Practical Confidentiality Preserving Big Data Analysis [*]

Julian James Stephen Savvas Savvides Russell Seidel Patrick Eugster
Department of Computer Science, Purdue University

Abstract

The "pay-as-you-go" cloud computing model has strong potential for efficiently supporting big data analysis jobs expressed via data-flow languages such as Pig Latin. Due to security concerns — in particular leakage of data — government and enterprise institutions are however reluctant to moving data and corresponding computations to public clouds. We present Crypsis, a system that allows execution of MapReduce-style data analysis jobs directly on encrypted data. Crypsis transforms data analysis scripts written in Pig Latin so that they can be efficiently executed on encrypted data. Crypsis to that end employs existing practical partially homomorphic encryption schemes, and adopts a global perspective in that it can perform partial computations on the client side when PHE alone would fail. We outline the original program transformations underlying Crypsis for reducing the cost of data analysis computations in this larger perspective. We show practicality of our approach by evaluating Crypsis on standard benchmarks.

1 Introduction

Cloud computing offers an attractive paradigm for industry and government wanting to perform cost-efficient data analysis. However, in order to adopt the paradigm, sensitive data has to be moved to the cloud, and trust placed on the infrastructure provider to ensure data confidentiality. Even with a trusted provider, malicious users and programs and software defects can lead to data leaks.

The ability to maintain sensitive data only in an encrypted form in the cloud and still perform meaningful data analysis is of great value. Even within an enterprise, maintaining data in encrypted format helps prevent insider attacks and accidental leaks. As a consequence, *homomorphic encryption* has become the holy grail of cryptography. Despite advances occurring regularly (e.g. [12]), generic, *fully* homomorphic encryption (FHE) still exhibits prohibitive overheads. However, FHE may not be necessary for many real-life computations. Several existing cryptographic systems ("cryptosystems") are namely *partially* homomorphic, i.e., homomorphic with respect to certain operations. More precisely, if $E(x)$ and $D(x)$ denote the encryption and decryption functions for input data x respectively, a cryptosystem is said to be homomorphic with respect to operation χ iff $\exists\psi \mid D(E(x_1)\psi E(x_2)) = x_1 \chi x_2$. For example, when χ is $\times$, the cryptosystem is said to be multiplicative homomorphic (e.g., unpadded RSA [18], ElGamal [11]). Leveraging such existing homomorphic schemes however goes through addressing several challenges, especially in the context of big data analysis. Many of these jobs are typically expressed in *domain-specific* security-agnostic high-level data flow languages, like Pig Latin [15] or FlumeJava [6], which are compiled to *sequences* of several MapReduce tasks [8]. This makes the application of established techniques hard. Recent approaches to leveraging partially homomorphic encryption (PHE) for secure cloud-based computing target different application scenarios or work-loads. For intance, CryptDB [17] is based on the MySQL database server and does not enable parallelization through a MapReduce substrate; MrCrypt [23] only supports single MapReduce tasks.

This paper presents Crypsis, a runtime system for Pig Latin which allows corresponding scripts to be executed efficiently by exploiting cloud resources but without exposing input data in the clear. More specifically, Crypsis extends the scope of encryption-enabled big data analysis based on the following insights:

Extended program perspective: By *analyzing entire data flow programs*, Crypsis can identify many opportunities for operating in encrypted mode. For example, Crypsis can identify operations in Pig Latin scripts that

[*]Financially supported by DARPA grant # N11AP20014, Northrop Grumman Information Systems, Purdue Research Foundation grant # 204533, and Google Award "Geo-Distributed Big Data Processing".

are inter-dependent with respect to encryption, or inversely, independent of each other. More precisely, when applying two (or more) operations to a same data item, many times the second operation does not use any *side-effect* of the former, but operates on the original field value. Thus, *multiple encryptions* of a same field can support different operations by carefully handling relationships between such encryptions.

Extended system perspective: By considering the possibility of performing subcomputations on the client side, Crypsis can still exploit cloud resources rather than giving up and forcing users to run entire data flow programs in their limited local infrastructure, or defaulting to FHE (and then aborting [23]) when PHE does not suffice. For example, several programs in the PigMix (I+II) benchmarks [3] end up averaging over the values of a given attribute for several records after performing some initial filtering and computations. While the summation underlying averaging can be performed in the cloud via an *additive homomorphic encryption* (AHE) scheme, the subsequent division can be performed on the client side.

Considering the amount of data continuously decreases as computation advances in most analysis jobs, it makes sense to *compute as much as possible in the cloud*.

