MrCrypt: Static Analysis for Secure Cloud Computations

Sai Deep Tetali
UC Los Angeles
saideep@cs.ucla.edu

Mohsen Lesani
UC Los Angeles
lesani@cs.ucla.edu

Rupak Majumdar
MPI-SWS
rupak@mpi-sws.org

Todd Millstein
UC Los Angeles
todd@cs.ucla.edu

Abstract

In a common use case for cloud computing, clients upload data and computation to servers that are managed by a third-party infrastructure provider. We describe MrCrypt, a system that provides data confidentiality in this setting by executing client computations on encrypted data. MrCrypt statically analyzes a program to identify the set of operations on each input data column, in order to select an appropriate homomorphic encryption scheme for that column, and then transforms the program to operate over encrypted data. The encrypted data and transformed program are uploaded to the server and executed as usual, and the result of the computation is decrypted on the client side. We have implemented MrCrypt for Java and illustrate its practicality on three standard benchmark suites for the Hadoop MapReduce framework. We have also formalized the approach and proven several soundness and security guarantees.

1. Introduction

A common use case for cloud computing involves clients uploading data and computation to servers managed by third-party infrastructure providers. Since the data and programs are no longer in an environment controlled by the client, private client data may be exposed to adversarial clients in the cloud server, either by accidental misconfigurations or through malicious intent. Publicized incidents involving the loss of confidentiality or integrity of customer data [20] only heighten these concerns. The threat of potential violations to the confidentiality and integrity of customer data is a key barrier to the adoption of cloud computing based on third-party infrastructure providers.

One way to alleviate these concerns is to store encrypted data on the cloud and decrypt it as needed during the cloud computation. However, this approach is insufficient to protect against adversaries who can potentially view the memory contents of the server, for example a curious cloud administrator or a malicious client running on the same machine. Therefore, all computations must be performed on the client side [22, 23], which severely reduces the attractiveness of the cloud model. Theoretically, fully homomorphic encryption schemes [14, 34] offer the possibility of uploading and storing encrypted data on the cloud and performing arbitrary operations on the encrypted data. Unfortunately, current implementations of fully homomorphic encryption schemes are still prohibitively expensive [15, 16].

In this paper we present MrCrypt, a system that automatically transforms programs in order to enforce data confidentiality. Our key insight is that many useful cloud computations only perform a small number of operations on each column of the data. While fully homomorphic encryption is expensive, there are efficient encryption schemes that support common subsets of operations. Thus, instead of encoding each column using a fully homomorphic encryption scheme, one can encrypt it using a more efficient scheme that supports only the necessary operations. For example, suppose that the client program sums all the elements of a column. Instead of a fully homomorphic encryption scheme, one can encrypt the column using the Paillier cryptosystem [29], which guarantees that

$$\mathsf{Paillier}(x) \cdot \mathsf{Paillier}(y) = \mathsf{Paillier}(x + y)$$

for any x, y, where $\mathsf{Paillier}(x)$ denotes the encryption of x using the scheme, and the multiplication of the codewords on the left is modulo a public key. Thus to compute on the encrypted data, the program must simply take the product of all codewords. When the result is decrypted on the client side, the sum of all the numbers is recovered. Similarly, the El Gamal cryptosystem [10] is homomorphic for multiplication:

$$\mathsf{ElGamal}(x) \cdot \mathsf{ElGamal}(y) = \mathsf{ElGamal}(x \cdot y)$$

In this way, MrCrypt reduces the problem of securing cloud computations to that of identifying the subset of primitive operations (such as addition, multiplication, equality checks, and order comparisons) performed on each column of the input, in order to determine the most efficient encryption scheme that can be used for the column. We have developed a static analysis to perform this task, which we call *encryption scheme inference*, on imperative programs. We formalize the problem and our solution as a variant of type qualifier inference [11], where each qualifier lattice element represents a particular homomorphic encryption scheme.

We have implemented MrCrypt for Java. Given a Java program and a lattice of encryption schemes, MrCrypt first performs encryption scheme inference to determine the most efficient scheme for each input data column, based on the operations performed by the program on that column. MrCrypt then performs a source-to-source transformation of the program to compute on the encrypted data rather than the original plaintext. Finally, the transformed program can be sent to the cloud and run on an encrypted version of the original data.

In our experiments, we evaluate MrCrypt's practicality by applying it to Java programs that run on the MapReduce framework [9] in Hadoop[1]. MapReduce is a natural target because it is a popular programming model for cloud computing, and the Hadoop implementation of MapReduce is widely used. Further, many useful MapReduce programs require only a small number of operations on each data column and so fit the requirements of our approach. The transformed programs produced by MrCrypt are compliant Hadoop MapReduce programs and so can run directly on unmodified Hadoop infrastructure in the cloud.

We have evaluated MrCrypt on three standard MapReduce benchmark suites [1, 27, 31] and evaluated our system on large datasets (up to 50GB) provided with the PUMA benchmarks [1]. On 24 of 36 benchmarks, MrCrypt can identify encryption schemes to support the necessary functionality without requiring fully homomorphic encryption. For the large dataset examples, encrypted execution takes 2.61 times as long on average as the plaintext versions and requires 3.92 times as much space for input data. However, ignoring one outlier benchmark, the benchmarks take only 1.57 times as long on average as the plaintext versions and require 2.88 times as much space.

The closest related work to MrCrypt is CryptDB [32], a system that interposes between a trusted application and an untrusted database server. CryptDB dynamically rewrites each SQL query generated by an application to work over homomorphically encrypted data. MrCrypt also relies on homomorphic encryption for security, but in the setting of cloud computing, which demands several important design differences. Specifically, MrCrypt must perform encryption scheme inference statically, must rewrite an entire application to work over encrypted data, and must be able to handle imperative Java code rather than declarative SQL. We compare with CryptDB and other related work in more detail in Section 7.

Like CryptDB, MrCrypt only considers threats against data confidentiality. In particular, MrCrypt does not provide guarantees of data integrity or completeness of results, which are orthogonal issues and topics for future work.

The rest of the paper is structured as follows. In the next section we provide necessary background on homomorphic encryption as well as the MapReduce programming model. Section 3 overviews MrCrypt and provides an illustrative example. In Section 4, we formalize the encryption scheme inference problem and present soundness and security guarantees. We explain our implementation in Section 5 and present the evaluation results in Section 6. Finally, we present related work and conclusions.

2. Background

2.1 Homomorphic Encryption Schemes

A (public-key) encryption scheme consists of three algorithms (K, E, D) for key-generation, encryption, and decryption. The key-generation procedure K is a randomized algorithm that takes a security parameter λ as input and outputs a secret key sk and public key pk. The encryption procedure E is a randomized algorithm that takes pk and a plaintext m as input and outputs a ciphertext ψ. The decryption procedure D takes sk and ψ as input and outputs the plaintext m, i.e., $D(sk, E(pk, m)) = m$. The computational complexity of all of these algorithms must be polynomial in λ.

Given a binary operation f, an encryption scheme is *homomorphic for* f if there exists a (possibly randomized) polynomial-time algorithm $Eval_f$, which takes as input the public key pk and a pair of ciphertexts (ψ_1, ψ_2) such that

$$D(sk, Eval_f(pk, \psi_1, \psi_2)) = f(D(sk, \psi_1), D(sk, \psi_2))$$

Informally, if ψ_1 and ψ_2 are respectively encryptions of plaintexts m_1 and m_2 under pk, then $Eval_f(pk, \psi_1, \psi_2)$ is an encryption of $f(m_1, m_2)$ under pk. For a set of operations F, an encryption scheme is homomorphic for F if it is homomorphic for each $f \in F$. An encryption scheme is said to be *fully homomorphic* if it is homomorphic for $\{+, \times\}$. It is easy to see that in this case, any polynomial-sized arithmetic circuit can be evaluated purely on the ciphertext.

In addition to homomorphic encryption schemes, we shall also consider encryption schemes with a related property in which the result of an operation can be computed (in clear text) directly on the ciphertext:

$$Eval_f(pk, \psi_1, \psi_2) = f(D(sk, \psi_1), D(sk, \psi_2))$$

For example, using a deterministic encryption scheme, one can check if two values are equal simply by comparing the ciphertexts for equality, without requiring any information about the original values. In the following, we abuse notation and call such encryption schemes "homomorphic" as well.

