_{a_{\hat{j}}} = \textsc{Enc}(\mathcal{S}', a_{\hat{j}})$ does not allow the decryption of the total sum $E_{\Sigma}$ due to different exponents of $b$ in the multivariate monomials, explained below.

Consider the cloud computation in (2). Every $\hat{j}$-th, $\hat{j} = (j_1, \ldots, j_m)$, multivariate monomial $\prod_{k=1}^{m}(E_{R_{i,k}})^{j_k}$ has a multi-degree that is dependent on $\hat{j}$. More precisely, $\deg \prod_{k=1}^{m}(E_{R_{i,k}})^{j_k} = \sum_{k=1}^{m} j_k$, denoted as $d_1(\hat{j})$. If $a_{\hat{j}}$ was simply encrypted to $E_{a_{\hat{j}}} = \textsc{Enc}(\mathcal{S}', a_{\hat{j}})$, each product $E_{a_{\hat{j}}} \cdot \sum_{i=1}^{n}\prod_{k=1}^{m}(E_{R_{i,k}})^{j_k}$ would contain $b$ of different degree $d_1(\hat{j})$. This would prohibit additions among them to obtain the *decryptable* final sum $E_{\Sigma}$. Our approach is to "augment" the exponents of $b$ in the coefficients. Specifically, we encrypt $a_{\hat{j}}$ to

$$E_{a_{\hat{j}}} := \textsc{Enc}(\mathcal{S}', a_{\hat{j}}) \cdot (\textsc{Enc}(\mathcal{S}, 1))^{d_2(\hat{j})} = b' \cdot b^{d_2(\hat{j})} \cdot e(a_{\hat{j}}) \bmod p, \tag{3}$$

where $d_2(\hat{j}) = \sum_{k=1}^{m}(|\mathcal{D}_k| - j_k - 1)$. Therewith, the final sum $E_{\Sigma}$ will contain $b'$ of degree 1, and $b$ of degree $d = d_1(\hat{j}) + d_2(\hat{j}) = \sum_{k=1}^{m}(|\mathcal{D}_k| - 1)$, which is independent of $\hat{j}$. All multivariate monomials now have the same exponents of $b$, allowing successful decryption of $E_{\Sigma}$.

**Decryption and requirements:** Consider a ciphertext $\mathcal{C} = \textsc{Enc}(\mathcal{P}) = b \cdot e(\mathcal{P}) \bmod p$ and its decryption procedure: $\textsc{Dec}(\mathcal{C}) = b^{-1} \cdot \mathcal{C} \bmod p \bmod q \overset{(1)}{=} e(\mathcal{P}) \bmod p \bmod q \overset{(2)}{=} e(\mathcal{P}) \bmod q \overset{(3)}{=} \mathcal{P}$. For this to hold, we need: (1) $b^{-1} \in GF(p)$; (2) $e(\mathcal{P}) \in GF(p)$; (3) $\mathcal{P} \in GF(q)$. While the first condition automatically holds for prime $p$, the other conditions need careful selection of $p$ and $q$. In EPiC, the user $\mathcal{U}$ receives the encrypted final sum $E_{\Sigma}$ from the cloud. The last two conditions apply to the decryption of $E_{\Sigma}$ as follows:

1. $\sigma \in GF(q) \Longleftrightarrow n < q$.
2. $e(\sigma) \in GF(p) \Longleftrightarrow \sum_{\hat{j} \in \mathcal{D}}(e(a_{\hat{j}}) \cdot \sum_{i=1}^{n}\prod_{k=1}^{m}(e(R_{i,k}))^{j_k}) < p$. The prime $p$ must be chosen such that it satisfies the last inequality for any value of $e(a_{\hat{j}})$ and $e(R_{i,k})$. Manipulating the last inequality, we can establish the lower bound on the size of $p$: $\|p\| \geq \|n\| + \|q\| + \sum_{k=1}^{m}(s_2 + \|q\|) \cdot (|\mathcal{D}_k| - 1)$. As our scheme relies

on the Hidden Modular Group Order assumption, a security parameter $s_1$ has to be added to the size of $p$ (see [27] for more details):

$$\|p\| \geq s_1 + \|n\| + \|q\| + \sum_{k=1}^{m} (s_2 + \|q\|) \cdot (|\mathcal{D}_k| - 1). \quad (4)$$