The contributions of this paper are as follows. After presenting background information (Section 2), we

1. propose an architecture for executing Pig Latin scripts in the cloud without sacrificing confidentiality of data (Section 3).
2. outline a novel field-sensitive program analysis and transformation for Pig Latin scripts that distinguishes between operations with side-effects (e.g., whose results are used to create new intermediate data) and without (e.g., filters). The results are semantically equivalent programs executable by Crypsis that maximize the amount of computations done on encrypted data in the cloud (Section 4).
3. present initial evaluation results for an implementation of our solution based on the runtime of Pig Latin scripts obtained from the open-source Apache Pig [2] PigMix benchmarks (Section 5).

Section 6 contrasts with related work. Section 7 concludes with final remarks.

2 Background

This section presents background information on homomorphic encryption and our target language Pig Latin.

2.1 Homomorphic Encryption

A cryptosystem is said to be homomorphic (with respect to certain operations) if it allows computations (consisting in such operations) on encrypted data. If $E(x)$ and $D(x)$ denote the encryption and decryption functions for input data x respectively, then a cryptosystem is said to be homomorphic with respect to addition if $\exists\psi$

$$D(E(x_1)\psi E(x_2)) = x_1 + x_2$$

Dually a cryptosystem is said to provide *additive homomorphic encryption* (AHE). Similarly a cryptosystem is said to be homomorphic with respect to multiplication or provide *multiplicative homomorphic encryption* (MHE) if $\exists\chi$

$$D(E(x_1)\chi E(x_2)) = x_1 \times x_2$$

Additional homomorphisms are with respect to operators such as "$\leq$" and "$\geq$" (*order-preserving encryption* – OPE) or equality comparison "=" (*deterministic* encryption – DET). *Randomized* (RAN) encryption does not support any operators, and is intuitively, the most desirable form of encryption because it does not allow an attacker to learn anything at all.

2.2 Pig

Apache Pig [2] is a data analysis platform which includes the Pig runtime system for the high-level data flow language Pig Latin [15]. Pig Latin expresses data analysis jobs as sequences of data transformations, and is compiled to MapReduce [8] tasks by Pig. The MapReduce tasks are then executed by Hadoop [13] and output data is presented as a folder in HDFS [**?**]. Pig allows data analysts to query big data without the complexity of writing MapReduce programs. Also, Pig does not require a fixed schema to operate, allowing seamless interoperability with other applications in the enterprise ecosystem. These desirable properties of Pig Latin as well as its wide adoption[1] prompted us to select it as the data flow language for Crypsis.

We give a short overview of Pig Latin here and refer the reader to [15] for more details.

Types. Pig Latin includes *simple types* and *complex types*. The former include signed 32-bit and 64-bit integers (**`int`** and **`long`** respectively), 32-bit and 64-bit floating point values (**`float`**, **`double`**), arrays of characters and bytes (**`chararray`**, **`bytearray`**), and **`boolean`**s. Pig Latin also pre-defines certain values for these types (e.g., `null`).

[1]According to IBM [9], "*Yahoo estimates that between 40% and 60% of its Hadoop workloads are generated from Pig [...] scripts. With 100,000 CPUs at Yahoo and roughly 50% running Hadoop, that's a lot[...]*".

Complex types include the following:

- **bag**s `{...}` are collections of **tuple**s.
- **tuple**s `(...)` are ordered sets of fields.
- **map**s `[...]` are sets of key-value pairs *key*#*value*.

Furthermore, a *field* is a data item, which can be a **bag**, **tuple**, or **map**. Pig Latin statements work with *relations*; a relation is simply a (outermost) **bag** of **tuple**s. Relations are referred to by named variables called *aliases*. Pig Latin supports assignments to variables.

Operators and expressions. Relations are also created by applying operators to other relations. The main relational operators include:

JOIN This same operator is used with various parameters to distinguish between inner and outer joins. The syntax closely adheres to the SQL standard.

GROUP Elements of several relations can be grouped according to various criteria. Note that **GROUP** creates a nested set of output **tuple**s while **JOIN** creates a flat set of output **tuple**s.

FOREACH...GENERATE Generates data transformations based on columns of data.

FILTER This operator is used with **tuple**s or rows of data, rather than with columns of data as **FOREACH**...**GENERATE**.

Operators also include arithmetic operators (e.g.,`+`, `-`, `\`, `*`), comparisons, casts, as well as **STORE** and **LOAD** operators.

Pig Latin is an expression-oriented language. Expressions are written in conventional mathematical infix notation, and can include operators and functions described next.