Given a set of operations $\mathcal{F}$, one can arrange encryption schemes in a partial order: an encryption scheme $\mathcal{E}_1$ is "less than" a scheme $\mathcal{E}_2$ if $\mathcal{E}_1$ is homomorphic for $F_1 \subseteq \mathcal{F}$, $\mathcal{E}_2$ is

[1] http://hadoop.apache.org/

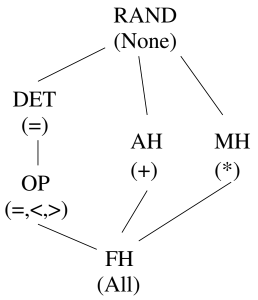

Figure 1. A lattice of encryption schemes.

homomorphic for $F_2 \subseteq \mathcal{F}$, and $F_2 \subseteq F_1$. A fully homomorphic encryption scheme is the unique minimal element in this ordering, and a random encryption scheme is the maximal element (it is not homomorphic for any operation). Typically, one expects that encryption schemes "higher" in the ordering (i.e., supporting fewer operations) will have more efficient implementations.

MrCrypt's implementation is parameterized by a lattice of encryption schemes. Our experiments in Section 6 employ several forms of encryption, which are shown as a lattice in Figure 1 along with the set of operations that each scheme supports. $RAND$ (random) supports no homomorphic operations [40]; DET (deterministic) supports equality testing [40]; OP (order-preserving) supports comparisons [5, 6]; AH (additive homomorphic) supports addition [29]; MH (multiplicative homomorphic) supports multiplication [10]; FH (fully homomorphic) supports all operations [14]. The DET and OP schemes produce their results in clear text, while the other schemes are homomorphic in the strict sense. Because fully homomorphic encryption is not currently practical, MrCrypt does not include an implementation of it, but we show that it is rarely required in our benchmark programs.

2.2 MapReduce

MapReduce [9] is a popular distributed programming model introduced by Google for processing large data sets on clusters. In this model, the computation is divided into three stages. The map stage invokes a user-defined *map* function in parallel over the data and produces a list of intermediate key/value pairs. A *shuffle* stage in the MapReduce framework then sorts all the resulting records based on the keys and groups together all values associated with the same key. Finally, the reduce stage invokes a user-defined *reduce* function in parallel to combine the values associated with each key in some fashion, typically producing just zero or one final values per key. Hadoop MapReduce is an open-source implementation of MapReduce that is widely used by both researchers and major corporations (e.g., Facebook, Twitter) to perform large-scale distributed computations.

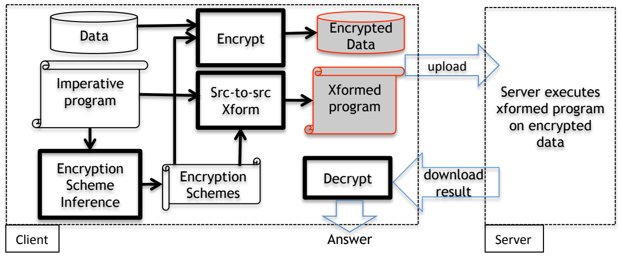

Figure 2. Architecture of MrCrypt; solid boxes show implemented components .

3. Overview

3.1 MrCrypt Architecture and Threat Model

The architecture of MrCrypt is shown in Figure 2. Given a Java program and a data set, MrCrypt performs static analysis on the program to determine an encryption scheme for each input column, program variable, and program constant such that each program operation can be performed homomorphically on encrypted data. We call this analysis *encryption scheme inference*. The analysis first generates constraints based on how operations in the program are used, and then it solves the constraints to determine the most efficient (i.e., highest in the lattice) encryption scheme to use for each part of the program.

Next the results of the encryption scheme inference are used to transform the program. Specifically, each call to a primitive operation f in the program is replaced by a call to $Eval_f^{\mathcal{E}}$, where $\mathcal{E}$ is the encryption scheme inferred for the arguments to f. Similarly, each program constant c is replaced by its encrypted value $E^{\mathcal{E}}(c)$, where $\mathcal{E}$ is the encryption scheme inferred for c. The data set is also encrypted according to the inferred encryption schemes on the client side.

Finally, the encrypted data and the transformed program are uploaded to the (untrusted) server, where the program is executed. The result is sent back to the client, where it is decrypted.

We assume a passive (honest-but-curious) adversary model. That is, the adversary can view all the data uploaded to the server, the program that is uploaded, as well as the entire execution trace of the program. However, we assume that the adversary does not change the data, the program, or the result of the program (i.e., no integrity attacks).

3.2 Applications

MrCrypt is potentially useful for any cloud computation, but we expect it to be especially applicable to scenarios where a small set of computations are run iteratively over a large evolving data set. In these cases, the client-side encryption required by MrCrypt can be performed incrementally as the

Program 1 An example MapReduce program.

```
1  Integer map(Integer entryDate, Integer
       entryMonth, Integer entryYear, Integer
        caloriesBurnt) {
2    if (entryYear > 2012)
3      return caloriesBurnt;
4    else
5      return 0;
6  }
7  Integer reduce(List<Integer>
       caloriesBurntList) {
8    Integer sum = 0;
9    for (Integer caloriesBurnt :
         caloriesBurntList)
10     sum += caloriesBurnt;
11   return sum;
12 }
```

Program 2 Translated Program

```
AH_Integer map(RAND_Integer entryDate,
    RAND_Integer entryMonth, OP_Integer
    entryYear,  AH_Integer caloriesBurnt)
    {
  if (OP_GT(entryYear, [[OP_E(2012)]])
    return caloriesBurnt;
  else
    return [[AH_E(0)]];
}
AH_Integer reduce(List<AH_Integer>
    caloriesBurntList) {
 AH_Integer sum = [[AH_E(0)]];
  for (AH_Integer caloriesBurnt :
      caloriesBurntList)
    sum = AH_PLUS(sum, caloriesBurnt);
  return sum;
}
```

data is generated, and the encryption costs are amortized over multiple runs of the cloud computations.

A broad class of applications have these characteristics. For example, health-monitoring systems continuously upload vital statistics of a user to the cloud, where running computations are performed on the data to alert users or caregivers when certain conditions hold. Companies such as FitBit, Jawbone, and Nike have devices in the market which track several metrics such as distance walked, calories burnt, and sleep patterns and upload them continuously to the cloud in order to compute aggregates. As another example, these characteristics apply to sensor networks in which a distributed set of sensors continually upload data to the cloud in order to run analytics.

3.3 Example

Program 1 is inspired by wireless fitness trackers. Users continually upload fitness information such as the number of calories burnt during a workout to the cloud. This program uses the MapReduce programming model to compute the total number of calories burnt by a user since the beginning of the year 2013. This result can be used further to compute statistics such as the average calories burnt per day. Every record includes the number of calories burnt and the date associated with the event (given by year, month and day fields). For expository purposes we omit some implementation details required by MapReduce frameworks, for example the need to parse input data from a file and to produce key-value pairs as results. However, the example is illustrative of common MapReduce use cases.

The user-defined map function is executed on each row of the data, and it has the effect of producing all entries from the $caloriesBurnt$ column for which the associated entry year is greater than 2012. The MapReduce framework collects the values returned by the map function invocations and passes the resulting list to the user-defined reduce function, which sums the calories.

Encryption Inference Consider our example in Program 1. For a variable x, let $\sigma(x)$ denote the encryption scheme for x, and similarly for a constant c. When necessary to disambiguate, we subscript a variable or constant with the name of the function in which it appears. From line 2, it is concluded that $\sigma(\text{entryYear}) = \sigma(2012)$ and that $\sigma(\text{entryYear})$ should support at least $>$. From lines 3 and 5, $\sigma(\text{caloriesBurnt}_{\text{map}}) = \sigma(0_{\text{map}})$. From line 8, $\sigma(\text{sum}) = \sigma(0_{\text{reduce}})$. From line 10, $\sigma(\text{sum}) = \sigma(\text{caloriesBurnt}_{\text{reduce}})$ and $\sigma(\text{sum})$ should support at least $+$. Finally, the semantics of the MapReduce framework requires that the result from the map function must use the same encryption scheme as the data items in the argument list to the reduce function. Given the lattice of encryption schemes from Figure 1, the best solution to these constraints maps $\sigma(\text{entryYear})$ and $\sigma(2012)$ to OP, $\sigma(\text{entryDate})$ and $\sigma(\text{entryMonth})$ to $RAND$ (since there are no constraints on these variables), and everything else to AH.

Given the results of encryption scheme inference, it is straightforward to produce the translated program that will be sent to the cloud along with the encrypted data. The translated program for the example is shown in Program 2. First, the primitive $>$ function is replaced by the corresponding operation in the order-preserving encryption scheme, which we denote `OP_GT`, and similarly $+$ is replaced by `AH_PLUS`. Second, each constant is replaced by an appropriately encrypted version of that constant. For example, we use `[[OP_E(2012)]]` to denote the encrypted value of the constant 2012 under the order-preserving encryption scheme. Note that this value is computed statically and inserted into the transformed program in place of the original constant.

