Program Analysis for Secure Big Data Processing[*]

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

ABSTRACT

The ubiquitous nature of computers is driving a massive increase in the amount of data generated by humans and machines. Two natural consequences of this are the increased efforts to a. derive meaningful information from accumulated data and b. ensure that data is not used for unintended purposes. In the direction of analyzing massive amounts of data (a.), tools like MapReduce, Spark, Dryad and higher-level scripting languages like Pig Latin and DryadLINQ have significantly improved corresponding tasks for software developers. The second, but equally important aspect of ensuring confidentiality (b.), has seen little support emerge for programmers: while advances in cryptographic techniques allow us to process directly on encrypted data, programmer-friendly and efficient ways of programming such data analysis jobs are still missing. This paper presents novel data flow analyses and program transformations for Pig Latin, that automatically enable the execution of corresponding scripts on encrypted data. We avoid fully homomorphic encryption because of its prohibitively high cost; instead, in some cases, we rely on a minimal set of operations performed by the client. We present the algorithms used for this translation, and empirically demonstrate the practical performance of our approach as well as improvements for programmers in terms of the effort required to preserve data confidentiality.

Categories and Subject Descriptors

D.2.2 [**Software**]: Software Engineering—*Design Tools and Techniques*; C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*

Keywords

Cloud computing, big data, privacy

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

ASE'14, September 15-19, 2014, Vasteras, Sweden.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3013-8/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642937.2643006.

1. INTRODUCTION

The cloud computing model has evolved as a cost-efficient data analysis platform for corporations, governments and other institutions. A rich set of tools like MapReduce [14], Dryad [21], or Spark [39] — just to name a few — allow developers to execute data analysis jobs in the cloud easily. However, in order to take advantage of these possibilities, developers are faced with the decision of moving sensitive data to the cloud, and placing trust on infrastructure providers. Even with a trusted provider, malicious users and programs and defective software can leak data.

1.1 Computing on encrypted data

Several existing cryptographic systems ("cryptosystems") allow meaningful operations to be performed directly on encrypted data. Advances occurring regularly in *fully homomorphic encryption* (FHE) schemes (e.g. [19]) further increase the scope and performance of computations on encrypted data. These cryptosystems present an opportunity to maintain sensitive data only in an encrypted form in the cloud while still enabling meaningful data analysis. Even within an enterprise, maintaining data in an encrypted format helps prevent insider attacks and accidental leaks.

Unfortunately, such advancements in cryptography have not translated into programmer-friendly frameworks for big data analysis with acceptable performance. Currently, for a big data programmer to *efficiently* put existing homomorphic schemes to work she will have to explicitly make use of corresponding cryptographic primitives in her data analysis jobs. Consider a simple program that finds the total marks obtained by students: the programmer is burdened by the task of identifying an encryption scheme suitable to represent such an operation (*Paillier* [26] in this example with addition) and express the operation (summing) in cryptographic terms (multiplication of encrypted operands followed by a modulo division by the square of the public key).

To make this task even more difficult for the programmer, big data analysis jobs are typically expressed in *domain-specific* security-agnostic high-level data flow languages, like Pig Latin [25] or FlumeJava [11], which are commonly compiled to *sequences* of several MapReduce tasks [14]. This makes the explicit use of cryptographic primitives even harder. Recent approaches to automatically leveraging partially homomorphic encryption (PHE) — e.g., addition of values encrypted with Paillier cryptosystem via multiplication — for secure cloud-based computing target different application scenarios or work-loads. For instance, the MySQL-based CryptDB [28] supports SQL queries on encrypted data yet does not enable parallelization through MapReduce; MrCrypt [36] only supports individual MapReduce tasks.

1.2 Approach

In this paper we present a program analysis and transformation scheme for the Pig Latin high-level big data analysis language which enable the efficient execution of corresponding scripts by exploiting cloud resources but without exposing data in the clear. We implemented this program analysis and transformation inside a secure Pig Latin runtime (SPR). More specifically, we extend the scope of encryption-enabled big data analysis based on the following insights:

Extended program perspective: By *analyzing entire data flow programs*, SPR can identify many opportunities for operating in encrypted mode. For example, SPR can identify operations in Pig Latin scripts that are inter-dependent with respect to encryption, or inversely, independent of each other. More precisely, when applying two (or more) operations to the 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 encryptions.

Extended system perspective: By considering the possibility of performing subcomputations on the client side, SPR 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 [36]) 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 that 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.*

1.3 Contributions

The contributions of this paper are as follows. After presenting background information on homomorphic encryption and Pig Latin, we

1. propose an execution model for executing Pig Latin scripts in the cloud without sacrificing confidentiality of data.
2. outline a novel data flow analysis and transformation technique for Pig Latin 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 SPR that maximize the amount of computations done on encrypted data in the cloud.
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.

The remainder of this paper is organized as follows. Section 2 presents background information on homomorphic encryption and Pig Latin. Section 3 outlines the design of our solution. Section 4 presents our program analysis and transformation. Section 5 outlines the implementation of SPR. Section 6 presents empirical evaluation. Section 7 discusses limitations. Section 8 contrasts with related work. Section 9 concludes with final remarks. A preliminary version of this report outlining our approach and infrastructure appeared at the HotCloud 2014 workshop [35].