### 3.6 Detailed Protocol Description

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

$\textsc{KeyGen}(s)$ Based on security parameter $\kappa$, cloud user $\mathcal{U}$ chooses $s_1, s_2$ for the somewhat homomorphic encryption together with a symmetric key $K$ for a block cipher such as AES. $\mathcal{U}$ also computes $\textsc{KeyGen}(s_1, s_2, n, |\mathcal{D}|)$ for the somewhat homomorphic encryption, determining an upper bound $n$ for the total number of patient records that might be stored and a value for the domain $\mathcal{D}$ of the countable field. Secret key $\mathcal{S}$ is the output of the somewhat homomorphic encryption $\textsc{KeyGen}$ and $K$, i.e., $\mathcal{S} := \{p, b, K\}$.

$\textsc{Encrypt}(\mathcal{S}, \mathcal{R})$ Assume $\mathcal{U}$ wants to store $n$ patient 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. $\mathcal{E} := \{E_{R_1}, \ldots, E_{R_n}\}$.

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

$\textsc{PrepareQuery}(\mathcal{S}, \chi)$ To prepare a query for $\chi$, $\mathcal{U}$ computes the $|\mathcal{D}|$ coefficients $a_{\hat{j}}$, $\hat{j} \in \mathcal{D}$, of the indicator polynomial $P_\chi(\hat{x})$ as described in Section 3.3. A new secret key $\mathcal{S}' = \{p, b'\}$ is generated (Section 3.5) and associated to the query. Coefficients $a_{\hat{j}}$ are encrypted according to Equation (3), using the new secret key $\mathcal{S}'$, and sent to the cloud. The cloud will be using these coefficients to perform the evaluation of $P_\chi(\hat{x})$. Consequently in EPiC, the output $Q$ of $\textsc{PrepareQuery}$ sent to the cloud is $Q := \{E_{a_{\hat{j}}}, \hat{j} \in \mathcal{D}\}$.

$\textsc{ProcessQuery}(Q, \mathcal{E})$ Based on the data set size and the cloud configuration, the MapReduce framework has selected $M$ Mapper nodes and 1 Reducer node. Each Mapper node stores one InputSplit.

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 in their locally stored InputSplits, the Mappers compute in parallel all monomials of the countable fields and add the same-degree monomials together. After the Mappers finish scanning over all records in their InputSplits, the sums of monomials are output as key-value pairs. These pairs contain the multi-degree $\hat{j}$ as key, and the computed sum $s_{\hat{j}}$ as value. In MapReduce, output of the Mappers is then automatically sent to the Reducer ("emit"). The sums $s_{\hat{j}}$ emitted by each Mapper are taken over records in the InputSplit corresponding to that Mapper only. The Reducer, therefore, based on the sums received

**Algorithm 1**: PROCESSQUERY

*For each Mapper $M$:*

```
init s_ĵ := 0, ∀ĵ ∈ D
forall E_{R_i} in InputSplit(M) do
    read {E_{R_{i,1}}, ..., E_{R_{i,m}}}
    forall ĵ = (j_1, ..., j_k) ∈ D do
        s_ĵ := s_ĵ + ∏_{k=1}^{m} (E_{R_{i,k}})^{j_k}
    end
end
emit {ĵ, s_ĵ}, ∀ĵ ∈ D
```

*Reducer $R$:*

```
init E_Σ := 0, S_ĵ := 0, ∀ĵ ∈ D
forall {ĵ, s_ĵ} in MappersOutput do
    S_ĵ := S_ĵ + s_ĵ
end
forall ĵ in D do
    E_Σ := E_Σ + E_{a_ĵ} · S_ĵ
end
write {E_Σ}
```

from all Mappers, combines them together to obtain the *global* sums, i.e., the sums over all records in the data set. In a last step, the Reducer uses the coefficients received from $\mathcal{U}$ to evaluate the polynomial by computing the inner product with the global sums. The result is sent back to $\mathcal{U}$ and can be decrypted to obtain the count value.

