Secure Data Types: A Simple Abstraction for Confidentiality-Preserving Data Analytics[*]

Savvas Savvides$^{\alpha}$ Julian James Stephen$^{\alpha}$ Masoud Saeida Ardekani$^{\alpha\dagger}$
Vinaitheerthan Sundaram$^{\alpha\beta}$ Patrick Eugster$^{\alpha\beta\gamma}$
$^{\alpha}$Purdue University, $^{\beta}$SensorHound Inc, $^{\gamma}$TU Darmstadt

ABSTRACT

Cloud computing offers a cost-efficient data analytics platform. However, due to the sensitive nature of data, many organizations are reluctant to analyze their data in public clouds. Both software-based and hardware-based solutions have been proposed to address the stalemate, yet all have substantial limitations. We observe that a main issue cutting across all solutions is that they attempt to support confidentiality in data queries in a way transparent to queries. We propose the novel abstraction of *secure data types* with corresponding annotations for programmers to conveniently denote constraints relevant to security. These abstractions are leveraged by novel compilation techniques in our system Cuttlefish to compute data analytics queries in public cloud infrastructures while keeping sensitive data confidential. Cuttlefish encrypts all sensitive data residing in the cloud and employs partially homomorphic encryption schemes to perform operations securely, resorting however to client-side completion, re-encryption, or secure hardware-based re-encryption based on Intel's SGX when available based on a novel planner engine. Our evaluation shows that our prototype can execute *all* queries in standard benchmarks such as TPC-H and TPC-DS with an average overhead of 2.34× and 1.69× respectively compared to a plaintext execution that reveals all data.

CCS CONCEPTS

• **Security and privacy → Management and querying of encrypted data**; *Distributed systems security*;

KEYWORDS

Cloud computing, Data confidentiality, Homomorphic encryption

[*]Work supported in parts by Northrop Grumman, Cisco, NSF #1117065, #1421910, #1618923, and ERC #617805, and by AWS Cloud Credits for Research.
[†] Work done at Purdue University. Now at Samsung Research America.

SoCC '17, September 24–27, 2017, Santa Clara, CA, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5028-0/17/09…$15.00
https://doi.org/10.1145/3127479.3129256

1 INTRODUCTION

Over the past few years, compute clouds have emerged as cost-efficient data analytics platforms for corporations and governments. Yet, many organizations still decline to widely adopt cloud services due to severe *confidentiality* and *privacy* concerns (cf. [9, 45]); and corresponding explicit regulations in certain sectors (e.g., healthcare and finance [7]).

1.1 Background & Challenges

To address the above issues, techniques have emerged that allow queries to be run against data kept in clouds in an encrypted manner. Two prominent approaches consist in (1) software-only solutions based on homomorphic encryption (HE), and (2) hardware-based trusted computing solutions such as Intel's Software Guard Extensions (SGX).

With the first approach, data is encrypted with specific crypto systems which allow computations to be performed directly on corresponding ciphertexts. Over the past few years, many researchers have vigorously investigated *fully* homomorphic encryption (FHE) schemes [28]. These support arbitrary computations on ciphertexts, but incur around $10^9\times$ slowdowns [29] compared to executing computations on plaintext, thus annulling the benefits of computing in a cloud. As a consequence they have not gained a great momentum in practice yet.

In response, several systems (e.g., [31, 42, 44, 46, 50, 52, 53]) have been proposed based on *partially* homomorphic encryption (PHE). These systems use different encryption schemes based on operations performed/supported, which leads to straightforward limitations. For instance, to compute $(x+y)\times z$ on ciphertexts, x and y need to be encrypted under additive homomorphic encryption; this would carry over to the result of their addition, rendering a subsequent multiplication by z impossible as that requires both its operands to be available under multiplicative homomorphic encryption. Existing PHE-based systems handle such limitations in one of the following three ways: (i) *abort* by rejecting a corresponding query (e.g., [52]); (ii) *split execution* by returning the computation back to the client (or trusted servers considered as part of the client's domain) with intermediate encrypted data, and completing the query there (e.g., [31, 42, 44, 46, 53]); (iii) *re-encryption* by sending the encrypted intermediate data to trusted servers, re-encrypting it under the desired scheme, and sending it back to the cloud to continue the query (e.g., [49, 51]). These approaches either limit expressiveness – the extent to which a query can be executed on encrypted data in the cloud – thus hampering performance, or make up for expressiveness by similarly introducing overheads.