Functions. Pig Latin includes *built-in* and *user-defined* functions. The former include several different categories:

- *Eval* functions operate on **tuple**s. Examples include **AVG**, **COUNT**, **CONCAT**, **SUM**, **TOKENIZE**.
- *Math* functions are self-explanatory. Examples include **ABS** or **COS**.
- *String* functions operate on character strings. Examples include **SUBSTRING** or **TRIM**.

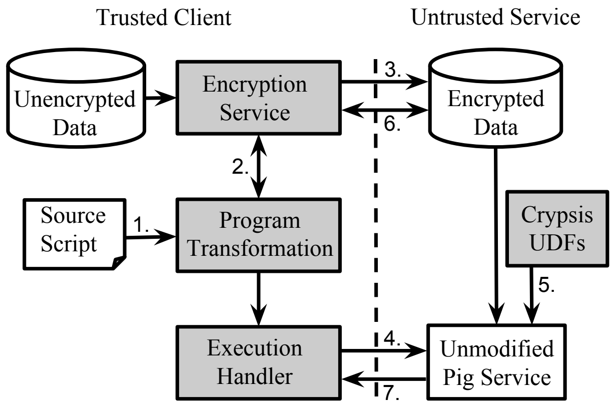

Figure 1: Architecture of Crypsis. Shading indicates novel components due to Crypsis.

User-defined functions (UDFs) can be defined in Java, Python, JavaScript and Ruby, with the support for Java being most matured. Java UDFs are expressed by subclassing `EvalFunc<T>`, with `T` corresponding to the respective input and return type, and implementing its only method with signature

```
T exec(T input) throws IOException
```

UDFs in Pig Latin can be classified into two families: (1) built-in UDFs, and (2) custom UDFs. The former are pre-defined functions that are delivered with the language toolchain. Examples include "composed" functions like **SUM**. In this paper we focus on the former kind of UDFs due to the complexity involved in analyzing the latter.

3 Architecture and System Overview

We designed Crypsis having in mind an adversary capable of fully manipulating the cloud infrastructure. The adversary can see encrypted data and Pig Latin scripts that operate on the data. The adversary can control the computation software and control the cloud infrastructure. Crypsis ensures confidentiality in the presence of such an adversary. However, Crypsis does not address integrity and availability issues. (Corresponding solutions are described in our previous work [22] which inversely, however, does not address confidentiality.)

Figure 1 illustrates the architecture of Crypsis prototype. Script execution proceeds by the following steps:

1. Program transformation. The user submits a Pig Latin script that operates on unencrypted data (source script). Crypsis analyzes it to identify the *required encryption schemes* under which the input data should be encrypted. Operators in source script are replaced with calls to Crypsis UDFs that perform the corresponding operations on encrypted data and constants are replaced with their encrypted values to generate a target script that executes entirely on encrypted data. Details of this transformation are presented in Section 4.

2. Infer encryption schemes missing from cloud. The encryption service keeps an *input data encryption schema* which tracks what parts of the input data are already encrypted and stored in the cloud. This is necessary since some parts of the input data might be encrypted under multiple encryption schemes (to support multiple operations) and other parts might not be available in the cloud at all. Based on the *input data encryption schema* and the *required encryption schemes* inferred in the previous step, the encryption service identifies the encryption schemes missing from the cloud.

3. Encrypt and send data to the cloud. In case some *required encryption schemes* are not available in the cloud, the encryption service draws the unencrypted data from the local storage, encrypts it using the appropriate cryptosystem and sends it to the cloud storage. Crypsis makes use of several encryption schemes each implemented using different cryptosystems. The first scheme is the *randomized* (RAN) encryption which does not support any operators, and is intuitively, the most secure encryption scheme. We implement RAN using Blowfish [21] to encrypt integer values, taking advantage of its smaller 64-bit block size, and use AES [7] which has a 128-bit block size to encrypt everything else. We use CBC mode in both of these cryptosystems with a random initialization vector. The next scheme is the *deterministic* (DET) encryption which allows equality comparisons over encrypted data. We construct DET using Blowfish and AES pseudo-random permutation block ciphers for values of 64 bits and 128 bits respectively, and pad smaller values appropriately to match the expected block size. For values longer than 128 bits we follow the approach used in CryptDB [17] and use a variant of CMC mode [14] with a zero initialization vector. We implement the *order-preserving encryption* (OPE) scheme which allows order comparisons using the order-preserving symmetric encryption [4] implementation from CryptDB. Lastly, we use the Paillier [16] cryptosystem to implement *additive homomorphic encryption* (AHE) which allows additions over encrypted data and ElGamal [11] cryptosystem to implement *multiplicative homomorphic encryption* (MHE) which allows us to perform multiplications over encrypted data.

4. Execute encrypted script. When all required encrypted data is loaded in the cloud, the execution handler issues a request to start executing the job.