DECODE$(\mathcal{S}, E_\Sigma)$ Cloud user $\mathcal{U}$ receives $E_\Sigma$ and computes $\sigma = \text{DEC}(E_\Sigma, d, \mathcal{S}, \mathcal{S}') := b'^{-1} \cdot b^{-d} \cdot E_\Sigma \bmod p \bmod q$, where $d = \sum_{k=1}^{m}(|\mathcal{D}_k| - 1)$.

**Privacy:** Due to space constraints, EPiC's formal proofs can be found in Appendix B.

# 4 Evaluation

To show its real-world applicability, we have implemented and evaluated EPiC in the Hadoop MapReduce framework v1.0.3 [4]. The source code is available for download [6]. 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 $< 2$ms on a 1.8GHz Intel Core i7 laptop, a single addition is not measurable with $< 1\mu$s. Figure 3 shows a benchmark of various operations on the ciphertexts using our encryption scheme. We would like to stress that the exponentiation, the most critical operation in our scenario, is two orders of magnitude faster than the one by Lauter et al. [17] – there, a single multiplication already consumes 40ms on a stronger CPU with 2.1GHz. In our evaluation, we use security parameters $s_1 = 400$ bits as suggested by Trostle and Parrish [27] 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. As our evaluation targets comparing the performance of EPiC counting to non-private counting, we use encryption and decryption on the countable fields only. Other fields are not considered during the evaluation.

In this section, we evaluate the performance of EPiC by comparing our "Basic" and "GF(2) arithmetized" solutions with "non-privacy-perserving" 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 valid comparison. For shorter presenta-

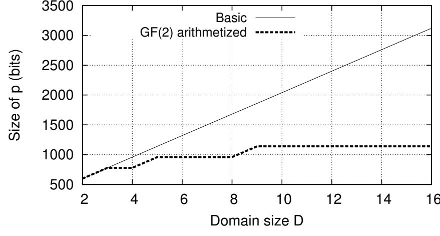



**Fig. 1.** Size of $p$ depends on size of domain $\mathcal{D}$

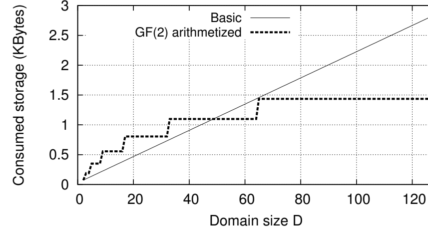



**Fig. 2.** Consumed storage for each field

tion, we use "B" as index of the cost in Basic approach, and "G" in GF(2) arithmetized approach. 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.

### 4.1 Size of prime $p$

As mentioned in Section 3.5, while the prime $q$ depends only on the number of records $n$, the prime $p$ also depends on $|\mathcal{D}|$. Derived from (4), we show the benefit of GF(2) arithmetized approach by demonstrating that a conversion to multiple binary fields reduces $\|p\|$ significantly: $\|p\|_B = u + (|\mathcal{D}| - 1) \cdot v$ and $\|p\|_G = u + \|\mathcal{D}\| \cdot v$.

Figure 1 shows the logarithmic increase of $\|p\|$ with GF(2) arithmetized approach and linear increase with Basic approach.

### 4.2 Storage cost

The storage cost depends on the size of the data stored on the cloud. The larger the data is, the more the user has to pay for to the cloud, e.g. Amazon S3 [3]. In the following, we only focus on the size of the countable fields, which is determined by the size of prime $p$. Other fields are not considered.

Since the encrypted value in a countable field is an element in $GF(p)$, its size is at most $\|p\|$. In Basic approach, each record stores $\|p\|_B$ bits for only one field. In GF(2) arithmetized approach, each record stores $\|\mathcal{D}\|$ binary fields, each of size $\|p\|_G$. Therefore, the required storage is $S_B = u + (|\mathcal{D}| - 1) \cdot v$ and $S_G = \|\mathcal{D}\| \cdot (u + \|\mathcal{D}\| \cdot v)$.

Again, in Figure 2, 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 bit fields.

### 4.3 User computation and communication cost

**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 1ms (Figure 3), 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 computation cost at the user side is negligible compared to the cloud cost.