3.4 Properties

As described in the next section, we have formalized the constraint generation phase of encryption scheme inference. (The constraints can be solved using well-known techniques [11].) We have proven that the generated constraints are *sound*: any solution to the constraints ensures that an encryption scheme is only asked to perform operations that it supports and that the operands to a homomorphic operation are encrypted with that same encryption scheme. We have also proven that the generated constraints are *complete*, so in particular they are compatible with the unique most efficient mapping that satisfies the above properties, according to the given lattice.

By design our approach never sends plaintext data or program constants to the server. Additionally, we can prove a strong security guarantee for MrCrypt. Informally, our result says that a polynomial-time adversary gets no advantage by having access to the transformed program and its (encrypted) inputs over and above having access to the encrypted input data alone. Therefore, the security guarantees of our framework are equivalent to those provided by the underlying homomorphic encryption schemes.

The semantic security of encryption schemes is formalized using notions of indistinguishability [17]. Intuitively, these guarantees say that an adversary cannot recover which one of two plaintexts a given ciphertext corresponds to, with better odds than flipping a fair coin. The $RAND$, AH, and MH schemes ensure this strong indistinguishability property. The DET and OP schemes provide weaker guarantees. For example, given an input column encrypted with DET, an attacker can easily determine whether two rows hold the same value or not. However, DET satisfies a natural weakening of indistinguishability ensuring that the results of such equality tests are the only things revealed to an adversary. The situation for the OP scheme is analogous. While these schemes offer weaker guarantees, in practice using them on data of high entropy provides a useful level of confidentiality. We show a multicast security result, following [2], that an adversary can learn no more facts about the original input than can be learnt from the encryption of any individual input value (i.e., there is no additional security loss in our system).

4. Encryption Scheme Inference

This section formally defines the encryption scheme inference problem on an extended simply-typed lambda calculus, formalizes our solution to the problem, and proves various correctness and security properties of the approach. The full formal details are available in a companion technical report [21].

Our formalism is parameterized by a set $\mathcal{M}$ of arithmetic operations and a set $\mathcal{R}$ of logical predicates, whose union we denote $\mathcal{O}$. The formalism is also parameterized by a lattice L of encryption schemes, each of which supports some subset of the operations in $\mathcal{O}$, with associated partial order $\sqsubseteq$. We assume that if $l_1 \sqsubseteq l_2$ and encryption scheme l_2 supports some operation $\oplus \in \mathcal{O}$, then l_1 also supports that operation. Also, for each operation $\oplus \in \mathcal{O}$ we assume there is a unique maximal element of L that supports $\oplus$, which we denote $l_\oplus$. In our implementation, $\mathcal{M} = \{+, \times\}$ and $\mathcal{R} = \{<, =, >\}$.

4.1 Syntax

The syntax of expressions is as follows:

$$\begin{array}{rcll} e & ::= & v \mid x \mid e_1\, e_2 \mid e_1 \oplus e_2 & \text{Program} \\ v & ::= & \lambda x{:}\rho.e \mid n \mid n_l & \text{Value} \end{array}$$

where $\oplus \in \mathcal{O}$. Metavariable x ranges over variables and n over integer constants. The value n_l denotes the value resulting from encrypting integer n with the encryption scheme l.

The set of types is defined as follows:

$$\begin{array}{rcll} \rho & ::= & \kappa\, \tau & \text{Qualified Type} \\ \kappa & ::= & l \mid \circ \mid \gamma & \text{Qualifier} \\ \tau & ::= & Int \mid \alpha \mid \rho \rightarrow \rho & \text{Type} \end{array}$$

Types are qualified with a lattice element l, which tracks the encryption scheme used for each program expression. Metavariable γ ranges over qualifier variables and α over type variables. For uniformity we use $\circ$ to denote the qualifier for unencrypted data. Since any operation is supported on unencrypted data, $\sqsubseteq$ is extended such that $\forall l \in L : \circ \sqsubseteq l$.

4.2 Operational Semantics

We define two operational semantics for the language: one for programs that expect plaintext inputs and another for programs that expect encrypted inputs. The former semantics models execution of the original programs while the latter semantics models execution of those programs after being transformed by MrCrypt. The reduction contexts for both semantics are standard and defined as follows:

$$R \;\; ::= \;\; [\,] \mid R\, e \mid v\, R \mid R \oplus e_2 \mid v \oplus R$$

The plaintext and ciphertext operational semantics relations, respectively denoted $\rightarrow_p$ and $\rightarrow$, are shown in Figures 3 and 4. We use $[[\oplus]]$ to denote the mathematical function corresponding to operation $\oplus$, with the relational operations in $\mathcal{R}$ returning the integer 1 to denote true and 0 to denote false. The plaintext semantics requires the operands to a mathematical operation to be unencrypted. In contrast, the ciphertext semantics requires the operands to a mathematical operation $\oplus$ to be encrypted with the same encryption scheme l, requires that l supports $\oplus$, and specifies that the operations in $\mathcal{M}$ result in encrypted values.

4.3 Typing and Type Inference

Figure 5 defines the typing judgment of the form $\Gamma \vdash e\colon \rho$, where as usual the type environment Γ maps variables to (qualified) types. The rules are mostly standard. The key new requirements are that the operands to an operation $\oplus$ have the same encryption scheme (qualifier) and that this encryption scheme supports the operation. The type qualifier

App-Trans

$$R[(\lambda x{:}\rho.e)\ v] \rightarrow R[e[x \mapsto v]]$$

Math-Trans

$$\frac{n\ [[\oplus]]\ n' = n'' \qquad \oplus \in \mathcal{M}}{R[n\ \oplus\ n'] \rightarrow_p R[n'']}$$

Rel-Trans

$$\frac{n'' = \begin{cases} 1 & \text{if } (n\ [[\oplus]]\ n') \\ 0 & \text{if } \neg(n\ [[\oplus]]\ n') \end{cases} \qquad \oplus \in \mathcal{R}}{R[n\ \oplus\ n'] \rightarrow_p R[n'']}$$

Figure 3. Plaintext Operational Semantics

Q-App-Trans

$$R[(\lambda x{:}\rho.e)\ v] \rightarrow R[e[x \mapsto v]]$$

Q-Math-Trans

$$\frac{n\ [[\oplus]]\ n' = n'' \qquad \oplus \in \mathcal{M} \qquad l \sqsubseteq l_\oplus}{R[n_l\ \oplus\ n'_l] \rightarrow R[n''_l]}$$

Q-Rel-Trans

$$\frac{n'' = \begin{cases} 1 & \text{if } (n\ [[\oplus]]\ n') \\ 0 & \text{if } \neg(n\ [[\oplus]]\ n') \end{cases} \qquad \oplus \in \mathcal{R} \qquad l \sqsubseteq l_\oplus}{R[n_l\ \oplus\ n'_l] \rightarrow R[n'']}$$

Figure 4. Ciphertext Operational Semantics

Q-Lam

$$\frac{\Gamma[x \mapsto \rho_1] \vdash e{:}\ \rho_2}{\Gamma \vdash \lambda x{:}\rho_1.e{:}\ \circ\,(\rho_1 \rightarrow \rho_2)}$$

Q-App

$$\frac{\Gamma \vdash e_1{:}\ \kappa\,(\rho_1 \rightarrow \rho_2) \qquad \Gamma \vdash e_2{:}\ \rho_1}{\Gamma \vdash e_1\ e_2{:}\ \rho_2}$$

Q-Int-L

$$\Gamma \vdash n_l{:}\ l\ Int$$

Q-Math

$$\frac{\Gamma \vdash e'{:}\ \kappa\ Int \qquad \Gamma \vdash e''{:}\ \kappa\ Int \qquad \kappa \sqsubseteq l_\oplus \qquad \oplus \in \mathcal{M}}{\Gamma \vdash e'\ \oplus\ e''{:}\ \kappa\ Int}$$

Q-Rel

$$\frac{\Gamma \vdash e'{:}\ \kappa\ Int \qquad \Gamma \vdash e''{:}\ \kappa\ Int \qquad \kappa \sqsubseteq l_\oplus \qquad \oplus \in \mathcal{R}}{\Gamma \vdash e'\ \oplus\ e''{:}\ \circ\ Int}$$

Q-Int

$$\Gamma \vdash n{:}\ \circ\, Int$$

Q-Var

$$\frac{\Gamma(x) = \rho}{\Gamma \vdash x{:}\ \rho}$$

Figure 5. Typing Rules

for a lambda abstraction is $\circ$ because we never encrypt functions (but rather just the numeric data manipulated by those functions).