Hardware approaches (2) have been recently hailed as ushering in a new era, with SGX garnering most attention. SGX focuses on providing isolated areas of execution called *enclaves*, where data can be decrypted and computed over. While yielding a powerful mechanism, SGX has limitations of its own, in particular making it hard for enclaves to be used as simple drop-in replacements for other techniques. For instance, enclaves can only (efficiently) access a limited amount of memory [10, 17]. Intel suggests to use enclaves only for sensitive code/data to minimize the trusted code base and its interfaces, in order to limit potential for bugs and thus exploits as these enclaves become the trust anchors of larger programs [5, 6]. Consequently, programs typically use SGX for code that omits advanced features such as automatic memory reclamation or dynamic code loading. Without dynamic code loading, enclave codes need to be implemented as "small interpreters" in order to be able to service other than predefined queries. Additionally, enclaves are non-trivial to set up, their initialization itself adds runtime overhead and involves attestation through Intel.

We observe that one major impediment to satisfactory performance and expressivity in data analytics across all approaches, software- or hardware-based, is that of *transparency*: In the case of FHE, the goal is for arbitrary programs to be executable on ciphertexts at the flip of a switch. Similarly, in the case of PHE, existing systems promote unmodified query languages and computational models (e.g., SQL [31, 44, 46, 53], MapReduce [52], PigLatin [49], Spark [56]), and their runtime systems execute queries pretty much line-by-line and treat individual encryption schemes mostly as black boxes; this not only leads to suboptimal performance but also confounds security guarantees of different crypto systems rather than reflecting them to programmers. Finally, hardware-based approaches such as SGX promise to execute arbitrary programs in isolated fashion, yet effectively mapping (parts of) a data analytics query to something like enclaves across many compute nodes is non-trivial.

1.2 Contributions

In this paper we thus propose to abandon transparency, and instead introduce specifically tailored yet intuitive programming abstractions that can capture relevant constraints on computations with respect to confidentiality. In turn, this allows data analytics queries to be mapped effectively to both software- and hardware-based approaches for secure cloud-based execution. We introduce Cuttlefish, a system able to perform analytical queries over encrypted data in public clouds leveraging both PHE and SGX in a concerted fashion.

Cuttlefish's compiler uses a wealth of novel techniques leveraging our abstractions to significantly improve performance; its planner engine demonstrates that PHE and SGX are not necessarily competing but rather can complement each other. While placing more emphasis on PHE for portability, Cuttlefish can leverage SGX when present to improve system performance, while keeping the corresponding code base very small and easily verifiable. To the best of our knowledge, Cuttlefish is the first system that demonstrates how PHE can be combined with hardware-based isolation to improve performance and expressiveness.

System	Expressivity	Data constraints	Specialized hardware
FHE [29]	unlimited[1]	no	n/a
Opaque [57]	unlimited[2]	no	required
CryptDB [44]	limited[3]	no	n/a
MrCrypt [52]	limited[3]	no	n/a
Monomi [53]	limited[4]	no	n/a
Seabed [42]	limited[4]	no	n/a
Crypsis [50]	limited[5]	no	n/a
Cuttlefish	unlimited[6]	yes	optional

Table 1: Comparison of Cuttlefish to state-of-the-art systems. [1]By definition of FHE. [2]In-enclave computations. [3]Abort on PHE limitations. [4]Split execution. [5]Re-encryption. [6]Planner engine to overcome PHE limitations without compromising performance

Table 1 shows how Cuttlefish compares with various state-of-the-art systems. Concretely, Cuttlefish makes the following contributions:

Programming abstractions: Cuttlefish introduces *secure (abstract) data types* (SDTs) to express data flow style data computations. SDTs are based on well-known data types but capture simple but fine-grained information relevant to security and encryption such as sensitivity levels and precise ranges of values.

Compiler: Our compiler leverages the above data types to substantially accelerate PHE-based query execution. For instance, the Cuttlefish compiler incorporates a set of compilation techniques aiming to reduce the times and extent of client (or trusted) platform involvement, for instance by reducing the number of required re-encryptions. In addition, it exploits intrinsic properties of crypto systems such as secondary homomorphic properties. Several of our query rewriting techniques go against traditional techniques in order to address the specific constraints of PHE.

Planner engine: Based on the observation that split execution is a special case of re-encryption at a trusted tier (e.g., clients), our planner engine automatically decides when to complete queries at the trusted tier, or when to return to the cloud after re-encryption, to achieve the best performance. In the latter case, and in the presence of cloud hosts supporting SGX, our planner engine leverages these areas to mitigate PHE limitations.

We built a prototype of Cuttlefish that runs on Apache Spark [56] and evaluated it using standard benchmarks. Our results show that in contrast to previous approaches, Cuttlefish can execute *all* queries of TPC-H and TPC-DS with an average overhead of 2.34× and 1.69× respectively compared to a plaintext execution. For TPC-H, Cuttlefish improves the average overhead by 3.35× and 3.71× over state-of-the-art systems Monomi [53] and Crypsis [50].

The rest of this paper is structured as follows. Section 2 provides an overview of Cuttlefish and background information. Section 3 introduces SDTs. Section 4 presents our novel compilation techniques. Section 5 describes our planner engine and the heuristics for re-encryption. Section 6 discusses the implementation of Cuttlefish. Section 7 empirically evaluates Cuttlefish. Section 8 contrasts Cuttlefish with related work. Section 9 concludes with final remarks.