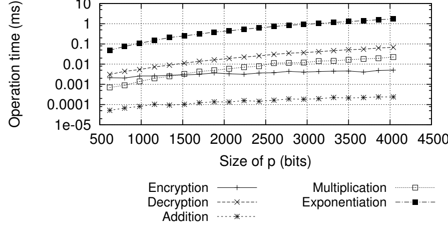



**Fig. 3.** Computation time on ciphertexts

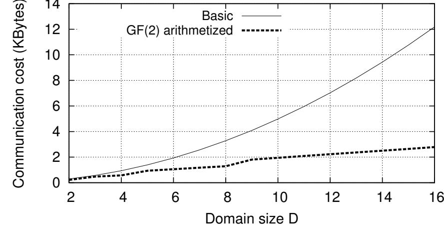



**Fig. 4.** Communication cost

**Communication cost**

*Sending query:* Due to oblivious counting, user $\mathcal{U}$ prepares and sends all $|\mathcal{D}|$ coefficients corresponding to *all* monomials to the cloud. The transferred size of the encrypted coefficients, however, is $|\mathcal{D}| \cdot \|p\|$, depending on the size of $p$. Consequently, the query size is $Q_B = |\mathcal{D}| \cdot (u + (|\mathcal{D}| - 1) \cdot v)$ and $Q_G = |\mathcal{D}| \cdot (u + \|\mathcal{D}\| \cdot v)$.

Example: for a data set containing $n = 10^6$ records with a countable field of domain size $|\mathcal{D}| = 1024$, the query size is $Q_B = 22.5$ MBytes, and $Q_G = 280$ KBytes.

*Receiving answer:* The size of the received ciphertext (final sum) depends on the maximum size of the multivariate monomial. The size of a monomial is determined by the number of performed multiplications, i.e., its multi-degree. Consequently, the answer size depends on the size of monomials of the maximum multi-degree, i.e., $|\mathcal{D}|$ in the Basic approach, and $\|\mathcal{D}\|$ 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 $A_G = \|\mathcal{D}\| \cdot \|p_G\| = \|\mathcal{D}\| \cdot (u + \|\mathcal{D}\| \cdot v)$.

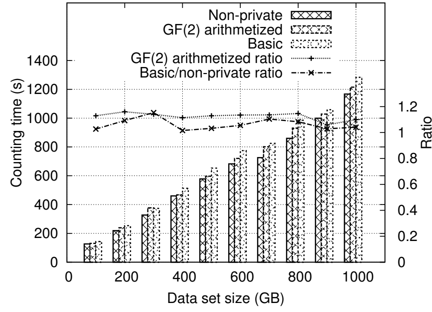
Example: for a data set containing $n = 10^6$ records with a countable field of domain size $|\mathcal{D}| = 1024$, the answer size is $A_B = 22.5$ Mbytes, and $A_G = 2.7$ KBytes.

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

### 4.4 Cloud computation

We evaluate 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 [2]. Each instance comprises 4 processors with 2.27GHz Intel Xeon CPU, 4MB of cache, and total 7.5GB of memory.

*Variable data set size:* Coming back to our example application scenario with patient records, in a first experiment, we fix the size of each record to 1MB. In the real world, this would reflect to a patient record that might, besides doctoral notes and prescription, also comprise, e.g., an X-ray picture. The data set size (x-axis) is varied from 100GB to 1TB. In this experiment, 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



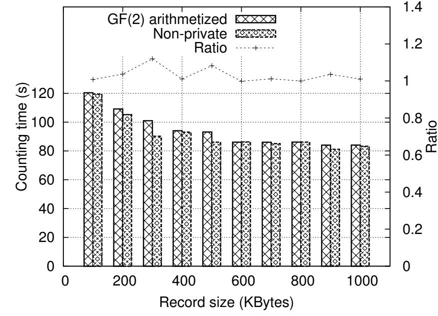
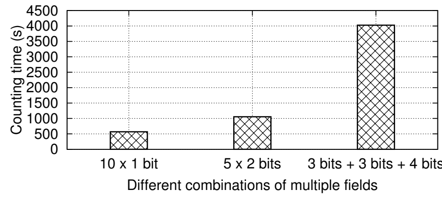