Finally, we formalize the encryption scheme inference problem as a form of type inference with qualifiers. We define a judgment of the form[2] $\Gamma \vdash e{:}\ \rho; C$ where C is a set of constraints of the following form:

$$C \quad ::= \quad \{\tau_1 = \tau_2\} \ \mid\ \{\kappa_1 = \kappa_2\} \ \mid\ \{\gamma \sqsubseteq l\} \ \mid\ C_1 \cup C_2$$

We treat equality constraints of the form $\kappa_1\ \tau_1 = \kappa_2\ \tau_2$ as shorthand for the two constraints $\{\kappa_1 = \kappa_2, \tau_1 = \tau_2\}$.

The type inference rules are shown in Figure 6. The rules "produce" constraints on the qualified types of all program expressions. A substitution σ, which maps type and qualifier variables to types and qualifiers, is a *solution* to constraints C if $\sigma(c)$ is satisfied for each $c \in C$. A solution to the constraints produced by type inference therefore represents a solution to the encryption scheme inference problem: it determines the encryption scheme needed for each program expression in order to make the program typecheck. The constraints have a unique best solution (maximal in the lattice L) and can be solved using standard techniques [11]. Note that while constants in our formalism are explicitly annotated with their encryption scheme, this is no loss of generality since we can always model program constants instead as extra parameters to the program.

4.4 Soundness Properties

We have proven several correctness properties for our formalism. First, we have proven type soundness through the standard "progress and preservation" style [42]:

Lemma 1 (Progress). *If $\emptyset \vdash e{:}\ \rho$, then either e is a value or there is some e' such that $e \rightarrow e'$.*

Lemma 2 (Preservation). *If $\Gamma \vdash e{:}\ \rho$ and $e \rightarrow e'$ then $\Gamma \vdash e'{:}\ \rho$.*

Type soundness ensures that a well-typed program never gets stuck at run time, which in our setting implies that an encryption scheme is only asked to perform operations that it supports and operations are always applied to operands that use the same encryption scheme.

Second, we have proven that execution of a transformed program in the ciphertext semantics is equivalent to execution of the original program in the plaintext semantics. Given an expression e we define $decr(e)$ to be the expression iden-

[2] In our technical report [21], the judgment also records the set of generated type and qualifier variables, which is necessary for the formal proof of completeness; we elide this here for presentation purposes.

Q-LAM-INF

$$\frac{\begin{array}{c}\Gamma[x \mapsto \rho_1] \vdash e\colon \rho_2; C \\ C' = C \cup \{\gamma = \circ, \alpha = \rho_1 \to \rho_2\} \\ \gamma, \alpha \text{ fresh}\end{array}}{\Gamma \vdash \lambda x{:}\rho_1.e\colon (\gamma\ \alpha); C'}$$

Q-APP-INF

$$\frac{\begin{array}{c}\Gamma \vdash e_1\colon \rho_1; C_1 \qquad \Gamma \vdash e_2\colon \rho_2; C_2 \\ C = C_1 \cup C_2 \cup \{\rho_1 = \gamma'\ (\rho_2 \to (\gamma\ \alpha))\} \\ \gamma, \gamma', \alpha' \text{ fresh}\end{array}}{\Gamma \vdash e_1\ e_2\colon (\gamma\ \alpha); C}$$

Q-MATH-INF

$$\frac{\begin{array}{c}\oplus \in \mathcal{M} \\ \Gamma \vdash e_1\colon (\kappa_1\ \tau_1); C_1 \qquad \Gamma \vdash e_2\colon (\kappa_2\ \tau_2); C_2 \\ C = C_1 \cup C_2 \cup \{\kappa_1 = \kappa_2 = \gamma, \gamma \sqsubseteq l_{\oplus}, \tau_1 = \tau_2 = Int = \alpha\} \\ \gamma, \alpha \text{ fresh}\end{array}}{\Gamma \vdash (e_1\ \oplus\ e_2)\colon (\gamma\ \alpha); C}$$

Q-REL-INF

$$\frac{\begin{array}{c}\oplus \in \mathcal{R} \\ \Gamma \vdash e_1\colon (\kappa_1\ \tau_1); C_1 \qquad \Gamma \vdash e_2\colon (\kappa_2\ \tau_2); C_2 \\ C = C_1 \cup C_2 \cup \{\kappa_1 = \kappa_2, \gamma = \circ, \kappa_1 \sqsubseteq l_{\oplus}, \tau_1 = \tau_2 = Int = \alpha\} \\ \gamma, \alpha \text{ fresh}\end{array}}{\Gamma \vdash (e_1\ \oplus\ e_2)\colon (\gamma\ \alpha); C}$$

Q-VAR-INF

$$\frac{\Gamma(x) = \rho}{\Gamma \vdash x\colon \rho; \emptyset}$$

Q-INT-L-INF

$$\frac{C = \{\gamma = l, \alpha = Int\} \qquad \gamma, \alpha \text{ fresh}}{\Gamma \vdash n_l\colon (\gamma\ \alpha); C}$$

Q-INT-INF

$$\frac{C = \{\gamma = \circ, \alpha = Int\} \qquad \gamma, \alpha \text{ fresh}}{\Gamma \vdash n\colon (\gamma\ \alpha); C}$$

Figure 6. Type Inference Rules

tical to e but with each occurrence of a value of the form n_l replaced by n. Intuitively, $decr(e)$ is the plaintext version of e. The following result formalizes this intuition.

Theorem 1. *1.* **[Encryption Domain Soundness]** *If $e \to e'$, then $decr(e) \to_p decr(e')$.*

2. **[Encryption Domain Completeness]** *If $\emptyset \vdash e_1\colon \rho$, $e'_1 = decr(e_1)$ and $e'_1 \to_p e'_2$, then there is some e_2 such that $e_1 \to e_2$ and $e'_2 = decr(e_2)$.*

Finally, we have proven that our type inference rules are sound and complete with respect to our typing rules, which means the generated constraints are compatible with only and all valid types of the program.

Theorem 2. *1.* **[Soundness of Type Inference]** *If $\Gamma \vdash e\colon \rho; C$ and σ is a solution to C, then $\sigma(\Gamma) \vdash \sigma(e)\colon \sigma(\rho)$.*

2. **[Completeness of Type Inference]** *If $\Gamma \vdash e\colon \rho; C$ and there is a substitution σ such that $\sigma(\Gamma) \vdash \sigma(e)\colon \rho'$, then there is a solution σ' to C such that $\sigma'(\rho) = \rho'$.*

4.5 Security Guarantees

We assume an honest-but-curious adversary model, where the server observes the data, the program, and the program execution and can perform polynomial-time computation over the observations. However, the server does not change the data or the computation. One caveat is that the server should run in polynomial time in the size of the data and the input, but not in the potentially exponential program trace. If we allow the adversary to run in time polynomial in the program trace, it may be able to execute an exponentially long computation in the security parameter, and so to decrypt all the encrypted values trivially.

We formalize our security guarantees in terms of *indistinguishability* [17]. Indistinguishability is formalized using an *adversary* $A = (A_1, A_2)$, performing a sequence of two (potentially randomized) polynomial-time algorithms. Initially keys $(pk, sk) = K(\lambda)$ are generated based on a security parameter λ. First, algorithm A_1 takes as input the public key pk and outputs two plaintext messages x_0 and x_1, together with some additional state information s. Next, a bit $b \in \{0, 1\}$ is chosen at random, and message x_b is encrypted as a challenge ciphertext y using pk. Finally, algorithm A_2 runs on (y, s) and has to guess the bit b. The advantage of the adversary is defined as

$$Adv_{\mathcal{E}}(A) = \Pr[A_2(y, s) = b] - \frac{1}{2}$$

where the random variables are distributed uniformly.

An encryption scheme $\mathcal{E} = (K, E, D)$ satisfies single-use *indistinguishability* against chosen plaintext attacks (IND-CPA) if for each adversary A we have that $Adv_{\mathcal{E}}(A)$ is negligible (recall that a function $f(n)$ is negligible if $|f(n)| < \frac{1}{poly(n)}$ for all sufficiently large n). Intuitively, a polynomial-time adversary cannot identify the plaintext from a ciphertext with advantage significantly better than that obtained by flipping a coin. For example, it is known that the El Gamal and Paillier cryptosystems satisfy IND-CPA.