2 OVERVIEW

In this section, we first explain the threat model of our system as well as provide some background on cryptographic primitives before we present our system design.

2.1 Threat Model

Cuttlefish's goal is to preserve data confidentiality in the presence of an honest but curious (HbC) adversary. We assume the adversary has access to the cloud nodes, and can observe data and computation. Although, the adversary cannot make changes in the queries, results or data stored in the cloud. We further assume that the system has access to a trusted service. This service can run either on some trusted client nodes, or on some specialized trusted hardware in the cloud such as Intel SGX [3].

Cuttlefish achieves this goal by utilizing a set of crypto systems to encrypt sensitive data. The security guarantees offered by each of these crypto systems vary from strong guarantees offered by probabilistic crypto systems, to relatively weaker guarantees such as deterministic crypto systems which reveal duplicate values, and order-preserving crypto systems which reveal order of values.

Despite improvements [15, 16, 19, 34, 36, 43], OPE and DET crypto systems remain susceptible to inference attacks such as frequency analysis attacks [30, 38]. As we discuss in subsequent sections, Cuttlefish deals with this issue by allowing the programmer to assign sensitivity levels to individual data fields and guaranteeing that a field is never stored in the cloud unless encrypted under a crypto system that offers the required security guarantees. Furthermore, in Section 4.6 we explain how Cuttlefish can apply a set of compilation techniques that can further improve the security of queries by limiting the use of less secure crypto systems.

2.2 Cryptographic Primitives

Each of the crypto systems used by Cuttlefish to achieve confidentiality inside an untrusted cloud allows computations over encrypted data with respect to some operations. Table 2 summarizes the set of crypto systems used by Cuttlefish per homomorphic property — RND: random, DET: deterministic, OPE: order-preserving encryption, SRCH: secure search, AHE: additive homomorphic encryption, MHE: multiplicative homomorphic encryption — and the corresponding operations they support. The operation supported by each crypto system requires that its operands are encrypted under the same crypto system.

In addition to this operation, some crypto systems support a second operation (secondary homomorphic property) as long as one of the operands is available in plaintext form (i.e., it holds no sensitive information). For example, the Paillier crypto system is an AHE crypto system which means its homomorphic property supports addition between two encrypted values: there exists some known operation $\odot$ s.t. $x + y = Dec(Enc(x) \odot Enc(y))$. Furthermore, if one of the two operands, say y, is in plaintext form, Paillier can also perform multiplication between the two operands: there exists some known operation $\otimes$ s.t. $x \times y = Dec(Enc(x) \otimes y)$. Similarly, the ElGamal crypto system supports multiplication between two encrypted operands and exponentiation between an encrypted and a plaintext operand. In Section 4.4 we discuss how intrinsic properties of crypto systems (such as ciphertext size overhead and secondary homomorphic property) that otherwise support the same operations, e.g., Paillier and ASHE are leveraged to reduce query execution time.

Crypto system	Property	Operations
AES-RND	RND	–
FNR [21], AES-DET	DET	=, `GROUP`, `JOIN`
Boldyreva et al. [14]	OPE	<, >, `ORDER`, `MIN`
SWP [48]	SRCH	`MATCHES` *pattern*
Paillier [41], ASHE [42]	AHE	+, −, `SUM`
ElGamal [25]	MHE	×

Table 2: Cuttlefish crypto systems and operations over encrypted data they support

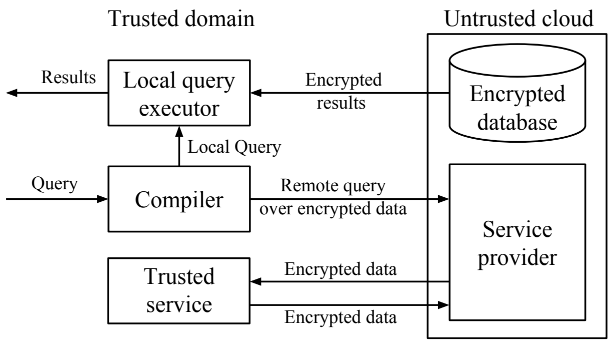

Figure 1: Cuttlefish architecture

2.3 Cuttlefish Overview

Figure 1 shows the high level architecture of Cuttlefish. Cuttlefish ensures the confidentiality of computations of submitted queries by transforming them into semantically equivalent queries that operate over encrypted data. When a user submits a query, the Cuttlefish compiler transforms it into a remote query and a local query. The remote query which operates on encrypted data is deployed on an untrusted cloud. Once the remote query completes, the encrypted results are returned to the local query executor and used as the input for the local query which decrypts the results of the remote query and performs any remaining computations on plaintext data before returning the final results to the user.