**Fig. 5.** Counting time vs. data set size. $|\mathcal{D}| = 16$.

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



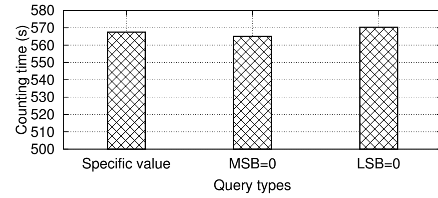




**Fig. 7.** Effect of different field combinations.

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

the time that a user has to pay for to, e.g., Amazon [2]. 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 $20\%$. As Amazon's pricing scales directly proportional with the CPU time. This additional overhead would also be the additional amount of money user $\mathcal{U}$ would have to pay Amazon. 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 50GB. The record size is changed from 100KB to 1MB. 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 100KB 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 subfields of different size, we conduct an experiment on a data set size of 100GB. The data set contains a countable field of domain size $|\mathcal{D}| = 1024$ (10 bits). We compare three cases: (a) transform $\mathcal{D}$ into 10 single bit fields; (b) transform $\mathcal{D}$ into 5 fields of 2 bits; (c) transform $\mathcal{D}$ into 3 fields of 3 bits, 3 bits, and 4 bits. 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 [28] for an overview), current implementations indicate high storage and computational overhead [11, 20], rendering fully homomorphic encryption impractical for the cloud.

Similar to EPiC, Lauter et al. [17] observe that, depending on the application, weaker "somewhat" homomorphic encryption might be sufficient. Lauter et al. [17]'s scheme is based on a protocol for lattice-based cryptography by Brakerski and Vaikuntanathan [8]. However, for the 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 [16] 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.

Other research has addressed similar problems of performing operations on outsourced data, such as privacy-preserving searching on encrypted data [7, 19, 24, 30]. While searching and counting might be 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. [7] rely on the computation of very expensive bilinear pairings for each element of a data set, rendering this approach impractical in a cloud setting.

## 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] Amazon. Amazon Elastic MapReduce Price, 2012. http://aws.amazon.com/elasticmapreduce/#pricing.

[3] Amazon. Amazon Simple Storage Service (Amazon S3) Price, 2012. http://aws.amazon.com/s3/#pricing.

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

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

[6] E-O. Blass, G. Noubir, and T. Vo-Huu. EPiC Source Code, 2013. http://www.ccs.neu.edu/home/vohuudtr/projects/epic/epic.tar.gz.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] Google. A new approach to China, 2010. http://googleblog.blogspot.com/2010/01/new-approach-to-china.html.

[13] Google. Google App Engine API for running MapReduce jobs, 2011. http://code.google.com/p/appengine-mapreduce/.

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

[15] IBM. InfoSphere BigInsights, 2011. http://www-01.ibm.com/software/data/infosphere/biginsights/.

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

[17] 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.

[18] Microsoft. Daytona, 2011. http://research.microsoft.com/en-us/projects/daytona/.

[19] V. Pappas, M. Raykova, B. Vo, S.M. Bellovin, and T. Malkin. Private Search in the Real World. In *Proceedings of Computer Security Applications Conference*, pages 83–92, Orlando, USA, 2011. ISBN 978-1-4503-0672-0.

[20] 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.

[21] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *Proceedings of ACM Conference on Computer and Communications Security*, pages 199–212, Chicago, USA, 2009. ISBN 978-1-60558-894-0.

[22] F. Rocha and M. Correia. Lucy in the Sky without Diamonds: Stealing Confidential Data in the Cloud. In *Proceedings of International Workshop on Dependability of Clouds, Data Centers and Virtual Computing y Environments*, Hong Kong, China, 2011. http://www.gsd.inesc-id.pt/~mpc/pubs/diamonds-final.pdf.

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

[24] 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.

[25] Techcrunch. Google Confirms That It Fired Engineer For Breaking Internal Privacy Policies, 2010.
http://techcrunch.com/2010/09/14/google-engineer-spying-fired/.
[26] 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.
[27] 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.
[28] 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.
[29] 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.
[30] C. Wang, K. Ren, S. Yu, and K.M.R. Urs. Achieving Usable and Privacy-assured Similarity Search over Outsourced Cloud Data. In *Proceedings of International Conference on Computer Communications*, pages 451–459, Orlando, USA, 2012. ISBN 978-1-4673-0773-4.
[31] Z. Whittaker. Microsoft admits Patriot Act can access EU-based cloud data. Zdnet, 2011.
http://www.zdnet.com/.