Unfortunately, IND-CPA is too strong a requirement for deterministic encryption schemes: for example, the adversary can store the encryptions of x_0 and x_1 and compare the challenge ciphertext y against the stored ciphertexts. Similarly, IND-CPA is too strong for order-preserving schemes. Thus, one defines weaker notions of indistinguishability for such schemes. We omit detailed definitions (see, e.g., [3, 5, 6]), but assume that each individual encryption scheme $\mathcal{E}$ has an associated indistinguishability property IND($\mathcal{E}$).

In our context, we have a set of inputs $x_1, \dots, x_n$ to the program and use possibly different encryption schemes $\mathcal{E}_1, \dots, \mathcal{E}_n$ for them. We ask, given that each scheme $\mathcal{E}_i$ satisfies IND($\mathcal{E}_i$), what we can guarantee about the full encrypted data. To do this, we define the notion of *program-indistinguishability* for a tuple of encryption schemes (see our companion technical report [21] for full details). Intuitively, the adversary now chooses two sequences of plaintexts, according to possible restrictions placed by the IND conditions. Now one of the two is chosen at random and componentwise encoded using its encryption scheme. The adversary has to guess which of the two sequences was encoded by looking at the encrypted vector. Notice that we do not consider the encrypted program in the definition, since the adversary can perform an arbitrary polynomial-time computation; in particular, it can run the program for a polynomial number of steps. The following theorem restates a result from [2].

Theorem 3. *Given encryption schemes $\mathcal{E}_i$ satisfying IND($\mathcal{E}_i$) for $i = 1, \dots, n$, $(\mathcal{E}_1, \dots, \mathcal{E}_n)$ is program-indistinguishable.*

Thus, MrCrypt provides a security guarantee that is as strong as the individual encryption schemes used for each data item.

5. Implementation

We have implemented our encryption scheme inference and transformation algorithms for Java programs.

5.1 Encryption Schemes

We briefly describe the encryption schemes that are currently supported in MrCrypt. Since there is no efficient scheme for FH currently, our system throws an exception if FH is required. (Our experimental evaluation shows that this is rarely the case.) In general, we follow the security parameters from prior work [32].

RAND is a probabilistic encryption scheme that guarantees IND-CPA but which does not support any operations on the encrypted data. We implement $RAND$ using Blowfish [39] for 32-bit integers and AES [8] for strings in CBC mode and with a random initialization vector. Blowfish produces a 64-bit ciphertext and AES outputs ciphertext as 128-bit blocks.

DET is a deterministic encryption scheme: the same plaintext generates the same ciphertext. Thus, DET allows checking for equality on the encrypted values. We make the standard assumption that Blowfish and AES block ciphers are pseudorandom permutations and use these encryption schemes in ECB mode. For values up to 64 and 128 bits, we use Blowfish and AES respectively after padding smaller plaintexts to at least 64 bits. For longer strings, we use AES with a variant of CMC mode [18] with a zero initial vector, as is done in CryptDB [32].

OP is an order-preserving encryption scheme that allows checking order relations between encrypted data items. We use the implementation of OP in CryptDB, which follows the algorithm in [5]. Since we only do order operations on 32-bit integers, we use a ciphertext size of 64 bits for each value.

AH allows performing addition on encrypted data. We use CryptDB's implementation of the Paillier cryptosystem [29] to support AH. We generate 512 bits of ciphertext for each 32-bit value.

MH allows performing multiplication on encrypted data. We use the El Gamal cryptosystem [10] to support MH. We use 1024-bit ciphertext for each 32-bit integer.

5.2 Encryption Scheme Inference

MrCrypt is built as an extension to the Polyglot compiler framework [26]. Polyglot is designed to allow language extensions and analysis tools to be written on top of a base compiler for Java. MrCrypt is written in Scala and interfaces with Polyglot's intermediate representation of the Java bytecode. The tool takes as input a Java program and an encryption-scheme lattice and outputs a translated Java program which runs on the encrypted domain. It uses Polyglot compiler's dataflow framework and soundly handles imperative updates, aliasing, and arbitrary Java control flow in the standard way. We omit them from the formalism to isolate the key novelties of our approach.

We have extended our inference algorithm in several ways to handle Java programs that employ the Hadoop MapReduce framework; most of these extensions would also be useful in conjunction with other cloud computing frameworks. First, the user-defined map function in Hadoop is given a portion of a file representing the input data and must perform custom processing based on the file format to parse the data into columns. We require programmers to annotate the parsing code so MrCrypt can understand which variables get values from which columns, which are identified by number. For example, the user should annotate the following statement, which gets the fifth field in a line of input, with `@getColumn(5)`:

```
x = Library.splitLine(input, ' ').get(5);
```

Similarly, the statement that outputs the value of `x` as the sixth field in a record should be annotated with `@putColumn(6,x)`.

Second, we have extended the inference algorithm to handle common data structures. The map function returns a list of key-value pairs and the reduce function accepts a list of values as an argument. Further, programmers often use container data structures such as hashmaps and hashsets to remove duplicates, order elements, etc. Our implementation recognizes these data structures by type and encrypts their elements rather than the data structures themselves. In general our implementation uses a single logical variable for the purpose of encryption-scheme inference for a data structure's elements, ensuring that all elements are encrypted with the same scheme. However, we introduce two logical variables to handle lists of key-value pairs, so that keys and values can use different encryption schemes from one another. We also require data structures to be annotated with the operations they perform on their elements, in order to preserve these operations in the encrypted domain. Specifically, the standard Java hashmap and hashset classes are annotated to require the equality operation on elements.

Third, the shuffle phase of MapReduce sorts the intermediate keys produced by the map phase, thereby requiring support for order comparisons. However, in many cases the final output does not depend on the keys being sorted, instead just requiring that intermediate values be grouped by their key. Therefore, we allow programmers to annotate that sorting is not required for correctness of the program, allowing MrCrypt to choose deterministic encryption for the keys (which preserves equality, necessary for grouping values by key) rather than order-preserving encryption. The shuffle phase will be performed as usual by the Hadoop framework but will no longer guarantee that the underlying plaintext keys are in sorted order.

5.3 Optimizations

In order to scale to large datasets, we implemented a number of optimizations to the translator and runtime which can be categorized as follows:

Data serialization. Textual formats are very commonly used for MapReduce programs, with numbers represented as decimal strings. This encoding is highly inefficient for the mostly binary ciphertext data. Hence we use a binary serialization system, Avro[3], to store ciphertext.

Tuning Hadoop framework parameters. We tune Hadoop framework parameters such as the number of simultaneous map and reduce tasks, heap size, RAM used for shuffle phase, total number of reduce tasks, block size for the distributed file system, etc. based on the hardware on which they run. This is a manual process which depends on the program, data size as well as the cluster resources. These optimizations apply equally well to both the plaintext and ciphertext programs as they have similar data access patterns.

Efficient encoding of constants. We implemented a simple optimization for the case when the map function emits constant integer values. For example, in the standard MapReduce implementation of word count the map function emits the tuple $\langle w, 1\rangle$ for every word w in the input, and the reduce function sums up the numbers in the second component of each tuple. While this is efficient in the plaintext, the AH ciphertext for the number 1 in the translated program is 512 bits long. This causes significant slowdowns as the map's output is saved to the disk and read back and the entire data is kept in memory while sorting.

Our tool applies an optimization whenever either a constant integer value or a `final` variable initialized to an constant integer value is emitted by the map function. The optimization creates a dictionary in the translated program which associates symbols (represented by integers) with their ciphertexts. In the word count example, the plaintext map function contains `@putColumn(col0, word)` and `@putColumn(col1, 1)` for every word and the translated map function contains `@putColumn(col1, S)` and the reduce function has access to the dictionary which maps S to the AH ciphertext of 1.

6. Evaluation

This section describes the experimental evaluation of MrCrypt. We have applied encryption scheme inference to all programs in three MapReduce benchmark suites, in order to illustrate the applicability of the approach. We have also executed programs from one of the three suites on a cluster at scale to determine the run-time overhead of executing on encrypted data. Finally, we have used a set of microbenchmarks to isolate the client-side and server-side costs of encryption.

6.1 Benchmark Programs

The three benchmark suites are respectively listed in Tables 1, 2 and 3. For each benchmark, we list the number of source lines of code determined by the SLOCCount tool[4].

PIGMIX2[5] is a set of 17 benchmark programs written for the Pig framework, which provides a high-level language for writing large-scale data analysis programs called Pig Latin [27]. The framework compiles Pig Latin scripts into MapReduce programs and the runtime manages the evaluation of these programs. The PIGMIX2 benchmarks each come with Pig Latin scripts as well as hand-written MapReduce programs which the authors believe are efficient ways to execute the scripts. The programs run on a dataset primarily comprised of two tables: the PageViews table has 9 columns and the Users table has 6 columns.