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 $\exists\psi$ s.t.

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

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

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

Other 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.

2.2 Pig/Pig Latin

Apache Pig [2] is a data analysis platform which includes the Pig runtime system for the high-level data flow language Pig Latin [25]. Pig Latin expresses data analysis jobs as sequences of data transformations, and is compiled to MapReduce [14] tasks by Pig. The MapReduce tasks are then executed by Hadoop [20] and output data is presented as a folder in HDFS [33]. 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 SPR.

We give a short overview of Pig Latin here and refer the reader to [25] for more details.

2.2.1 *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

[1]According to IBM [15], "*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[...]*".

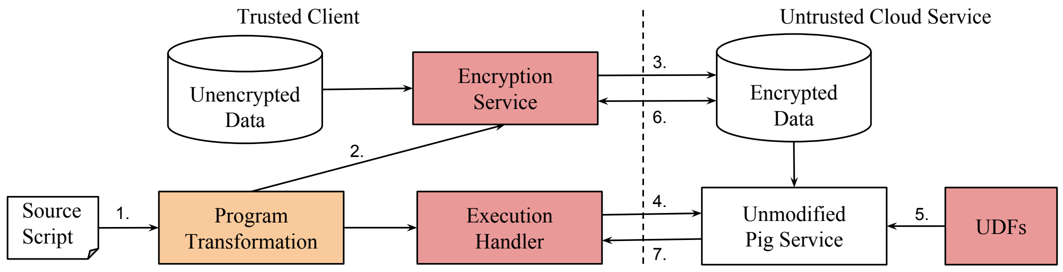


Figure 1: Execution Model of SPR. Shading indicates components introduced by SPR.

(**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`).

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.

2.2.2 *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, and **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.

2.2.3 *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**.

User-defined functions (UDFs) in Pig Latin are classified as: (1) built-in UDFs, or (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.

2.2.4 *Example*

To understand the constructs of Pig Latin, we present a simple word count example in Listing 1. In this example, a relation `input_lines` is first generated with elements called `line` of type **chararray** read from an input `input_file`. Next, the script breaks these lines into tokens, representing individual words. Then, occurrences of a same word are **GROUP**ed, and their respective number of occurrences **COUNT**ed. Finally the data in this table is **STORE**d in a file `output_file`.

```
input_lines = LOAD 'input_file' AS
  (line:chararray);
words = FOREACH input_lines GENERATE
  FLATTEN(TOKENIZE(line)) AS word;
word_groups = GROUP words BY word;
word_count = FOREACH word_groups GENERATE
  group, COUNT(words);
STORE word_count INTO 'output_file';
```

Listing 1: Source Pig Latin script S_1

3. EXECUTION MODEL

SPR is a runtime for Pig Latin that supports cloud-based execution of scripts written in the original Pig Latin language in a confidentiality-preserving manner. The adversary in our model can have full control of the cloud infrastructure. This means the adversary can see all data stored in the cloud and the scripts that operate on data. In order to preserve confidentiality in the presence of such an adversary only encrypted data is maintained in the cloud and SPR operates on this encrypted data. However, SPR does not address integrity and availability issues. (Corresponding solutions are described in our previous work [34] which inversely, however, does not address confidentiality.)

Figure 1 presents an overview of the execution model of SPR. Script execution proceeds by the following steps:

1. Program transformation. Script execution starts when a user submits a Pig Latin script operating on unencrypted data (source script). SPR analyzes the script to identify the *required encryption schemes* under which the input data should be encrypted. For the transformed script to operate on encrypted data, operators in the source script are replaced with calls to secure UDFs that perform the corresponding operations on encrypted data. E.g., an addition $a+b$ in the source script will be replaced by $a \times b \mod n^2$, with n the public key used for encrypting a and b. Constants are also replaced with their encrypted values to generate a target script that executes entirely on encrypted data. Details of this transformation yielding an *encryption-enabled* script are presented in Section 4.

2. Infer encryption schemes. Using the input file names used in the source script, the encryption service checks what parts of the input data are already encrypted and stored in the cloud. Some parts of the input data might already be encrypted under multiple encryption schemes (to support multiple operations) and other parts might not be available in the cloud at all. The encryption service maintains an *input data encryption schema* which keeps track of this mapping between plain text input data and encrypted data available in the cloud. 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. Once the *required encryption schemes* that are missing from the cloud is identified, the encryption service loads the unencrypted data from local storage, encrypts it appropriately and sends it to the cloud storage. The encryption service makes use of several encryption schemes each implemented using different cryptosystems. Implementation details of these encryption schemes are presented in 5. Each of these encryption schemes has its own characteristics. The first scheme is randomized (RAN) encryption which does not support any operators, and is intuitively, the most secure encryption scheme. The next scheme is the deterministic (DET) encryption scheme which allows equality comparisons over encrypted data. Order-preserving encryption (OPE) scheme allows order comparisons using the order-preserving symmetric encryption [9]. Lastly, additive homomorphic encryption (AHE) allows additions over encrypted data, and multiplicative homomorphic encryption (MHE) allows us to perform multiplications over encrypted data.