# A Somewhat homomorphic properties

**Additively homomorphic:** $\textsc{Dec}(\mathcal{C}_1^j + \mathcal{C}_2^j, j)$

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

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

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

**Ciphertext size:** As multiplications and additions are performed on the integers without modulo operations, the size of result increases after the 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: $\|\mathcal{C}^j\| = j \cdot \|\mathcal{C}\| - (j - 1)$.

# B Privacy

**Lemma 1 (Storage privacy).** *Based on the security of the cPIR scheme by Trostle and Parrish [27], EPiC preserves storage privacy.*

*Proof (Sketch).*

PIR security of Trostle and Parrish [27] can be summarized as follows: an adversary receives an encrypted PIR request $Q_{PIR} = (E_{v_1}, \ldots, E_{v_u})$ from a challenger, where $v_i = 1$, *iff* the $i$-th row is requested from a $u \times u$ bit database, and $v_i = 0$, otherwise. Informally, the encrypted $E_{v_i}$ are computationally indistinguishable from random for the adversary.

We now prove our lemma for a single-field data set. We show that any PPT $(t, \epsilon)$-adversary $\mathcal{A}$ breaking EPiC's storage privacy (Definition 2) in $t$ steps with non-negligible advantage $\epsilon$ can be used to construct an $(t', \epsilon')$-adversary $\mathcal{A}'$ as a subroutine breaking the cPIR protocol by Trostle and Parrish [27].

The proof is by straightforward reduction, and we construct $\mathcal{A}'$ as follows. $\mathcal{A}'$ receives $Q_{PIR} = (E_{v_1}, \ldots, E_{v_u})$ from a challenger. $\mathcal{A}'$ splits $Q_{PIR}$ into two halves $\mathcal{E}_0, \mathcal{E}_1$, i.e., treating the PIR request as two encrypted data sets of the same size and same field types. If $u$ is odd, $\mathcal{A}'$ removes an arbitrary element and splits. Since $v_i$ are binary, $\mathcal{E}_0, \mathcal{E}_1$ are data sets containing only one binary field. $\mathcal{A}'$ randomly selects $l_1, l_2, l_3, l_4$ and creates two counting queries $Q_{\chi_0} = \{E_{v_{l_1}}, E_{v_{l_2}}\}, Q_{\chi_1} = \{E_{v_{l_3}}, E_{v_{l_4}}\}$. These are two valid queries, because for a binary field, any query contains only 2 coefficients. The queries correspond to two arbitrary patterns $\chi_0, \chi_1$. Then $\mathcal{A}'$ runs PROCESSQUERY on $\mathcal{E}_0$ with $Q_{\chi_0}$, and $\mathcal{E}_1$ with $Q_{\chi_1}$, thereby obtaining $E_{\Sigma_{\chi_0}}$ and $E_{\Sigma_{\chi_1}}$. Now, $\mathcal{A}'$ forwards $\{\mathcal{E}_0, \mathcal{E}_1, Q_{\chi_0}, Q_{\chi_1}, E_{\Sigma_{\chi_0}}, E_{\Sigma_{\chi_1}}\}$ to $\mathcal{A}$. This adversary determines in $t$ steps whether $\mathrm{bit}(j_0, R^{(b_0)}_{i_0,1})$ and $\mathrm{bit}(j_1, R^{(b_1)}_{i_1,1})$ in $\mathcal{E}$ are different, i.e., distinguishes any two bits in $Q_{PIR}$, with advantage $\epsilon$. Consequently, $\mathcal{A}'$ automatically breaks the PIR security in $t' = t$ steps with advantage $\epsilon' = \epsilon$, rendering our reduction tight.