5. Crypsis UDFs. Crypsis does not impose any changes to the Pig service. Instead, operations on encrypted data are handled by a set of pre-defined UDFs stored in the cloud storage along with the encrypted data.

6. Re-encryption. During the target script execution, it is possible that intermediate data are generated after some operations are performed. The encryption scheme of that data depends on the last operation performed on that data. For example, after an addition operation, the resulting *sum* will be encrypted under AHE. If that intermediate data is subsequently involved in an operation that requires an encryption scheme other than the one it is encrypted under (for example multiplying *sum* with another value requires MHE), the operation cannot be performed. Crypsis handles this situation by re-encrypting the intermediate data. Specifically, the intermediate data is sent to the client where it can be securely decrypted and then encrypted under the required encryption scheme (for example *sum* is re-encrypted under MHE), before sent back to the cloud. Once the re-encryption completed, execution of target script can proceed.

7. Results. Once the job is complete, the encrypted results are sent to the client where they can be decrypted.

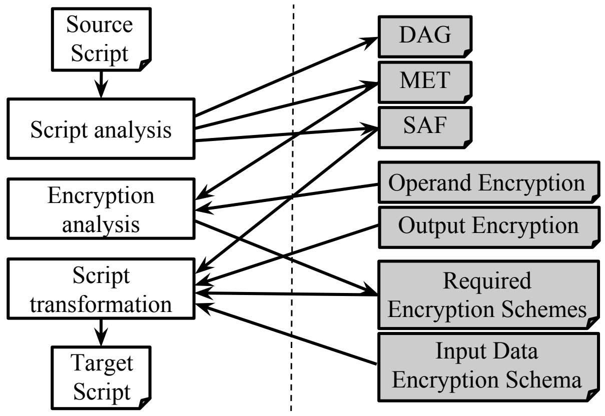

Figure 2: Program transformation in Crypsis

4 Program Analysis and Transformation

We use the Pig Latin script shown in Listing 1 as a running example to explain the analysis and subsequent transformation process for Pig Latin scripts in Crypsis. This script loads two input files: `input1` with two fields and `input2` with a single field. The script then filters out all rows from `input1` which are less than or equal to 10 (Line 3). Lines 4 and 5 group `input1` by the first field and find the sum of the second field for each group. Line 6 joins the sum per group with the second input file `input2` to produce the final result which is stored into an output file (Line 7). This script is representative of the most commonly used relational operations in Pig Latin and allow us to explain key features about Crypsis.

```
1 A = LOAD 'input1' AS (a0, a1);
2 B = LOAD 'input2' AS (x0);
3 C = FILTER A BY a0 > 10;
4 D = GROUP C BY a1 ;
```

```
E = FOREACH D GENERATE group AS b0,
  SUM(C.a0) AS b1;
F = JOIN E BY b0, B BY x0;
STORE F into 'out';
```

Listing 1: Source Pig Latin script S_1

4.1 Definitions

In order to describe our program analysis and transformation we first make the following defnitions.

Map of expression trees (MET). All the expressions that are part of the source script are represented as trees and added to the MET as values. The keys of MET are simple literals used to access these expression trees.

Data flow graph. We represent a Pig Latin script S using a data flow graph (DFG). More precisely, $S = \{V, E\}$, where V is a set of vertices and E is a set of ordered pairs of vertices. Each $v \in V$ represents a relation in S. Each $e \in E$ represents a data flow dependency between two vertices in V. Note that, it follows from the nature of Pig Latin scripts that DFG is acyclic. For describing our analysis, we consider a vertex $v \in V$ representing the Pig Latin relation R to have two parts: (a) the Pig Latin statement representing R with all expressions replaced by their keys in MET and (b) the fields in the schema of R.

Programatically a data flow graph G exposes the following interface

- `Iterator getIter()` Returns an iterator representing the list of relations in G in topologically sorted order.
- `Relation getRelation(Vertex v)` Returns the relation represented by vertex v in G.
- `Field[] getField(Vertex v)` Returns all fields in the schema of the relation represented by vertex v
- `String[] getExp(Vertex v)` Returns all keys, if any, that are part of MET and belong to the relation represented by vertex v

Set of Annotated Fields (SAF). SAF contains one entry for each field in the schema of each relation. In other words every relation, field pair maps to one entry in SAF. We refer to an individual entry simply as AF (annotated field). The intution behind using AFs instead of actual field names is that, in the transformed program, one field in the source script may be loaded as multiple fields, each under a different encryption scheme. Each AF consists the following three annotations.