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

5. UDFs. SPR defines a set of pre-defined UDFs that handle cryptographic operations. Such UDFs are used to perform operations like additions and multiplications over encrypted data. The target script calls these UDFs using standard Pig Latin syntax as part of the script execution process.

6. Re-encryption. During the target script execution, intermediate data may be generated as 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 is already 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 the sum with another value requires MHE), the operation cannot be performed. (The ability to apply additions and multiplications in sequence is viewed as defining characteristic of FHE.) SPR 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 the sum is re-encrypted under MHE), before sent back to the cloud. Once the re-encryption is complete, the execution of target script can proceed.

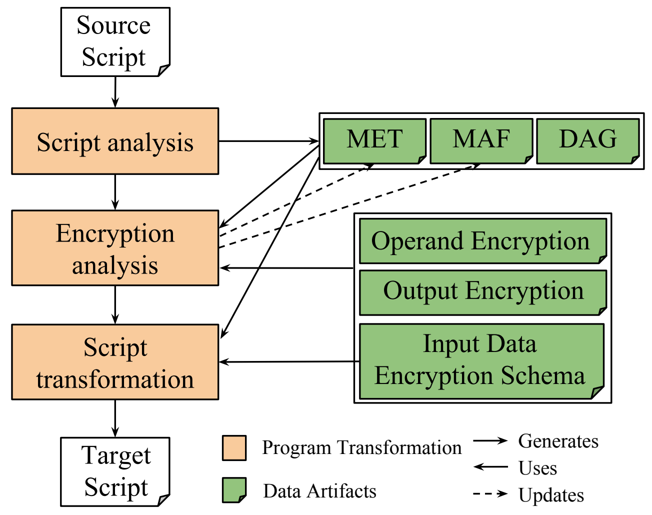


Figure 2: Program transformation components and artifacts

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

4. PROGRAM ANALYSIS AND TRANSFORMATION

In this section we give a high level overview of how Pig Latin scripts are analyzed by SPR and transformed to enable execution over encrypted input data.

4.1 Running example

We use the Pig Latin script shown in Listing 2 as a running example to explain the analysis and subsequent transformation process. This script is representative of the most commonly used relational operations in Pig Latin and allows us to explain key features about SPR. The 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 `out` (Line 7).

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

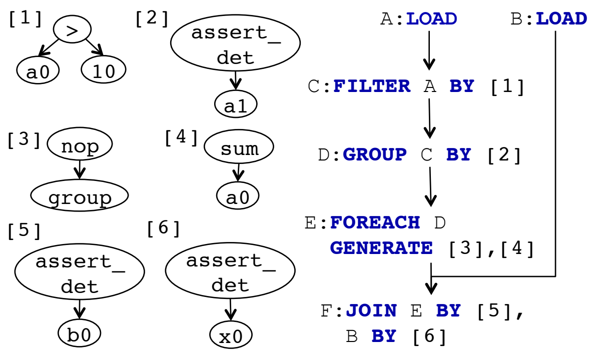
Listing 2: Source Pig Latin script S_1

4.2 Definitions

In order to describe our program analysis and transformation we first introduce the following definitions.

4.2.1 *Map of expression trees (MET)*

All the expressions that are part of the source script are represented as trees and added to the *map of expression trees* (MET) as values. The keys of the MET are simple literals used to access these expression trees. Figure 3a shows the MET for the Pig Latin script in Listing 2. Keys of the MET are shown in square brackets. Note that operands that are not part of any expression are surrounded by a *nop* (no operation) vertex and operands that are part of a group by or join operation are surrounded by an `assert_det` (assert deterministic) vertex. The `assert_det` vertex can be seen as a simple operator that requires its operand to be present in DET encryption.



(a) Map of expression trees (b) Data flow graph

Figure 3: MET and DFG for Pig Latin script in Listing 2. Keys of MET are shown in square brackets.

4.2.2 *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 the Pig Latin language 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 the MET and (b) the fields in the schema of R.

Programmatically 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.

4.2.3 *Annotated field (AF)*

We represent each field in the schema of each relation by an annotated field abstraction. In other words there will be one AF for every $\langle relation, field\rangle$ pair. 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 of the following three annotations.

- parent: AFs track their lineage using this parent field. The parent of an annotated field could be another annotated field or an expression tree in the 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 for encryption-enabled execution.

4.2.4 *Map of annotated fields (MAF)*

We capture the mapping between $\langle relation, field\rangle$ pairs and corresponding AFs using *map of annotated fields* (MAF). The MAF for the Pig Latin script in Listing 2 is shown in Table 1. Note that annotated fields are unique within a Pig Latin script which allows us to implement the MAF as a bidirectional map allowing lookups based on keys as well as values.