Since PHE schemes allow computations with respect to certain operations, it is possible that some parts of the query cannot be executed in the cloud without giving away sensitive information. To mitigate this limitation, Cuttlefish utilizes a trusted re-encryption service that has access to the decryption keys. The trusted re-encryption service would receive a small amount of data to decrypt, optionally perform simple computations over the data, encrypt the result under another crypto system and send the results back to the cloud service, so that computation can proceed. Cuttlefish has two ways of realizing the trusted re-encryption service: using hardware at the client-side or using trusted hardware (e.g., Intel SGX) in the cloud.

Compilation technique	G1	G2	G3	G4
Expression rewriting	✓	✓	✓	
Condition expansion		✓	✓	
Selective encryption		✓		✓
Efficient encryption		✓	✓	✓

Table 3: Cuttlefish compilation techniques and high level goals they achieve

In order to improve expressiveness and general performance as well as to reduce the amount and extent of re-encryption — applied naïvely re-encryption can induce high overheads — Cuttlefish allows the programmer to specify fine-grained information about the involved data through a novel abstraction (SDTs). Cuttlefish compiler then uses this information to apply a set of *compilation techniques.* Table 3 summarizes how these techniques contribute to our high level goals, namely by (G1) enabling queries not previously executable in a public cloud without sacrificing confidentiality, (G2) reducing the involvement of the trusted service, (G3) reducing the amount of computation, and (G4) reducing the amount of encrypted data. Below, we give an intuition of the categories of techniques, and in later sections describe how they are used in Cuttlefish.

Expression rewriting. Cuttlefish replaces expressions that are not supported by any of Cuttlefish's crypto systems with equivalent expressions that can be performed in the cloud over encrypted data. Cuttlefish also rewrites expressions in queries in encryption-sensitive ways that reduce execution latency. For example, $(x + y) \times z$ can be rewritten to $x \times z + y \times z$ or vice versa, depending on the crypto systems x, y and z are encrypted under and on subsequent operations. In addition, since Cuttlefish executes expressions over encrypted data, the cost of recomputing common subexpressions is compounded. We eliminate such subexpressions in our compilation phase.

Condition expansion. Similarly, Cuttlefish expands conditions to improve execution performance. For example, if it is known that variables x and y are non-negative integers, we can safely rewrite $x+y > 0$ to $x > 0 \,||\, y > 0$. The changed condition has the potential to eliminate expensive additive homomorphic encryption for x and y and the need for re-encryption.

Selective encryption. Cuttlefish allows fields that do not contain sensitive information to exist in plaintext in the cloud, thus supporting more homomorphic operations and keeping the data size overhead small.

Efficient encryption. Cuttlefish reduces the amount and size of encrypted data by identifying situations where one field is involved in multiple operations that can be supported by the same crypto system. Inversely, when the same operation can be performed by multiple crypto systems Cuttlefish selects the crypto system leading to more efficient execution.

3 SECURE DATA TYPES

Cuttlefish utilizes a set of compilation techniques to improve the expressiveness and performance of queries executed on encrypted data. A cornerstone of these techniques consists in Cuttlefish's ability to define input data in ways not typically allowed by query languages. In particular, Cuttlefish allows the definition of: (i) sensitivity levels, (ii) ranges and decimal accuracy for data values, (iii) uniqueness and tokenization, (iv) enumerations and (v) composition of data types.

These are captured by our secure data types (SDTs) and leveraged by the Cuttlefish compiler to substantially optimize analytical queries. Table 4 cross-references our SDTs with the compilation techniques that leverage them.

Secure data type	T1	T2	T3	T4
Sensitivity level	✓		✓	✓
Data range and decimal accuracy	✓	✓		
Uniqueness and tokenization	✓			✓
Enumerated type				✓
Composite type	✓			

T1: Expression rewriting
T2: Condition expansion
T3: Selective encryption
T4: Efficient encryption

Table 4: Secure data types and corresponding compilation techniques they enable

Sensitivity level	Encryption scheme
HIGH	RND, AHE, MHE, SRCH, DET^U
LOW	DET, OPE
NONE	plaintext

Table 5: Sensitivity levels and associated encryption schemes. DET^U denotes deterministic encryption on fields with unique values

3.1 Sensitivity Level

As previously mentioned, crypto systems used by Cuttlefish offer different levels of security. Cuttlefish captures this difference by categorizing crypto systems under the categories HIGH and LOW (and NONE) as shown in Table 5. Category HIGH captures crypto systems that are probabilistic or have no leakage except for the length of the plaintext. We also allow deterministic schemes to be listed under this category, but only in situations where the values of a field to be encrypted are unique (DET^U) and hence determinism does not reveal duplicate entries. Category LOW captures encryption schemes such as DET and OPE which reveal duplicates or order. Correspondingly, Cuttlefish allows each field of a table (and each constant in queries) to be optionally annotated with the sensitivity level HIGH, LOW or NONE.