Pavlo *et al.* [31] compare the performance of parallel databases that accept SQL queries with equivalent MapReduce programs. Their evaluation employs a standard wordsearch task [9] along with five other MapReduce benchmarks that perform various analytics queries[6], which we hereafter refer to as "the Brown suite."

The Purdue MapReduce Benchmarks (PUMA) Suite [1] contains 13 diverse mapreduce programs dealing with different computational and data patterns. In addition to performing encryption scheme inference, we also run these benchmarks on the large datasets that are provided with the bench-

[3] http://avro.apache.org

[4] http://www.dwheeler.com/sloccount/

[5] https://cwiki.apache.org/PIG/pigmix.html

[6] http://database.cs.brown.edu/projects/mapreduce-vs-dbms

marks: 50GB Wikipedia documents for the Word Count, Grep, Inverted Index, Term Vector, and Sequence Count benchmarks; 27GB movies data for the Histogram Movies and Histogram Ratings benchmarks; and upwards of 28GB of synthetic data for the rest of the benchmarks.

We made a few modifications to the benchmarks to work around current limitations of MrCrypt:

- The supported encryption schemes do not handle floating-point numbers, so we have converted all benchmarks that use floating-point numbers to instead use integers.
- Our implementation of OP supports comparisons for integers but not for strings, necessitating modifcations to three benchmarks. First, we modified the Aggregate Variant benchmark in the Brown suite to represent an IP address as four integers rather than a single string. Second, Self Join in the PUMA suite takes as input alphanumerically sorted text consisting of the string "entryNum" followed by 10 digits. We modified the input dataset to only include the numbers. Finally, Tera Sort in the PUMA suite sorts a column for which the input data consists of 10 random characters. We restrict the input data for this column to be populated by numeric characters.
- Three benchmarks in PIGMIX2 — L8, L15, and L17 — compute an average over some columns, which requires support for division. We modified these benchmarks to instead return a pair of the sum and the element count.

6.2 Experimental Setup

The experiments were run on the compute cluster at Max Planck Institute for Software Systems. The MapReduce computations were run on two Dell R910 rack servers each with 4 Intel Xeon X7550 2GHz processors, 64 x 16GB Quad Rank RDIMMs memory and 174GB storage. Our experiments ran on a total of 64 cores and had access to 1TB of RAM and 348GB of permanent storage. The machines were a shared resource and were under light load from other research projects. The Hadoop framework was configured to run 60 map and reduce tasks in parallel across the 64 available computational units.

In addition we used four Dell R910 rack servers (each with 2 Intel Xeon X5650 2.66GHz processors, 48GB RAM and 1TB hard disks) to host the distributed file system. No MapReduce computations were run on these machines and they were only used to serve input data and to store results. These machines were also a shared resource under regular load from other researchers.

6.3 Experimental Results

We are interested in three key metrics:

1. Annotation burden: How much extra work must the programmer do to make the existing MapReduce programs run securely?
2. Inference effectiveness: Does MrCrypt find the most efficient encryption scheme? How often is fully homomorphic encryption required?
3. Time and space overhead: How much runtime and storage cost does encrypted execution incur?

6.3.1 Annotation Burden

As mentioned in the implementation section, MrCrypt requires programmers to annotate parsing code to correlate variables with the input columns from which they are read. Our simple `getColumn` and `putColumn` annotations were sufficient to cover all of the file formats used in the benchmarks.

The encryption inference can otherwise be accomplished without any user annotations. However, as mentioned earlier, we allow users to annotate the fact that keys in a MapReduce program's output need not be in sorted order. We found that sorting is unnecessary in 29 of the 36 benchmarks because the specification does not require sorted output, so we included the associated annotation for these programs.

On average we added 12 annotations to each benchmark, which amounts to 7% of the lines of code.

6.3.2 Encryption Scheme Inference

Since *FH* is inefficient in practice, the utility of our tool depends on whether it is able to find efficient encryption schemes for real-world MapReduce programs. We present the results for the three benchmark suites in Tables 1, 2 and 3. For each benchmark, we measure the source lines of code by using the SLOCCount tool[7] along with the encryption schemes inferred for the input columns. For each encryption scheme the number of columns for which that scheme was inferred is mentioned in parenthesis. For each benchmark, the analysis time was less than 1 second, and the entire compilation time (including analysis and translation) was less than 5 seconds.

On 24 of 36 benchmarks, MrCrypt can identify encryption schemes to support the necessary functionality without requiring fully homomorphic encryption. Hence 66.7% of the programs can be executed securely through the system. We also manually analyzed each benchmark to verify the correctness of these results.

In the four cases of the PIGMIX2 suite where *FH* is required, the programs perform both equality and addition on the same column of data, for example to obtain a sum of all distinct values in the column. One of the benchmarks in the Brown suite (UDF) invokes performs string operations that MrCrypt does not support, one (Search) requires regular expression evaluation, and the other benchmark (Join) performs a sort on data obtained by computing a sum over some column. We are not aware of any homomorphic encryption scheme other than *FH* supporting both order comparisons and addition. In the PUMA suite, *FH* is required

[7] http://www.dwheeler.com/sloccount/

for regular-expression evaluation (Grep) and for computing cosine similarity (K-means and Classification).

Finally, MrCrypt determines that two benchmarks in the PUMA suite require FH for intermediate data produced by the map function. First, Term Vector counts all occurrences of words in documents and sort them by their frequency. This is implemented by using the map function to output $\langle$doc-name, word, 1$\rangle$ for every word, and the reduce to sum up all the 1s for each word in a document and then sort the words using the sums. Hence the numbers are both summed up and compared for order which results in FH for that variable. However, since we need to use DET to encrypt the input words to preserve equality, the number of occurrences of each (encrypted) word is already being leaked to the adversary. Hence leaving the integers in plaintext would not entail any extra loss of confidentiality, so in fact the benchmark can be executed securely without FH.

Second, Histogram Movies uses the map function to calculate the average rating of each movie rounded to the nearest 0.5. The reduce function then counts the number of movies with the same average rating. This functionality requires addition, division, and rounding operations and hence requires FH. However, we observe that we can refactor the benchmark into two different MapReduce programs to avoid FH. We refer to these two programs as Histogram Movies 1 and Histogram Movies 2, and they are also listed in Table 3. Histogram Movies 1 performs just the map phase of the original benchmark, with a trivial reduce, outputting the sum of all ratings of each movie along with their count. Histogram Movies 2 takes as input the average rating of each movie and performs just the reduce phase of the original benchmark, counting the number of movies with each average rating. MrCrypt infers encryption schemes for each of these benchmarks that allows them to execute securely without requiring FH.

To achieve the functionality of the original Histogram Movies benchmark, the client must decrypt the AH ciphertext output from Histogram Movies 1, re-encrypt it to use DET after computing the average and rounding it to the nearest 0.5 (and then doubling it to make it an integer), and provide the resulting ciphertext as input to Histogram Movies 2. While the client must perform some extra work, it does so on a small amount of data. On our input dataset, Histogram Movies 1 operates on 27GB of movie-rating data while Histogram Movies 2 only operates on 4MB of data that results from summing those ratings per movie (Table 4).

6.3.3 Time and Space Overhead

Our approach incurs two main sources of performance overhead, which we evaluate separately.

Client-side Overhead The client-side overhead consists of the need to encrypt the input data before sending it to the cloud and decrypt the output data from the computation. We evaluated this cost by measuring the time taken for encrypting and decrypting 500 random 32-bit integers. The OP, AH, and MH schemes take an average of 10ms, 4ms, and 1.5ms to encrypt each integer, respectively, and less than 0.5ms per decryption. Blowfish (the basis for $RAND$ and DET) has much less overhead of 200ns for each encryption and decryption operation. Thus, for example, encrypting one million data items with AH requires a bit more than one hour. However, in our target application domains the encryption can be performed incrementally as data is generated, and the encryption cost is amortized across multiple runs of the cloud computations.

Server-side Overhead The server-side overhead consists of the need to perform homomorphic operations on encrypted data rather than the original operations on the plaintext data. To isolate this overhead we developed a set of microbenchmarks, each of which performs a single operation one million times on the input data. For each operation we have one version of the microbenchmark that accepts plaintext integers and another version that uses the appropriate homomorphic encryption scheme to operate on ciphertext. We use a corpus of 10,000 32-bit integers and their corresponding ciphertexts as the input data. The performance overhead for encrypted execution is significant: slowdowns of 2× for DET, 4× for OP, 500× for AH, and 75× for MH.