Table 1: Map of annotated fields for Pig Latin script in Listing 2. †Keys are pairs of $\langle relation, field\rangle$ and ‡values are corresponding annotated fields. `NE` represents no encryption.

Key†	Value‡	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>

4.3 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, the field or expression which is the grouped field of **`GROUP BY`**, require the field or expression to be available in DET encryption. Further the encryption scheme of the new "group" field generated by the **`GROUP BY`** operator will also be available in DET encryption. For example, the `assert_det` function that we introduced will be registered with required encryption scheme as DET and output encryption scheme also as DET. To keep the description

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

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

```
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 3: Transformed Pig Latin script

simple, we encapsulate this static information as two function calls in our algorithm: OUTENC(*oper*) which returns the output encryption scheme of the operator *oper* and INENC(*oper*) which returns the operand encryption scheme required for *oper*. These functions are summarized in Table 2.

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 by Algorithm 1. The available encryption schemes for AFs part of the `LOAD` operation are set based on metadata about the encrypted file. For other than `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 are 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 by 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`.

Table 2: Description of functions used in program analysis and transformation

Function name	Description
GETAFS(maf, r)	Returns list of annotated fields which contain the relation r as a part of their key in maf
OUTENC(op)	Returns the encryption scheme in which the result will be after performing the operation op
INENC(op)	Returns encryption schemes in which the operands of operator op much be specified
ADDTOLIST($list, elem$)	Adds $elem$ to the list of values represented by $list$
REPLACEAF(af, fld)	Replaces all occurrences of the annotated field af in expressions and relations with the field fld
INSERTVERTEX(v, bv)	Inserts new vertex v before vertex bv in met. The parent of v is set to be the parent of bv and the parent of bv is set to be v

Next we describe how the required encryption field is populated in each AF. As part of our initialization, once the MAF is generated, we replace each field in the 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 vertices of all expression trees in the MET. The required encryption scheme for each leaf vertex is the same as INENC($parent\_operator$), where $parent\_operator$ is the operator for which the leaf vertex is the operand. This procedure is thus straightforward and we do not present this as a separate algorithm.

4.4 Transformation

Next we describe how a Pig Latin script, represented as $S = \{V, E\}$, is transformed into the encryption-enabled target script. We describe the transformation process in multiple sections. In each section we give an overview of why we perform a specific transformation and then describe the transformation process itself.

4.4.1 Multiple encryption fields

A potential overhead for the client is having to re-encrypt data in an encryption scheme appropriate for execution 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 2 column $a0$ is used for comparison (line 3) and for addition (line 5). The naïve approach in this case would be to load column $a0$ in OPE to do the comparison and to re-encrypt it to AHE between lines 3 and 5 to enable addition. But this puts a computational burden on the client cluster, and requires sending data back-and-forth between the client and the cluster, thus increasing latency. Such a re-encryption can be avoided if we transform the source Pig Latin script such that both encryption forms for $a0$ are loaded upfront as two columns. We

Algorithm 2 Program Transformation