The HIGH level of sensitivity is intended for fields that hold highly sensitive values. These fields are only allowed to exist in the untrusted cloud encrypted under a crypto system that belongs in category HIGH. The LOW sensitivity level is intended for fields that are less sensitive such as fields with high entropy like timestamps.

SDT annotation	Description
`+ | -`	Positive or negative numeric values
`range(numA-numB)`	Values from `numA` to `numB`
`accuracy(num)`	`num` decimal point(s) preserved
`unique`	No duplicate values
`delimiter(char)`	Tokens separated by `char`
`enum{value1, value2, ...}`	Enumerated values
`composite`	Composite values

Table 6: SDTs annotations

These fields must be encrypted under crypto systems of at least `LOW` security. Finally, a field can be annotated with `NONE` to indicate that it is a non-sensitive field (e.g., publicly available information) and can hence reside in the cloud in its plaintext form. This distinction of sensitivity level allows Cuttlefish to be more expressive and achieve better performance without compromising security, by enabling the selective encryption and efficient encryption compilation techniques described in Section 4.

We note that the programmer does *not* have to specify a sensitivity level for all fields. If no sensitivity level is explicitly assigned to a field, the default behavior is to use an appropriate crypto system that supports the operation the field is involved in (i.e., no lower than `LOW`). In practice, we found that it is sufficient to specify: (i) the fields with high sensitivity so that Cuttlefish never allows those fields to exist under a crypto system that does not offer high security guarantees, and (ii) the fields that do not hold sensitive information, allowing Cuttlefish to leave those fields in plaintext which can lead to generating queries with improved expressiveness and performance.

3.2 Data Range and Decimal Accuracy

Apart from the sensitivity level, programmers can *optionally* provide information about data ranges and decimal accuracy of types, helping Cuttlefish generate more expressive and efficient queries, by simplifying and rewriting expressions and conditions. Table 6 summarizes this information.

For numerical types, the programmer can specify the sign of the values of a field. For example, `x:int[+]` specifies that field `x` holds only positive integers and similarly `x:int[-]` specifies that field `x` holds only negative integers. For more constrained numerical values the programmer can specify the `range` of values. For instance, `x:int[range(100-200)]` indicates that field `x` holds integer values within the range `100` to `200`.

For floating point numbers, the programmer can specify the number of decimal points that need to be preserved when encrypting values. For example, `x:double[accuracy(2)]` indicates that `2` decimal points need to be preserved. If the number of decimal points to preserve is not specified, Cuttlefish truncates all decimal points of the floating number making it a whole number. This is necessary since the crypto systems used by Cuttlefish for arithmetic operations do not support floating point numbers by default. Cuttlefish uses the decimal accuracy information to multiply the values of the field appropriately, truncating the remaining decimal points, before encrypting.

3.3 Uniqueness and Tokenization

Apart from numerical types, the programmer can declare any data type as being `unique` to indicate that the field does not contain duplicates, e.g., `ssn:string[unique]` specifies that `ssn` field holds unique strings, which then allows Cuttlefish to encrypt that field under a DET crypto system while preserving high security guarantees. In addition, for fields that hold string values the programmer can specify a `delimiter` that separates words. This is particularly important when encrypting under a SRCH crypto system such as SWP, since the latter requires to tokenize the string into individual words before encrypting.

3.4 Enumerated Types

Fields containing a small and fixed set of values can be represented as an enumerated type in Cuttlefish. This is done by declaring a field as an `enum` and listing the possible values of that `enum`. For example, `continent:enum{Africa, Antarctica, Asia, ...}`, declares field `continent` as an `enum` that can only take one of the given values. Cuttlefish parses the given `enum` type into a key-value map, by assigning a unique key to each of the given values, and stores the map with an associated label that can be used internally to refer to this map. Encryption of `enum` values is then reduced to replacing the enum value with the corresponding enum key. The `enum` type is explored by the Cuttlefish compiler to apply the efficient encryption compilation technique to reduce the encrypted data size overhead that can ultimately lead to substantial reductions of query execution latencies.

3.5 Composite Types

Oftentimes, computation is applied only to "subparts" of a value, e.g., a query may extract only the month of a *date* type value and use it in subsequent computations. Usually crypto systems do not allow performing such operations on subparts of an encrypted value, which can limit the query expressiveness. To allow such computations to be performed, Cuttlefish introduces composite types. Composite types are specified according to the following BNF syntax:

$$\begin{aligned} c &:= \texttt{composite}[\ t\ s\ t\ ] \\ t &:= (num : \alpha) \mid (num : \alpha)\ s\ t \\ s &:= \texttt{-} \mid \texttt{/} \mid \texttt{,} \mid \ldots \\ num &:= \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \ldots \end{aligned}$$