Fortunately, the overheads on real MapReduce benchmarks are much lower, since the homomorphic operations contribute only a small percentage of the overall time. To evaluate the overhead of encryption on real-world data, we ran the PUMA benchmarks at scale on large data on a cluster. For each benchmark, we report the runtime for the original program, and the runtime for the transformed program. We also report the plaintext size and ciphertext size of the input data. We tabulate the results in Table 4.

The homomorphic operations add an insignificant overhead and the size of the ciphertext is the main factor in determining the runtime of the translated programs. On average the translated programs take 2.61× as long to execute as the original programs. However, Histogram Movies 1 is an outlier due to the need for AH, which uses 512 bits of ciphertext for each 32-bit integer, on a large amount of data. Excluding this benchmark the translated programs take an average of 1.57× as long to execute as the original programs.

In the three benchmarks where the program operating on ciphertext runs faster than the plaintext program (Adjacency List, Self Join, and Tera Sort), the speedup is due to using binary formats for encoding the encrypted numbers while the plaintext input uses a particularly inefficient textual format to encode numbers. In these benchmarks, numbers are padded with zeros to keep the length of each column the same so as to make use of the built-in sorting algorithm in the shuffle phase. Hence the number 1 would be represented as 0000000001. This approach uses 10 bytes to encode the range of 32-bit integers while the encrypted data uses at most 8 bytes to store the resulting 64-bit OP ciphertext. The bi-

Benchmark	Lines Of Code	Encryption Schemes Inferred for Inputs
L1	137	DET(2), RAND(7)
L2	148	DET(1), RAND(8)
L3	185	AH(1), DET(1), RAND(7)
L4	141	DET(2), RAND(7)
L5	169	DET(1), RAND(8)
L6	139	DET(3), FH(1), RAND(5)
L7	158	DET(1), OP(1), RAND(7)
L8	170	AH(2), RAND(7)
L9	196	OP(1), RAND(8)
L10	245	OP(3), RAND(6)
L11	184	DET(1), RAND(8)
L12	218	AH(1), DET(3), OP(1), RAND(4)
L13	182	DET(1), RAND(8)
L14	183	DET(1), RAND(8)
L15	188	DET(2), FH(2), RAND(5)
L16	134	DET(1), FH(1), RAND(7)
L17	259	FH(5), OP(20)

Table 1. Inference results on the PIGMIX2 benchmarks.

Benchmark	Lines of Code	Encryption Schemes Inferred for Inputs
Search	109	FH(1)
Select	71	OP(1), RAND(2)
Aggregate	99	AH(1), DET(1), RAND(7)
Aggregate Variant	162	AH(1), DET(3), RAND(7)
Join	518	AH(1), DET(2), FH(1), OP(1), RAND(4)
UDF	58	AH(1), DET(1), FH(1), RAND(6)

Table 2. Inference results on benchmarks from the Brown suite.

nary format uses variable-length encoding and hence might use fewer than 8 bytes in some cases.

6.4 Discussion

Space Efficiency. Encryption schemes like AH require a significant blowup in space, which has a direct impact on execution time as well. We can reduce the overhead for Pallier encryption (our implementation of AH) using a packing optimization [13].

Avoiding Fully Homomorphic Encryption. We showed earlier how refactoring the Histogram Movies benchmark can make it amenable to our approach, and we believe there are additional opportunities along these lines. For example, four benchmarks that currently require FH require a "sum of distinct elements" functionality, which typically looks as follows:

```
Integer f(List<Integer> revenues) {
  HashSet<Integer> hs = new HashSet<
      Integer>();
  for (Integer r: revenues) hs.add(r);
  int sum = 0;
  for(Integer r: hs) sum += r.intValue();
  return new Integer(sum);
}
```

The revenues column has two operations performed on it: equality (from the hashset) and addition. Hence our tool infers FH in this case. However, this program can be run securely by keeping two copies of the revenues column, one for equality and the other for addition, and keeping a correspondence between them (we use the class P2 for pairs, along with associated operations, from the Java library fj[8]):

```
Integer f(List<Integer> erevenues,
          List<Integer> arevenues) {
  HashSet<Integer> hs = new HashSet<
      Integer>();
  List<Integer> distincts = list();
  for (P2<Integer, Integer> p:
        erevenues.zip(arevenues)) {
    if (!hs.contains(p._1())) {
      hs.add(p._1());
      distincts.cons(p._2());
    }
  }
  int sum = 0;
```

[8] http://functionaljava.org/

Benchmark	Lines Of Code	Encryption Schemes Inferred for Inputs
Word Count	88	DET(1)
Inverted Index	126	DET(1)
Term Vector*	187	DET(1)
Self Join	136	OP(1)
Adjacency List	157	OP(2)
K-Means	428	DET(1), FH(1), OP(1)
Classification	228	DET(1), FH(1), OP(1)
Histogram Movies*	132	AH(1), RAND(2)
Histogram Movies 1	113	AH(1), RAND(2)
Histogram Movies 2	98	AH(1), DET(1)
Histogram Ratings	115	DET(1), RAND(2)
Sequence Count	124	DET(1)
Ranked Inverted Index	127	DET(4), OP(1)
Tera Sort	192	OP(1), RAND(1)
Grep	55	FH(1)

Table 3. Inference results on the PUMA benchmark suite. An asterisk denotes that *FH* was inferred for an intermediate variable in the benchmark. The Histogram Movies 1 and Histogram Movies 2 benchmarks were created by us and are discussed in Section 6.3.2.

Benchmark	Original Program Runtime (sec)	Transformed Program Runtime (sec)	Plaintext Size (GB)	Ciphertext Size (GB)
Word Count	528	1064	50	79
Inverted Index	395	658	50	79
Term Vector	556	1114	50	79
Self Join	252	234	28	26.1
Adjacency List	823	769	28	26.5
Histogram Movies 1	138	1801	27	388
Histogram Movies 2	22	32	0.004	0.067
Histogram Ratings	214	427	27	36
Sequence Count	492	1006	50	79
Ranked Inverted Index	305	525	37.8	60.3
Tera Sort	1080	1062	28	26.9

Table 4. Performance results on the PUMA benchmark suite

```
  for(Integer r: distincts)
      sum += r.intValue();
  return new Integer(sum);
}
```

It would be interesting to explore performing such preprocessing automatically in order to extend the applicability of our approach.

7. Related Work

Computing over encrypted data. The problem of (fully) homomorphic encryption was posed by Rivest, Adleman, and Dertouzos [34], and the first fully homomorphic scheme was discovered by Gentry [14]. Implementations of Gentry's construction remains prohibitively expensive [16]. A more efficient encryption scheme [25] can perform unbounded additions but only a bounded number of multiplications. Cryptographically secure multi-party computations are also theoretically possible for general circuit evaluation [37, 43]. Homomorphic encryption schemes have been proposed to protect data security in several applications including secure financial transactions [4], secure voting [19], and sensor networks [7].

As discussed in Section 1, the work closest to our own is the CryptDB project [32], which uses homomorphic encryption to run queries securely on relational databases. CryptDB encrypts the data in all possible encryption schemes, layered on top of each other in a structure resembling our lattice. A trusted proxy stands between clients and the database system, analyzes the SQL queries on the fly, and decrypts the relevant columns to the right encryption layers so that the query can be executed. The key difference between these two efforts is that MrCrypt performs static analysis of imperative Java programs while CryptDB performs analysis on database queries and so is limited to computations that are expressible in pure SQL (i.e., no user-defined functions). Further, because MrCrypt has up-front access to the programs, it can statically determine the best encryption schemes to use, avoiding the need to encrypt data with multiple schemes and to employ a trusted proxy. However, encrypting data with multiple schemes allows some queries to be executed using CryptDB that cannot be handled by our system. Finally, we have formalized our approach and proven its correctness and security guarantees, while CryptDB provides only informal guarantees.

Other work in the database community has used homomorphic encryption for particular kinds of queries. For example, SADS [33] allows encrypted text search and other work uses additive homomorphic schemes to support sum and average queries [13]. These systems do not support general imperative computations.

Cryptographic schemes have been used to provide privacy and integrity in systems running on untrusted servers [22, 23]. However, these systems have so far required application logic to be executed purely on the client. Our goal, on the other hand, is to enable computations to run directly on untrusted servers. It may be possible to incorporate ideas from these systems in order to augment our approach to guarantee integrity in addition to confidentiality.