```
  $S$                                              ▷ Pig Latin script S
  $met$                                            ▷ MET
  $file$                                           ▷ Input file used in LOAD
  procedure TRANSFORMMAIN($S$)
                               ▷ Process plain text input files loaded in script S
      for each $input\_file \in S$ do
          TRANSFORM($input\_file$)
      end for
  end procedure
  function TRANSFORM($file$)
      INITILIZE_EMPTY($list$)
      for each $pt\_col\_id \in file$ do
          $af \leftarrow$ AFFROMCOL($pt\_col\_id$)
          FINDREQ($list, af, pt\_col\_id$)
                    ▷ list now contains all the encrypted columns
                    ▷ corresponding to pt_col_id that needs to be
  loaded
          ENCFILELOADER(list, file)
      end for
  end function
  function FINDREQ($list, parent\_af, pt\_col\_id$)
      for each $af \in maf \mid af$.parent $== parent\_af$ do
          if $af$.req $\nsubseteq af$.avail then
              INSERTVERTEX($re\_enc, af$)
          else
              $enc\_field \leftarrow$ ENCSERVICE($pt\_col\_id, af$.req)
              ADDTOLIST($list, enc\_field$)
              REPLACEAF($af, enc\_field$)
              FINDREQ($list, af$)
          end if
      end for
  end function
```

identify opportunities for such optimizations using the MAF abstraction. We describe next how the different encryption schemes to be loaded for a field a_i (a_i being a field in the source script and not an AF) can be identified. Once the req field is populated in MAF as part of analysis, we query the MAF for all AFs where a_i is part of the key. These AFs represent all usages of field a_i in the Pig Latin script. The set of values of req of these AFs and their child AFs represents the different encryption schemes required for 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 the encryption service, and modifies the MET to use the newly loaded fields. This is described as function FINDREQ in Algorithm 2. Listing 3 shows the transformed Pig Latin script for our example. Note that field `a0` is loaded twice, under two encryption schemes.

4.4.2 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 operation. In such cases, we wrap the AFs in a specific **`reencrypt`** operation in the target script. This 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 3 shows the constant `10` encrypted using the OPE scheme to perform the comparison.

5. IMPLEMENTATION

In this section we provide details about our prototype implementation of SPR.

5.1 Overview

SPR is implemented as 6084 lines of Java code with two separate builds: (a) a client component which is deployed in the trusted space and (b) a cloud component which is deployed in the untrusted cloud (see Figure 1). The client component can generate and read both public and private keys used for encrypting and decrypting data as well as access the plain text input. The cloud component can only read the public keys used to encrypt data and has no access to private keys or plain text input. We use ØMQ [7] as the underlying messaging passing subsystem of SPR and the GNU multiple precision arithmetic library GMP [5] to perform fast arbitrary precision arithmetic operations. Both these libraries are implemented in native code and we use JNI to invoke the corresponding functions from Java. Our current version of SPR works with Apache Pig 0.11.1 running on top of Hadoop 1.2.1.

5.2 Program transformation

Program transformation is the component responsible for generating the transformed Pig Latin data flow graph that works on encrypted data by performing the transformation process described in Section 4.4. In particular, if a transformed script contains re-encryption operations, the program transformation component assigns a unique *operation_id* to each such operation. This *operation_id* is included as part of the re-encryption function in the transformed Pig Latin script. Subsequently, the details needed to perform the re-encryption (for example the encryption schemes involved) are assigned to the *operation_id* which is then registered with the encryption service. When the script is executed, the encryption service uses the *operation_id* to retrieve the information needed to perform the re-encryption.

5.3 Encryption service

The encryption service handles the initial data encryption as well as runtime re-encryptions. Initial data encryption are performed using Pig Latin scripts on the client-side where data is available as plain text. Encryption is done using UDFs that take each data item and convert it into a ciphertext corresponding to the required encryption scheme. Re-encryption requests are received by the trusted client over TCP sockets. These requests are added into blocking queues of worker threads. Worker threads process each request and return a response.

Encryption schemes. We implement randomized encryption (RAN) using Blowfish [31] to encrypt integer values, taking advantage of its smaller 64-bit block size, and use AES [13] 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. We construct deterministic encryption (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. We perform order-preserving encryption (OPE) using the implementation from CryptDB. We use the Paillier [26] cryptosystem to implement additive homomorphic encryption (AHE), and the ElGamal [18] cryptosystem to implement multiplicative homomorphic encryption (MHE).

5.4 UDFs

The cloud component consists of custom UDFs and communication channels. Re-encryption operations are implemented as aggregation UDFs that take an entire relation as argument. This allows us to pipeline re-encryption requests as opposed to re-encrypting tuple by tuple. We also defined cryptographic equivalents of built-in Pig Latin UDFs like `SUM()`, or <.

6. EVALUATION