FI[†]	AnField	After analysis
`A,a0`	`f0<null,NE,NE>`	`f0<null,{ALL},OPE>`
`A,a1`	`f1<null,NE,NE>`	`f1<null,{ALL},NE>`
`B,x0`	`f2<null,NE,NE>`	`f2<null,{ALL},DET>`
`C,a0`	`f3<f0,NE,NE>`	`f3<f0,{ALL},NE>`
`C,a1`	`f4<f1,NE,NE>`	`f4<a1,{ALL},DET>`
`D,gr`	`f5<[2],NE,NE>`	`f5<[2],{DET},NE>`
`D,a0`	`f6<f3,NE,NE>`	`f6<f3,{ALL},AHE>`
`D,a1`	`f7<f4,NE,NE>`	`f7<f4,{ALL},NE>`
`E,b0`	`f8<[3],NE,NE>`	`f8<[3],{DET},DET>`
`E,b1`	`f9<[4],NE,NE>`	`f9<[4],{AHE},NE>`
`F,b0`	`f10<f4,NE,NE>`	`f10<f4,{ALL},NE>`
`F,b1`	`f11<f3,NE,NE>`	`f11<f3,{AHE},NE>`
`F,x0`	`f12<f3,NE,NE>`	`f12<f3,{ALL},NE>`

Table 1: Annotated Fields for Pig Latin script in Listing 1

- parent: AFs track its lineage using this parent field. The parent of an annotated field could be another annotated field or an expression tree in MET. In cases where an AF represents a field which is newly generated as part of a Pig Latin relational operation (GROUP..BY, FOREACH etc.), the parent will be the expression tree used to generate it. Otherwise, the parent of an AF will be the corresponding AF in the vertex in the DAG that comes before it.
- avail: This annotation represents the set of encryption schemes available for the field being represented. At the begining of the analysis this property will be empty. This will be filled in as part of the program analysis.
- req: This represents the encryption scheme that is required for this field by our execution engine.

Programatically SAF exposes the following interface

- `fieldsForDataSet(Relation r)` Returns all the annotated fields that represent fields in relation `r`.

4.2 Analysis

Using the programming abstractions defined above, we describe the program analysis and transformation that we perform. Before the analysis, as part of initialization, all operators and UDFs to be used in the script are pre-registered with the encryption scheme required for operands and the encryption scheme of output generated. Some Pig Latin relational operators also require fields to be in specific encryption schemes. For example, fields or expressions that become the grouped field of **GROUP BY**, require the field or expression to be in DET. Further the encryption scheme of the new 'group' field generated by the **GROUP BY** operator will also be DET. To keep the description simple, we encapsulate this static information as two function calls: OUTENC(*oper*) which returns the

```
A = LOAD 'enc_input1' AS (a0_ope,
  a0_ah, a1_det);
B = LOAD 'enc_input2' AS (x0_det);
C = FILTER A BY OPE_GREATER(a0_ope ,
  '0xD0004D3D841327F2CCE7133ABE1EFC14');
D = GROUP C BY a1_det ;
E = FOREACH D GENERATE group AS b0,
  SUM(B.a0_ah) AS b1;
F = JOIN E BY b0, B BY x0_det;
STORE F into 'out';
```

Listing 2: Transformed Pig Latin script

output encryption scheme of the operator *oper* and INENC(*oper*) which returns the operand encryption scheme required for *oper*.

The available encryption scheme for each AF is identified by observing the lineage information available through the parent field of AFs and metadata about the encrypted input file. Details of available encryption scheme analysis is presented as Algorithm 1. The available encryption schemes for AFs part of the `LOAD` operation is set based on metadata about the encrypted file. For non `LOAD` operators, the available encryption schemes for AFs depend on their lineage. If the AF is derived by an expression, available schemes for the AF is determined by the deriving expression. For example, the available encryption scheme for the AF representing field `b1` in "`B = FOREACH A GENERATE a0+a1 as b1;` is determined by the expression `a0+a1`" or more precisely the operator `+`. If the AF is not explicitly derived, but is carried forward from a parent relation, then the AF simply inherits the available encryption schemes of the parent AF. For example, the available encryption schemes for the AF representing field `a0` in "`B = FILTER A BY a0 < 10;`" is same as the encryption schemes available for the parent AF representing field `a0` in relation `A`.

Now we describe how the required encryption field is populated in each AF. As part of our initialization, once SAF is generated, we replace each field in MET by the AF representing that field. Once this is done, the required encryption for each AF can be identified by iterating over all leaf nodes of all expression trees in the MET. The required encryption scheme for each leaf node is same as INENC(parent), where *parent* is the operator for which the leaf node is the operand. This procedure is thus straightforward and we do not present this as a separate algorithm.