The proof for storage privacy in multiple field data sets is identical, if we view the PIR request as a concatenation of many single-field data sets, i.e., each subset of the PIR request is a single-field data set. □

**Lemma 2 (Counting privacy).** *Based on the security of the cPIR scheme by Trostle and Parrish [27], EPiC preserves counting privacy.*

*Proof (Counting privacy).*

As of Lemma 1, $\{\mathcal{E}_0, \mathcal{E}_1\}$ does not provide any information for a PPT $(t, \epsilon)$-adversary $\mathcal{A}$ that distinguishes queries in $t$ steps with non-negligible probability advantage $\epsilon$. We prove counting privacy again by reduction of the security of the cPIR protocol by [27]. Assume the existence of a PPT $(t, \epsilon)$-$\mathcal{A}$ above that returns "equal" or "different". We will use $\mathcal{A}$ as a subroutine to construct a new $(t', \epsilon')$-adversary $\mathcal{A}'$ against PIR.

Consider a PIR setting, where a challenger wants to retrieve the $i_0$-th row from a $u \times u$ database. Let $q$ be a prime greater than 2. The challenger prepares a PIR request $Q_{PIR} = (E_{v_1}, \ldots, E_{v_u})$, where $E_{v_{i_0}} = b \cdot (r_{v_{i_0}} \cdot q + 1) \bmod p$ for the requested row, and $E_{v_j} = b \cdot r_{v_j} \cdot q \bmod p$ for other rows, $j \neq i_0$, $r$ are random numbers generated for every $E_{v_j}$. The challenger sends $Q_{PIR}$ to $\mathcal{A}'$, proceeding as follows:

1. $\mathcal{A}'$ creates superset $S = \{Q^{(j)}\}_{1 \leq j \leq u}$ of queries $Q^{(j)}$, where $Q^{(j)} = Q_{PIR} \backslash \{E_{v_j}\}$. So, $Q^{(j)}$ comprises all elements of $Q_{PIR}$ except element $v_j$. $\mathcal{A}'$ also creates an empty set $S_0$. Note that $Q^{(i_0)}$ contains only encryptions of 0. For each $j \neq i_0$, $Q^{(j)}$ contains $v_{i_0}$, an encryption of 1. $\mathcal{A}'$ wants to determine $i_0$.

2. $\mathcal{A}'$ selects any two subsets $Q^{(i)}, Q^{(j)}$ out of $S$, runs ProcessQuery and sends $\{\mathcal{E}_0, \mathcal{E}_1, Q^{(i)}, Q^{(j)}, E_{\Sigma_i}, E_{\Sigma_j}\}$ with randomly chosen values $\{\mathcal{E}_0, \mathcal{E}_1, E_{\Sigma_i}, E_{\Sigma_j}\}$ to $\mathcal{A}$. Note that due to Lemma 1, $\mathcal{A}$ cannot determine that $\{\mathcal{E}_0, \mathcal{E}_1, E_{\Sigma_i}, E_{\Sigma_j}\}$ are random and do not fit queries $Q^{(i)}, Q^{(j)}$. Otherwise, $\mathcal{A}$ would learn bits of $\mathcal{E}_0, \mathcal{E}_1$. Subsets $Q^{(i)}, Q^{(j)}$ are treated as two counting queries, each of which contains $u-1$ coefficients. These are valid queries for EPiC data sets with (multi-)domain size $|\mathcal{D}| = u - 1$.
3. If $\mathcal{A}$ returns "equal", $\mathcal{A}'$ adds indices $i$ and $j$ into $S_0$, i.e., $S_0 = S_0 \cup \{i, j\}$ and repeats at step 2 until $S$ is empty. However, if $\mathcal{A}$ returns "different", $\mathcal{A}'$ knows that one of them must be $Q^{(i_0)}$. $\mathcal{A}'$ then picks any $k$ from $S_0$ and sends $\{\mathcal{E}_0$, $\mathcal{E}_1$, $Q^{(i)}$, $Q^{(k)}$, $E_{\Sigma_i}$, $E_{\Sigma_k}\}$ to $\mathcal{A}$. If $\mathcal{A}$ returns "equal", $i_0 = j$, and $i_0 = k$, otherwise.

The above algorithm performs at most $u/2$ calls to $\mathcal{A}$ before it breaks PIR. Thus, $\mathcal{A}'$ is a $(t \cdot u/2, \epsilon)$-adversary against PIR. □