This section evaluates our approach in terms of runtime performance and programmer effort. We also share lessons learned.

6.1 Synopsis

We evaluate four aspects of our approach:

(a) practicality to compute over encrypted data in terms of latency,

(b) programmer effort saved by our automatic transformations with respect to manual handling of (partially) homomorphic encryption,

(c) performance compared to existing solutions for querying on encrypted data, and

(d) scalability by running queries on large data-sets TBs.

6.2 Practicality: PigMix

We ran the Apache PigMix [3] benchmark to evaluate the practicality and performance of SPR . The evaluation was carried out using a cluster of 11 *c*3.*large* nodes from Amazon EC2 [1]. The nodes had two 64 bit virtual CPUs and 3.75 GB of RAM. Input data with 3300000 rows (5GB) was generated by the PigMix data generator script and encrypted at the client side. We discuss these results here.

Figure 4 shows the results of the PigMix benchmark. On average we observe 3× overhead in terms of latency, which is extremely low compared to FHE (e.g., for simple multiplications we measured around 300000× slowdown with `HElib` [4]) which is currently still at least 1000′000×. We also observed that this overhead is correlated more towards size of the input data and not the actual computation. Some of the challenges we faced during the tranformation deals with limitations of Pig Latin in dealing with constants bigger than 32 bit `integers` or 64 bit `longs`. The range of numbers that appear in the cipher text space may exceed what can be represented using `integers` and `longs` and we overcome this by representing numeric constants as strings and converting them to arbitary presicion objects like `BigIntegers` within UDFs. More details are outlined in our workshop report in [35].

6.3 Programmer effort: PigMix

We now compare the number of tokens in the Pig Latin script written to be computed on plain text data with the Pig Latin script generated by our transformation. This gauges the additional work that a programmer would have to do in order to explicitly enable her script to compute on encrypted input data. We use the PigMix Pig Latin benchmarks for this comparison. Table 3 summarizes this result.

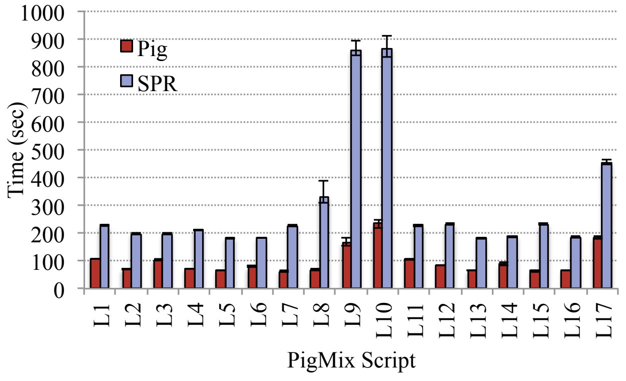


Figure 4: Latency of Pig Latin scripts in PigMix

Script	Original	Transformed	% Increase
L1	85	100	15
L2	75	86	11
L3	89	99	10
L4	62	70	8
L5	84	95	11
L6	66	74	8
L7	68	91	23
L8	61	72	11
L9	41	46	5
L10	46	57	11
L11	76	89	13
L12	134	158	24
L13	76	81	5
L14	73	79	6
L15	73	82	9
L16	62	80	18
L17	101	139	38

Table 3: Number of tokens in original and transformed PigMix scripts

As can be observed, our transformation generates Pig Latin scripts with 18% more keywords on average. For the scripts used in the PigMix benchmark, our transformation produces scripts which are between 5% and 36% more complex than the original scripts. As can be understood from our analysis, these additions are far from trivial.

6.4 Performance comparison: weather data

In order to compare our solution against the state of the art CryptDB system, we chose an application that analyses historical weather data set. The data set comprises of station names, dates and temperatures recorded on those dates on respective stations. The application issues common queries against an encrypted version of this data set. We compare the time taken by our solution to that of CryptDB. Cognizant of the fact that CryptDB was designed to address a different set of challenges than SPR we perform this comparison to understand how well our system and CryptDB performs when size of data increases. CryptDB is built on top of MySQL, a database that stores data in specialized data structures like B+ trees and optimized for data retrieval using indices and supports transactions including updates. In contrast, SPR is built on top of the Pig runtime, which specializes on reading from flat files. Furthermore, the Pig runtime depends on MapReduce, which has an intermediate stage which involves writing data to disk before reducers read it remotely over TCP channels. Conversely, by using MapReduce, Pig can leverage the processing power and memory of multiple machines while CryptDB is confined to a single node. We perform three comparisons: (1) how our solution performs on a fixed size cluster as size of input data set varies, (2) how our solution scales by increasing the number of nodes in the cluster and (3) what is the monetary cost of performing the computation (in Amazon EC2). The results presented here compare the latency of running a query that finds the average temperature for each station using encrypted data. We compare the time taken by CryptDB (single node) with the latency of running equivalent Pig Latin script on a Hadoop cluster with 4 nodes (3 worker nodes and 1 control node). We used *m3.medium* nodes on Amazaon EC2 for all our evaluations. As can be observed from Figure 7 SPRis able to exploit the parallelism offered by multiple nodes and scales well. CryptDB was designed for high throughput in terms of queries per second and not for processing high volumes of data. This can be observed by the increase in time taken for the queries over CryptDB as the number of records increases. For 15 million rows, SPR is 2.7× faster than CryptDB and shows much slower increase in job completion latency as the number of rows increases. Figure 5 shows how time taken by SPR changes as the number of nodes in the cluster increases. We show the time taken by CryptDB running on one node as a base line. We can observe that running on encrypted data shows trends comparable to regular Pig Latin jobs and latency reduces as the number of nodes increases. It may seem surprising that the time taken by the Pig Latin script running on one node was quite comparable to the time taken by CryptDB. The reason for this is that for both CryptDB and SPR the time taken is bound by the number of cryptographic operations that the CPU has to perform; which is the same for both systems. Furthermore, in order to find the average temperature, we find the AHESUM and **COUNT** of temperature readings for each group. Because both the UDFs implements `combiner` and `algebraic` interfaces, and the number of distinct groups we have is not high, most of the output generated by the map phase is combined before being pulled by the reducer. Also, even with a single node, we have two mappers and one reducer running simultaneously providing some level of parallelism. These factors limit the overhead caused by data transfers and the latency is predominantly determined by the time taken by the CPU to perform a fixed number of operations.