4.3 Transformation

Now we describe how a Pig Latin script, represented as $S = \{V, E\}$, is transformed into the target script. First we give an overview of the different types of transfomations

Algorithm 1 Determining available encryption scheme for fields in SAF fs for a DFG G

```
procedure AVAILABLEENC($G, fs$)                ▷ $fs$ is FieldSet
    $iter \leftarrow G$.getIter();
    while $r \leftarrow iter$.next() do
        $anFields \leftarrow fs$.getFieldsOf($r$)
        for each $af \in anFields$ do
            FINDAVAIL(af)
        end for
    end while
end procedure
$met$                                            ▷ MET for DFG $G$
function FINDAVAIL($af$)                ▷ $af$ is Annotated Field
    if $af$.parent is $null$ then
        $af$.avail ← ENCSERVICE($af$.getField())
    else if $af$.parent $\in met$.keys() then
        $exprTree \leftarrow met$.get($af$.parent)
        $af$.avail ← OUTENC($exprTree$.root())
    else
        $af$.avail ← $af$.parent.avail
    end if
end function
```

Algorithm 2 Program Transformation

```
procedure TRANSFORM(SAF saf)
    $dfg$                                        ▷ DFG
    $met$                                        ▷ MET
    Loader                     ▷ The Encrypted File Loader
    for each $af \in saf$ do
        if $af.req \neq NE$ then
            $field \leftarrow getField(af.fieldName)$
            if $af$.req $\not\subseteq af$.avail then
                $met$.insert(reEnc, af)
            else
                $encCol$ ← ENCSERVICE(field, req)
            end if
            $met$.replace($af, encCol$)
        end if
    end for
end procedure
```

that we perform.

Multiple encryption fields. A potential overhead for the client is having to re-encrypt data in an encryption scheme appropriate for evaluation in the cloud. We use multiple encryptions for the same column whenever possible to minimize such computations on the client side. For example in Listing 1 column $a0$ is used for comparison (line 3) and for addition (line 5). The naive approach in this case would be load column $a0$ in OPE to do the comparison and re-encrypt it to AHE between lines 3 and 5 to enable addition. But this puts a computational burden on the client cluster. Such a re-encryption can be avoided if we transform the source Pig Latin script such that both encryption forms for $a0$ is loaded up front as two columns. We identfify oppertunities for such optimizations using the SAF abstraction. We describe next, how the different encryption schemes to be loaded for a field a_i can be identified. Once the req field is populated in SAF as part of analysis, we query the SAF for all AFs where a_i is part of the key. These AFs represent all useages of field a_i in the Pig Latin script. The unique set of values of req of these AFs will represent the different encryption schemes required for a field a_i. The transformation then generates the list of fields to be loaded from the encrypted version of the input file using metadata returned by ENCSERVICE, and modifies the MET to use the newly loaded fields. Listing 2 shows the transformed Pig Latin script for our example. Note that field a0 is loaded twice, under two encryption schemes.

Re-encryption and Constants AFs where the req encryption is not a subset of avail represent cases where a valid encryption scheme is not present to perform an operaton. In such cases, we wrap the AFs in a *reencrypt* operation. The reencrypt operation contacts the encryption service on the trusted client to convert the field to the required encryption scheme. Constants in MET are also transformed into encrypted form as required for the operation in which they are used. Note that Listing 2 shows the constant 10 encrypted using the OPE scheme to perform the comparison.

5 Evaluation

In this section we demonstrate the practicality of Crypsis.

Microbenchmarks. We first constructed a microbenchmark that compares the size of the unencrypted data with the size of the encrypted data. We also compare the time taken to perform simple operations on unencrypted data with the time taken to perform the same operations on encrypted data and show the results in Table 2. The evaluation of this microbenchmark was performed on a single machine with two 32 bit CPUs and 3 GB of RAM.

PigMix. We ran the Apache PigMix [3] benchmark to evaluate the perfomance of Crypsis . The evaluation was carried out using a cluster of 11 *c3.large* nodes from Amazon EC2 [1]. The nodes have two 64 bit virtual CPUs and 3.75 GB of RAM. We discuss the results here and some lessons learned.