where s is a delimiter and α refers to a type such as `int`, `long`, etc. with an optional annotation as shown in Table 6 (except for `composite` – nested composite types are not allowed). These types specify that values of a field are composed of one or more annotation types, allowing Cuttlefish to reason about individual parts. For example, a string of the format `YYYY-MM` representing a date, where `YYYY` is a 4 digit number for the year and `MM` for the month, can be declared as `yearMonth:composite[(4:int[+])-(2:int[range(1-12)])]`. By doing so, Cuttlefish can split the input into its parts and encrypt them individually, which then allows for more expressive queries to be generated.

```
TABLE Lineitem (
 orderkey,       long,   [+],
 linenumber,     long,   [+, unique],
 tax,            double, [accuracy(2), NONE],
 shipdate,       string, [composite[(4:int[+])-
                   (2:int[range(1-12)])-
                   (2:int[range(1-31)])]],
 shipinstruct,   string, [enum{IN_PERSON, MAIL,
                   RETURN, COLLECT}],
 quantity,       long,   [HIGH])
```

Listing 1: Cuttlefish table definition with secure data types

3.6 SDT Example

Listing 1 shows the table definition for a subset of fields of the `Lineitem` table used in TPC-H benchmark, defined using Cuttlefish secure data types. Line 2 declares field `orderkey` with type `long` that holds only positive integers, and line 3 declares field `linenumber` specifying that each value will be a positive and unique integer. Line 4 declares `tax` to be of type `double` with two decimal points of accuracy, and specifies that the `tax` field does not hold sensitive information since it has sensitivity level `NONE`. Line 5 declares `shipdate` field as a `composite` type in the format `YYYY-MM-DD`. Line 8 declares the field to be of `enum` type with four possible values, `IN_PERSON`, `MAIL`, `RETURN` and `COLLECT`. Finally, line 10 declares `quantity` to be of type `long` and sets it as a field of `HIGH` sensitivity. Fields for which sensitivity level is not explicitly given are treated as having a sensitivity level of no less than `LOW`.

4 COMPILATION

When a user submits a query for execution, Cuttlefish compiles it into a query that operates over encrypted data. Our compiler design follows an approach similar to Crypsis [50] to carry out this transformation. In this section we focus on the set of compilation techniques that leverage SDTs and are used by Cuttlefish to support more expressive and efficient queries. Being PHE-centered, our compilation techniques are however different from standard compiler optimizations, and in the light of re-encryption may even seem counter-intuitive.

4.1 Expression Rewriting

The first compilation technique that Cuttlefish employs is to rewrite expressions in queries into simpler but semantically equivalent expressions. The benefits of rewriting expressions are many-fold; from allowing queries not previously able to execute securely in the cloud, to improving efficiency and security of queries, as we explain next, and in subsequent sections.

Composite-type Expression Simplification. Cuttlefish makes use of SDT information on `composite` types to apply Composite-type Expression Simplification (CES) to replace expressions that cannot be executed in the cloud over encrypted data, hence improving expressiveness of queries. As an example, the expression `SUBSTRING(date, 0, 4) == '2000'` which compares the first four digits of a date to a constant cannot be performed in the cloud since there is no crypto system that supports the substring operation. By defining `date` as a composite type, Cuttlefish can automatically replace the substring operation with the encrypted field that contains only the year of the date, therefore rewriting the original expression to `year == '2000'`. As another example consider the expression `phone >= 123-0000000 && phone < 124-0000000`. Declaring phone as `composite[(3:int[+])-(7:int[+])]` to capture the three digit phone area code allows Cuttlefish to replace this expression with the simpler `area_code == 123`. This simplification has more significance than just improving the efficiency of the query, which we further discuss in Section 4.6.

More generally, the CES technique is applied in two phases. In the first phase, the original expression is replaced with a conjunction of sub-expressions applied to *all* sub-parts of the composite type separately. In the second phase, all obsolete sub-expressions are eliminated. As we explain in Section 4.5, the second phase of this compilation technique can interleave with other techniques.

PHE-aware Algebraic Expression Simplification. Cuttlefish can rewrite expressions in a way that reduces (or eliminates altogether) the number of re-encryptions the query requires to complete, by using PHE-aware Algebraic Expression Simplification (PAES). For example, consider the expression $(x + y) \times z$ where x, y and z are initially encrypted under an MHE scheme that only supports multiplication. If approached naïvely, x and y must first be re-encrypted into an AHE scheme so that the two can be added together and then the result must be re-encrypted back to an MHE scheme so that the multiplication with z can be performed. Instead, the expression can be replaced, unconventionally, by the equivalent $x \times z + y \times z$. In this case, $x \times z$ and $y \times z$ can be performed without re-encryptions, then the results can be re-encrypted to an AHE scheme and added together. Therefore, by rewriting the expression, the number of re-encryptions required was reduced from three to two. Note that this transformation goes against traditional compiler optimizations, and it's indeed the opposite of solutions that propose factoring [18, 33].