We also look at the monetary cost of running our computation and compare it with the cost of running CryptDB. We compute the cost by summing up the total amount charged by Amazon for all the nodes used for running our experiments. We present the calculated cost in Figure 8. As can be observed, we incur a (surprisingly) similar overall cost despite completing the job up to 2.7× faster.

6.5 Scalability: word count

In order to test the efficacy of our solution on big data, we ran experiments using 1 TB of data obtained from Wikipedia dumps [6]. 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 encrypted word count, we also ran our word count script on the plain text. The time for SPR does not include the time needed to encrypt the data, since it is assumed that the user will have the encrypted data on the untrusted cloud. We measure the latency of running word count on the encrypted input. We use Hadoop clusters of 50 and 75 nodes and summarize the results in Figure 6. We observe that for both jobs, reduction in latency as number of nodes increase follows the same trend.

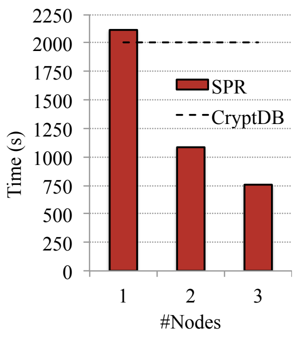


Figure 5: Comparison of SPR and CryptDB runtime latencies with varying #nodes

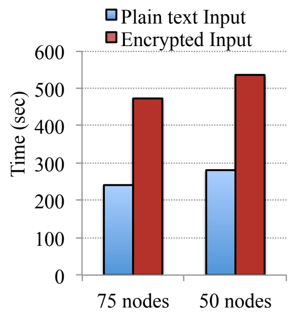


Figure 6: Latency to run word count on encrypted and plain text input

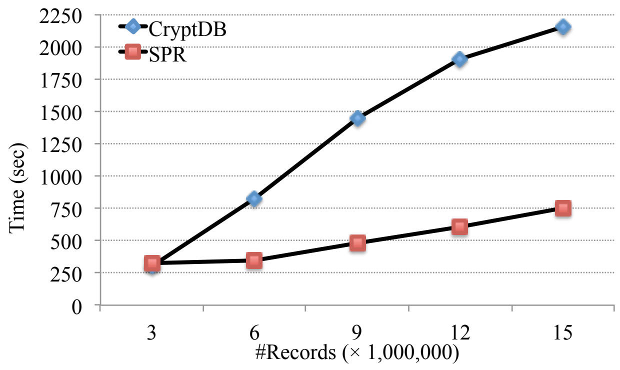


Figure 7: Comparison of SPR and CryptDB runtime latencies with varying #records

6.6 Threats to validity

In evaluating our system, we have considered several threats to validity. Multi-tenancy and the virtualized nature of Amazon EC2 can lead to variations in latency measurements. We thus ran all our evaluations several times (5 typically) and took the average to counter variations. The variations in latency we observed in different runs of the same Pig Latin scripts were relatively low (4%). For CryptDB we observed a variance of up to (10%). We have tried to use benchmarks that are representative, meaningful and used by other publications. PigMix, for example is a standard benchmark used by Apache on every new release of the Pig runtime system. While the data sets we used for it are only in the GB order, such sizes are not atypical for big data in fact [32], and used also by others [36]. One factor to con-

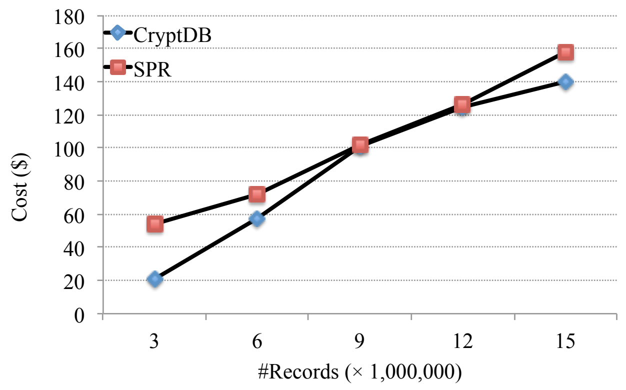


Figure 8: Comparison of SPR and CryptDB execution cost

sider with respect to big data analysis is that when volume of data becomes very high, characteristics on input data may heavily determine the time taken for completion.

7. DISCUSSION

Two big drawbacks of computing on encrypted data is bloating of data size and additional CPU cycles required to do more complicated arithmetic. For example, if we are using Paillier with 1024 bit length keys, a 64 bit `long` may get converted to a cipher text 2048 bits in length (an increase of 32 times) and an addition in plain text is replaced by multiplication of cipher text followed by a modulo operation in Paillier. Unfortunately we cannot avoid paying a higher price in terms of storage and computation based on state of art cryptosystems. Another aspect of security that we do not consider is that even though data is encrypted, an attacker can still see data access patterns. This includes frequency of data access, groups of data accessed together, operations done on data etc.

8. RELATED WORK

Approaches to protect confidentiality of data vary based on adversary models. Differential privacy[17, 16, 8] aims to improve the accuracy of statistical queries, without revealing information about individual records. The server which performs the query in this case is trusted with access to plain text data. This is in contrast to SPR that assumes data and computation reside in an untrusted server. sTile [10] 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 [29] 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. Excalibur [30] utilizes trusted platform modules to ensure cloud administrators cannot modify or examine data. Compared to SPR, Excalibur adds extra cost because the trusted platform modules need to be installed and maintained. However, SPR does not guarentee integrity. We plan to combine it with our previous work ClusterBFT [34] to ensure integrity as well as availability.

CryptDB [28] is a seminal system leveraging PHE for data management. As mentioned earlier CryptDB is a database system, focusing on SQL queries. CryptDB uses a trusted proxy that analyzes queries and decrypts columns to an encryption level suitable for query execution. It does not consider the more expressive kind of data flow programs as in Pig Latin, or MapReduce-style parallelization of queries. Monomi [37] 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 [36] 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 incompletely covered for evaluation.

Many static [22, 12] and runtime tools [38, 27] exist to assist programmers in discovering and fixing vulnerabilities in their code. However, these systems require the programmer to create queries or assertions to state security properties they want their programs to exhibit. This is not the case with SPR. A user simply submits her Pig Latin script, and after the transformations it meets our security properties.