We used the data generator script that is part of PigMix to generate an input data set with 3300000 rows (5GB). We wrote Pig Latin queries to encrypt the data at the client side. One problem we faced was in Pig Latin scripts that projects the value of **`map`** fields using **`chararray`** constants as keys (**`map`**`#'key'`). Current Pig implementation does not allow variables, functions or data types other than **`chararray`**s to act as keys for such projections. This causes an issue because we consider encrypted text always as **`bytearray`**s. Further, even if we cast a **`bytearray`** to **`chararray`** in order use it as a key in the map, we will have to deal with representing special characters in the **`bytearray`** in script file. To avoid this, during encryption of keys for maps, we generate the hex representation of the byte array and store it as a string of characters. This allows us to transform a projection like `page_links#'b'` into `page_links#'4E87D339A9550DCDB6137AAD'`. Also, Pig Latin supports **`int`**, **`long`**, **`float`** and **`double`** literals for representing numbers within the Pig Latin script. Unfortunately, the range of numbers that may appear in the cipher text space may exceed the max limits in each of these supported data types. We overcome this limitation by representing the numeric constants as strings and performing the actual comparison using `BigInteger` or `BigDecimal` classes within UDFs.

Figure 3 shows the results of the PigMix benchmark. On average we observe 3× overhead in terms of latency, which is extremely low compared to FHE. We also observed that this overhead is correlated more towards size of the input data and not the actual computation.

Word Count We ran Crypsis on 100 GB of data obtained from a Wikipedia dump, and our cluster consisted of 25 Amazon EC-2 nodes. Since the data was in XML format, we first ran a Pig Latin script to obtain the text section from the file. Next, we tokenized and encrypted all the words, then executed our word count script on the encrypted words. Lastly, we decrypted and stored the top 5 words. In addition to an encyrpted word count, we also ran our word count script on the plaintext. Both the encrypted and plaintext versions were run 4 times and the results can be found in Figure 4. The time for Crypsis does not include the time needed to encrypt the data,

	Size (KB)			Time (ms)			
Λ[†]	NE[‡]	AHE	MHE	NE +	NE $\times$	AHE +	MHE $\times$
1	134	6,035	6,077	16.88	17.19	267.41	1,337.59
2	269	12,071	12,153	32.05	31.92	476.78	2,266.50
3	404	18,106	18,229	48.40	47.78	686.74	3,202.11
4	538	24,142	24,306	62.93	61.82	895.22	4,117.67
5	673	30,177	30,382	77.24	76.00	1,108.11	5,049.68
6	807	36,212	36,459	91.70	89.98	1,314.20	5,977.62
7	942	42,248	42,535	106.00	104.08	1,521.14	6,894.75
8	1,076	48,283	48,611	120.57	118.38	1,730.35	7,818.04
9	1,211	54,319	54,688	134.96	132.71	1,938.66	8,733.67
10	1,345	60,354	60,764	149.62	146.86	2,147.37	9,658.56

Table 2: Comparing size of data and latency of addition and multiplication operations over plaintext and encrypted data. [†]Λ is the number of operations performed in multiples of 1000. [‡]NE denotes no encryption or plaintext data.

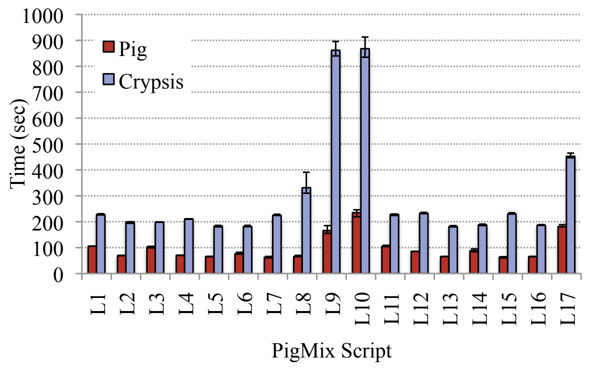

Figure 3: Latency of Pig Latin scripts in PigMix

since it's assumed the user will have the encrypted data on the untrusted cloud.

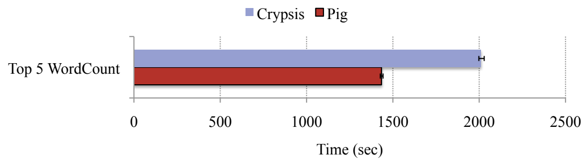

Figure 4: Word Count run on Crypsis and plaintext

6 Related Work

Differential privacy and functional encryption solve different problems associated with confidentiality. Differential privacy aims to improve the accuracy of statistical queries, without revealing information about individual records; data and computation reside within the trusted server. Functional encryption focuses on ensuring that an untrusted process learns only the ouput of a function $f(x)$ about data. In comparison, Crypsis assumes data and computation to reside in an untrusted environment and prevents the untrusted process from learning anything about data. sTile [5] keeps data confidential by distributing computations onto large numbers of nodes in the cloud, requiring the adversary to control more than half of the nodes in order to reconstruct input data. Airavat [19] combines mandatory access control and differential privacy to enable MapReduce computations over sensitive data. Both sTile and Airavat require the cloud provider and cloud infrastructure to be trustworthy.