Mitchell *et al.* formalize a domain-specific language (DSL) whose type system ensures that programs can be translated to run securely using either FH or secure multiparty computation [24]. They also describe an implementation of their DSL embedded in Haskell. This approach can potentially be more expressive than ours but requires programmers to write programs in a specialized language, while MrCrypt handles existing Java programs with minimal code annotations. Finally, Mitchell *et al.* do not consider the use of partially homomorphic encryption schemes.

Static and dynamic analysis for security. There is a large body of work on static and dynamic techniques for enforcing security policies or for finding security vulnerabilities. Most language-based approaches to enforcing confidentiality are based on the notion of *secure information flow* [36]. These approaches are less applicable to the setting of cloud computing, where the adversary can have direct access to the machine on which a computation is being performed. For example, a common threat model in the context of secure information flow assumes the adversary has access only to the public inputs and outputs of a computation. Researchers have augmented traditional information-flow type systems to reason about confidentiality in the presence of cryptographic operations [12, 41], but these approaches require programmers to manually employ cryptography in their programs.

MrCrypt also leverages static analysis techniques, but for a different purpose — to identify the most efficient encryption schemes to use for each input column of data. As described in our formalism, this analysis is similar to techniques for flow-insensitive type qualifier inference [11, 28].

Computing in untrusted environments. The Excalibur system [38] uses trusted platform modules (TPMs) to guarantee that privileged cloud administrators cannot inspect or tamper with the contents of a VM. While this approach provides the same security guarantees as MrCrypt, it requires additional investment from the cloud companies to install special TPM chips on each node in the cloud and for managing keys. CLAMP [30] prevents web servers from leaking sensitive user data by isolating code running on behalf of one user from that of other users. However, CLAMP does not protect user confidentiality against honest-but-curious cloud administrators. Finally, work on differential privacy for MapReduce (e.g., [35]) is dual to our concern: in that setting the server is trusted but information exposed to clients is minimized.

8. Conclusion

Data confidentiality is a key challenge for shared computing infrastructures such as cloud computing. We have presented MrCrypt, a practical solution to ensure confidentiality through the use of homomorphic encryption.

MrCrypt performs a static analysis on Java programs to identify the most efficient homomorphic encryption scheme supporting the necessary operations on each column of input data, and it then automatically rewrites the program to execute on encrypted data. We have formalized the approach and proven strong correctness and security guarantees. Our experimental results on three Hadoop MapReduce benchmark suites indicate that fully homomorphic encryption is unnecessary most of the time, and as a result a rewritten program provides strong confidentiality guarantees while incurring only a modest execution-time slowdown.

Acknowledgments. We thank Christian Mickler for help with the MPI-SWS clusters. This work is supported in part by the National Science Foundation under awards CCF-1048826 and CNS-1064997.

References

[1] Faraz Ahmad, Seyong Lee, Mithuna Thottethodi, and TN Vijaykumar. Puma: Purdue mapreduce benchmarks suite. 2012.

[2] O. Baudron, D. Pointcheval, and J. Stern. Extended notions of security for multicast public key cryptosystems. In *ICALP '00*, volume 1853 of *Lecture Notes in Computer Science*, pages 499–511. Springer, 2000.

[3] M. Bellare, T. Kohno, and C. Namprempre. Authenticated encryption in ssh: provably fixing the ssh binary packet protocol. In *CCS '02*, pages 1–11. ACM, 2002.

[4] M. Bellare, T. Ristenpart, P. Rogaway, and T. Stegers. Format-preserving encryption. In *Selected Areas in Cryptography*, volume 5867 of *Lecture Notes in Computer Science*, pages 295–312. Springer, 2009.

[5] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. In *EUROCRYPT*, volume

5479 of *Lecture Notes in Computer Science*, pages 224–241. Springer, 2009.

[6] A. Boldyreva, N. Chenette, and A. O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *CRYPTO*, volume 6841 of *Lecture Notes in Computer Science*, pages 578–595. Springer, 2011.

[7] C. Castelluccia, E. Mykletun, and G. Tsudik. Efficient aggregation of encrypted data in wireless sensor networks. In *Proceedings of the The Second Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services*, MOBIQUITOUS '05, pages 109–117, Washington, DC, USA, 2005. IEEE Computer Society.

[8] J. Daemen and V. Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer, 2002.

[9] J. Dean and S. Ghemawat. MapReduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[10] T. ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[11] J.S. Foster, R. Johnson, J. Kodumal, and A. Aiken. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst.*, 28(6):1035–1087, November 2006.

[12] Cédric Fournet, Jérémy Planul, and Tamara Rezk. Information-flow types for homomorphic encryptions. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 351–360. ACM, 2011.

[13] T. Ge and S. Zdonik. Answering aggregation queries in a secure system model. In *Proceedings of the 33rd international conference on Very large data bases*, pages 519–530. VLDB Endowment, 2007.

[14] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC 09: Symposium on Theory of Computing*. ACM, 2009.

[15] C. Gentry. Computing arbitrary functions of encrypted data. *Commun. ACM*, 53(3):97–105, 2010.

[16] C. Gentry and S. Halevi. Implementing Gentry's fully-homomorphic encryption scheme. In *EUROCRYPT 11*, volume 6632 of *Lecture Notes in Computer Science*, pages 129–148. Springer, 2011.

[17] S. Goldwasser and S. Micali. Probabilistic encryption. *J. Computer and Systems Sciences*, 28:270–299, 1984.

[18] S. Halevi and P. Rogaway. A tweakable enciphering mode. *Advances in Cryptology-CRYPTO 2003*, pages 482–499, 2003.

[19] M. Hirt and K. Sako. Efficient receipt-free voting based on homomorphic encryption. In *Proceedings of the 19th international conference on Theory and application of cryptographic techniques*, EUROCRYPT'00, pages 539–556, Berlin, Heidelberg, 2000. Springer-Verlag.

[20] E. Kowalski. Insider threat study: Illicit cyber activity in the information technology and telecommunications sector. Technical report, Technical report, U.S. Secret Service and CMU, 2008.

[21] M. Lesani, R. Majumdar, T. Millstein, and S. Tetali. MrCrypt: Static analysis for secure cloud computations (technical report). Technical Report 130012, UCLA Computer Science Department, July 2013.

[22] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (sundr). In *OSDI 04: Operating Systems Design and Implementation*, pages 91–106. ACM, 2004.

[23] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *OSDI 10: Operating Systems Design and Implementation*. ACM, 2010.

[24] J.C. 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. IEEE, 2012.

[25] M. Naehrig, K. Lauter, and V. Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, CCSW '11, pages 113–124, New York, NY, USA, 2011. ACM.

[26] N. Nystrom, M. Clarkson, and A. Myers. Polyglot: An extensible compiler framework for java. In *Compiler Construction*, pages 138–152. Springer, 2003.

[27] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[28] Peter Ørbæk and Jens Palsberg. Trust in the λ-calculus. *Journal of Functional Programming*, 7(6):557–591, November 1997.

[29] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT 99: Theory and Applications of Cryptographic Techniques*, 1999.

[30] 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, Washington, DC, USA, 2009. IEEE Computer Society.

[31] A. Pavlo, E. Paulson, A. Rasin, D.J. Abadi, D.J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 35th SIGMOD international conference on Management of data*, pages 165–178. ACM, 2009.

[32] R.A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100. ACM, 2011.

[33] M. Raykova, B. Vo, S.M. Bellovin, and T. Malkin. Secure anonymous database search. In *CCSW 09: Cloud Computing Security Workshop*, pages 115–126. ACM, 2009.

[34] R. Rivest, L. Adleman, and M.L. Dertouzos. On data banks and privacy homomorphisms. In *Foundations of Secure Computation*, pages 169–179. Academic Press, 1978.

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

[36] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[37] T. Sander, A. Young, and M. Yung. Non-interactive cryptocomputing for NC^1. In *FOCS 99: Foundations of Computer Science*. IEEE, 1999.

[38] N. Santos, R. Rodrigues, K.P. Gummadi, and S. Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *Usenix Security Symposium*. USENIX Association, 2012.

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

[40] B. Schneier. *Applied cryptography*. Wiley, 2nd edition, 1996.

[41] J.A. Vaughan. Auraconf: a unified approach to authorization and confidentiality. In *Proceedings of the 7th ACM SIGPLAN workshop on Types in language design and implementation*, TLDI '11, pages 45–58, New York, NY, USA, 2011. ACM.

[42] A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[43] A. Yao. How to generate and exchange secrets. In *FOCS 86: Foundations of Computer Science*, pages 162–167. IEEE, 1986.