Work has also been done on secure programming languages for distributed systems. These include languages like DSL [23] and SMCL [24], which include type systems that ensure that programs run securely. However, these languages focus on other issues such as interaction between different users, and (thus) do not support automatic parallelization via MapReduce or a similar framework.

9. CONCLUSIONS AND FUTURE WORK

In this paper, we presented original data flow analysis and program transformation used in SPR, a system able to perform big data analysis jobs on encrypted data, which allows us to leverage untrusted cloud resources while at the same time preserving confidentiality. Our program transformations translate Pig Latin scripts into semantically equivalent scripts that operate entirely on encrypted data. By automatically transforming Pig Latin scripts, developer efforts for achieving security are minimized.

We evaluated our approach through standard benchmarks and applications and demonstrated its applicability and performance. Our evaluations show that our approach scales well and can handle big volumes of encrypted data whilst retaining good performance. Furthermore, we gauge the savings in terms of developer effort of our automatic approach by comparing scripts that operate on unencrypted data with the transformed Pig Latin script which our system generates.

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 are investigating paths which perform re-encryption of data at clients (in contrast to client-side termination or costly re-encryption in the cloud as with FHE) and minimizing these re-encryptions themselves. To this end we are also employing sampling to determine amounts of data at different points in analysis jobs in order to better select between different options.

10. 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] HElib. `https://github.com/shaih/HElib`.

[5] The GNU Multiple Precision Arithmetic Library. `https://gmplib.org/`.

[6] Wikipedia database download. `http://en.wikipedia.org/wiki/Wikipedia:Database_download`.

[7] Zero MQ. `http://zeromq.org`.

[8] A. Blum, C. Dwork, F. McSherry, and K. Nissim. Practical Privacy: The SuLQ Framework. In *PODS*, pages 128–138, 2005.

[9] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. In *EUROCRYPT*, pages 224–241, 2009.

[10] Y. Brun and N. Medvidovic. Keeping Data Private while Computing in the Cloud. In *IEEE CLOUD*, pages 285–294, 2012.

[11] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *PLDI*, pages 363–375, 2010.

[12] A. Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–. USENIX Association, 2010.

[13] J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer Verlag, Berlin, Heidelberg, New York, 2002.

[14] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

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

[16] I. Dinur and K. Nissim. Revealing Information While Preserving Privacy. In *PODS*, pages 202–210, 2003.

[17] C. Dwork and K. Nissim. Privacy-Preserving Datamining on Vertically Partitioned Databases. In *CRYPTO*, pages 528–544, 2004.

[18] T. ElGamal. A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[19] C. Gentry, A. Sahai, and B. Waters. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *CRYPTO*, volume 1, pages 75–92, Aug. 2013.

[20] Hadoop. Hadoop. `http://hadoop.apache.org/`.

[21] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *EuroSys*, 2007.

[22] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: A program query language. *SIGPLAN Not.*, 40(10):365–383, Oct. 2005.

[23] J. Mitchell, R. Sharma, D. Stefan, and J. Zimmerman. Information-flow control for programming on encrypted data. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, pages 45–60, June 2012.

[24] J. D. Nielsen and M. I. Schwartzbach. A domain-specific programming language for secure multiparty computation. In *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security*, pages 21–30. ACM, 2007.

[25] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD*, pages 1099–1110, 2008.

[26] P. Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *EUROCRYPT*, pages 223–238, May 1999.

[27] B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen, and A. Perrig. Clamp: Practical prevention of large-scale data leaks. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 154–169. IEEE Computer Society, 2009.

[28] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *SOSP*, pages 85–100, 2011.

[29] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and Privacy for MapReduce. In *NSDI*, pages 297–312, 2010.

[30] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *Proceedings of the 21st USENIX Conference on Security Symposium*, pages 10–10. USENIX Association, 2012.

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

[32] M. Schwarzkopf, D. Murray, and S. Hand. The Seven Deadly Sins of Cloud Computing Research. In *HotClouds*, 2012.

[33] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *MSST*, pages 1–10, 2010.

[34] J. J. Stephen and P. Eugster. Assured Cloud-Based Data Analysis with ClusterBFT. In *Middleware*, pages 82–102, 2013.

[35] J. J. Stephen, S. Savvides, R. Seidel, and P. Eugster. Practical Confidentiality Preserving Big Data Analysis. In *HotCloud*, 2014.

[36] S. Tetali, M. Lesani, R. Majumdar, and T. Millstein. MrCrypt: Static Analysis for Secure Cloud Computations. In *OOPSLA*, pages 271–286, 2013.

[37] S. Tu, F. Kaashoek, S. Madden, and N. Zeldovich. Processing Analytical queries over encrypted data. In *PVLDB*, pages 289–300, 2013.

[38] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 291–304. ACM, 2009.

[39] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. HotCloud'10, 2010.