CryptDB [17] is a seminal system leveraging PHE for cloud-based data management. As suggested by its name, CryptDB is a database system, focusing on SQL queries. It does not consider the more expressive kind of dataflow programs as in Pig Latin, or MapReduce-style parallelization of queries. Monomi [24] improves performance of encrypted queries similar to those used in CryptDB and introduces a designer to automatically choose an efficient design suitable for each workload. MrCrypt [23] consists in a program analysis for MapReduce jobs that tracks operations and their requirements in terms of PHE. When sequences of operations are applied to a same field, the analysis defaults to FHE, noting that the system does not currently execute such jobs at all due to lack of available FHE cryptosystems. Thus several PigMix benchmarks are disregarded for evaluation.

7 Conclusions and Outlook

In this paper, we have outlined the necessity for computing on encrypted big data and the potential of partially homomorphic encryption towards this goal. We also presented the high-level design of Crypsis, a system that achieves this goal.

We are currently investigating a number of optimizations, and further, heuristics for selecting from different possible execution paths in the transformed Pig Latin script. In particular we're investigating paths which perform re-encryption of data at clients (in contrast to costly re-encryption in the cloud as with FHE) and minimizing these re-rencryptions themselves. To this end we're also employing sampling to determine amounts of data at different points in analysis jobs.

References

[1] Amazon EC2. `http://amazon.com/ec2`.

[2] Apache Pig. `http://pig.apache.org`.

[3] Apache PigMix benchmark. `https://cwiki.apache.org/confluence/display/PIG/PigMix`.

[4] BOLDYREVA, A., CHENETTE, N., LEE, Y., AND O'NEILL, A. Order-preserving Symmetric Encryption. In *EUROCRYPT* (2009).

[5] BRUN, Y., AND MEDVIDOVIC, N. Keeping Data Private while Computing in the Cloud. In *IEEE CLOUD* (2012).

[6] CHAMBERS, C., RANIWALA, A., PERRY, F., ADAMS, S., HENRY, R. R., BRADSHAW, R., AND WEIZENBAUM, N. Flumejava: Easy, Efficient Data-parallel Pipelines. In *PLDI* (2010).

[7] DAEMEN, J., AND RIJMEN, V. *The Design of Rijndael: AES - The Advanced Encryption Standard.* Springer Verlag, Berlin, Heidelberg, New York, 2002.

[8] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI* (2004).

[9] DEVELOPERWORKS, I. Process your Data with Apache Pig, 2012. `http://www.ibm.com/developerworks/library/l-apachepigdataquery/`.

[10] DINUR, I., AND NISSIM, K. Revealing Information While Preserving Privacy. In *PODS* (2003).

[11] ELGAMAL, T. A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *IEEE Transactions on Information Theory 31*, 4 (1985).

[12] GENTRY, C., SAHAI, A., AND WATERS, B. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *CRYPTO* (2013).

[13] HADOOP. Hadoop. `http://hadoop.apache.org/`.

[14] HALEVI, S., AND ROGAWAY, P. A tweakable enciphering mode. In *CRYPTO* (2003).

[15] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD* (2008).

[16] PAILLIER, P. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *EUROCRYPT* (1999).

[17] POPA, R. A., REDFIELD, C. M. S., ZELDOVICH, N., AND BALAKRISHNAN, H. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *SOSP* (2011).

[18] RIVEST, R., SHAMIR, A., AND ADLEMAN, L. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM 21*, 2 (1978), 120–126.

[19] ROY, I., SETTY, S. T. V., KILZER, A., SHMATIKOV, V., AND WITCHEL, E. Airavat: Security and Privacy for MapReduce. In *NSDI* (2010).

[20] SAHAI, A., AND WATERS, B. Fuzzy Identity Based Encryption. *IACR Cryptology ePrint Archive 2004* (2004).

[21] SCHNEIER, B. Description of a new variable-length key, 64-bit block cipher (blowfish). In *Fast Software Encryption* (1994), Springer-Verlag, pp. 191–204.

[22] STEPHEN, J. J., AND EUGSTER, P. Assured Cloud-Based Data Analysis with ClusterBFT. In *Middleware* (2013).

[23] TETALI, S., LESANI, M., MAJUMDAR, R., AND MILLSTEIN, T. MrCrypt: Static Analysis for Secure Cloud Computations. In *OOPSLA* (2013).

[24] TU, S., KAASHOEK, F., MADDEN, S., AND ZELDOVICH, N. Processing Analytical Queries over Encrypted Data. In *PVLDB* (2013).