More generally, the compiler first applies well known expression simplification techniques, namely, Constant Folding (CF), Algebraic Simplification (AS) and Common Subexpression Elimination (CSE). Then, PAES is applied by exhaustively examining semantically equivalent forms of the expression after applying the distributive property and factoring, and considering the schemes the operands of the expression are encrypted under and the subsequent operations they are involved in, with the objective of reducing the number of re-encryptions in the query.

4.2 Condition Expansion

Cuttlefish uses SDT `range` information it has on fields to improve the efficiency or even remove conditions entirely from queries by applying Condition Expansion (CE). For example, knowing that both x and y are non-negative numbers allows the Cuttlefish compiler to expand the condition $x + y > 0$ to $x > 0 \mid\mid y > 0$, and by doing so the query does not have to perform an addition over encrypted data followed by an expensive re-encryption. If instead, x has a range of 50-200 and y has a range of 60-100, the expression $x + y > 100$ can be removed entirely since it will always evaluate to true.

Cuttlefish assigns an initial range to all numerical fields based on their type and on SDT `range` information provided, but importantly, also statically determines the range of a field after each arithmetic or filter operation with a constant or with another field with known range. CE has two general forms. Given a condition of the form $x+y > c$ where c is a constant and x has values in the range a_x to b_x, the compiler will replace it with the condition $y > c-b_x$ && $x+y > c$ where $c - b_x$ is computed during compile time and embedded to the query as a constant.

Following the same logic, given the condition of the form $x+y > z$ where x has values in the range a_x to b_x, and z has values in the range a_z to b_z, the compiler will replace it with the condition $y > a_z - b_x$ && $x + y > z$. Once again, expanding a condition to more clauses might seem counter-intuitive, and goes against traditional compiler optimizations that attempt to simplify conditions by reducing the number of clauses. The intuition here is that if the added clause of the expanded condition evaluates to false, the query does not evaluate the subsequent clauses and avoids unnecessary re-encryptions. These two forms are similarly applied to conditions involving addition, subtraction, multiplication, greater than, greater than or equal, less than, and less than or equal.

4.3 Selective Encryption

Cuttlefish uses the sensitivity level of a field along with the operations that the field is involved in to infer what encryption schemes it should be encrypted under. If a field is marked as non-sensitive (**NONE**), Cuttlefish employs Selective Encryption (SE) to allow that field to exist in the cloud in its plaintext form. This allows for the secondary homomorphic property (cf. Section 2.2) to become applicable which provides increased expressiveness and efficiency in the resulting queries. For example, consider the expression $(x + y) \times z$. Since AHE and MHE are incompatible with each other, after the addition is performed on encrypted data, the result must be re-encrypted under MHE before the multiplication can be performed. Instead, if field z is a non-sensitive field and it is available in the cloud as plaintext, the multiplication can be directly performed using Paillier's secondary homomorphic property, hence avoiding an expensive re-encryption.

SE has the added benefit of reducing the encrypted data size by not encrypting non-sensitive fields unnecessarily while allowing those fields to be involved in operations with sensitive fields.

4.4 Efficient Encryption

One of the sources of overhead when computing over encrypted data stems from the increased data size. To mitigate this, Cuttlefish employs Efficient Encryption (EE) to reduce the size of encrypted data. Firstly, Cuttlefish identifies situations where one field is involved in multiple operations. To accommodate this need, Cuttlefish will have to encrypt the field under multiple crypto systems. When doing so, Cuttlefish minimizes the number of crypto systems a field is encrypted under by recognizing that some crypto systems can accommodate multiple operations. For example, the OPE scheme that Cuttlefish uses is also deterministic, which means that a field that is involved in both equality and order operations needs only be encrypted under the OPE scheme and can avoid having a separate deterministic encryption in the cloud.

Another way Cuttlefish reduces encrypted data overhead is through the use of enumerated types described in Section 3.4. In situations where an enumerated type is involved in deterministic

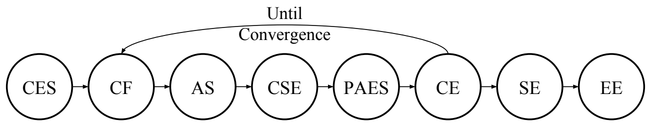

Figure 2: Cuttlefish compilation techniques order

or ordering operations, instead of encrypting the input value itself which could lead to a longer ciphertext, Cuttlefish assigns a unique integer key to it and keeps the mapping of the value-key pairs locally so that when decrypting results the original value can be retrieved. If the relative order of the values is not needed (i.e., when the field is only involved in equality operations) the key sequence is randomized to avoid giving out order information.

In some cases, an operation can be realized over encrypted data through different crypto systems. For example, addition can be achieved by both the Paillier and the ASHE crypto system, multiplication can be achieved by ElGamal and Paillier (if one operand is in plaintext) and equality comparison can be achieved by FNR, AES-DET and SWP (if one operand is a fixed constant). In such cases, Cuttlefish uses SDT information to select the crypto system that is most suited for a field in terms of security requirements and also would lead to more efficient execution. For example, if both operands of a multiplication operation are marked as highly sensitive fields (**HIGH**), the only option is to use ElGamal to encrypt both operands. Instead, if one operand is marked as non-sensitive (**NONE**) and the product of the multiplication is directly followed by an addition (e.g., $x \times y + z$), it is more efficient to use Paillier and its secondary homomorphic property to carry out the multiplication, as it saves the need to re-encrypt the product before performing the subsequent addition. In general, Cuttlefish will always pick ASHE to perform additions over encrypted data, and use Paillier only when the resulting sum is subsequently involved in a multiplication with a non-sensitive value. In the latter case, Paillier is preferred to avoid re-encryption. Since ASHE is a symmetric crypto system, it leads to a smaller encrypted data size overhead and more efficient query execution.

Lastly, Cuttlefish follows the approach of previous work [32, 53] and uses FNR [21], a format preserving encryption which allows Cuttlefish to encrypt data under a DET scheme with no size overhead, as long as the data fits to its 128 block size. For longer values, we use AES in CMC mode as used in CryptDB [44].

4.5 Order of Compilation Techniques

Figure 2 shows the order in which our compilation techniques are applied to all expressions in a query. First, Cuttlefish applies CES to simplify composite expressions. Then the set of CF, AS, CSE, PAES and CE techniques are applied in order. Any of the techniques in this set, can bring the expression to a form in which another technique in the set, not previously applicable, can be applied. Therefore, this set of techniques is applied repeatedly until the expression converges. Finally, the Cuttlefish compiler applies SE and EE to finalize the form of the expression and chose the best crypto systems to use.

4.6 Security Considerations

Apart from allowing for more expressive and efficient queries, Cuttlefish compilation techniques can also make queries more secure by reducing the use of less-secure schemes such as OPE and DET, as we empirically demonstrate in Section 7.2. Section 4.1 outlined an example where an expression involving the `phone` field was simplified from an inequality comparison to an equality comparison (`area_code == 123`) and therefore from requiring OPE, to requiring DET. This transformation improves the security of the query since DET has better security guarantees than OPE. Furthermore, if the field is only involved in comparisons with a constant, the EE compilation technique will further replace the expression with `area_code.matches('123')`. The latter expression requires a SRCH crypto system, which is probabilistic and therefore provides much better security guarantees than OPE or DET.

SDTs provide more opportunities for improving the security of queries. For instance, `range` information can be used to replace the use of OPE with *RangeMatch* [26, 35, 36], which reveals membership of a value to a group (range of items) but does not allow comparison between two values. Similarly, use of OPE and DET can be replaced with *KeywordMatch* [47] which allows a value to be matched to a filtering rule, but does not allow two values to be compared to each other. We plan to add support for RangeMatch and KeywordMatch in a future version of Cuttlefish.

5 PLANNER ENGINE

When an operation cannot be executed over encrypted data, we can either use re-encryption and continue computation in the cloud or continue computation on the client-side over plaintext data (split execution). In this section, we first give an intuition of where re-encryption can lead to better performance than split execution, in the case of such unsupported operations. We then describe the heuristic the Cuttlefish planner engine uses to identify these situations. Then we describe two different ways Cuttlefish can realize the re-encryption service and finally, we describe a set of optimizations that reduce the cost of re-encryption.

5.1 Re-encryption Heuristic

To identify which of the two options will lead to better performance we start by observing that the following hold *oftentimes* for analytical queries (we empirically verify these observations in Section 7):

a. Queries start with a large volume of input data but the results of the query are much smaller in size.

b. Queries over encrypted data include unsupported operations involving small amounts of data, the results of which are then used in computations with larger amounts of data.

c. Incurring a higher cost for re-encrypting (decrypting and encrypting afresh) compared to split execution can be favorable because it allows a larger portion of the query to be executed in the cloud (more compute resources) and it can reduce the amount of data that needs to be shipped to the client and subsequently decrypted with the latter option.

On the other hand, the ability to unrestrictedly re-encrypt from one scheme to another can allow an adversary to infer information about the underlying data. For instance, allowing an AHE scheme to be re-encrypted to DET, allows the adversary to repeatedly add 1 to an AHE encrypted value, re-encrypt it to a DET scheme and compare it to a *known* DET constant until the two match. By counting the number of steps it took before a match, the adversary can infer the original value. Similar attacks can be performed for any scheme that allows comparisons, e.g., OPE, DET and SRCH. To prevent data leakage, Cuttlefish does not allow re-encryptions from a scheme that allows arithmetic operations to one that allows comparisons over encrypted data. Restricting re-encryption does not limit the expressiveness of queries but can cause Cuttlefish to use split execution at an earlier stage in the query, as we describe next.

Cuttlefish stores profiles of previously executed queries that capture critical performance characteristics, similar to the approach used by Verma et al. [54]. In particular, we store the execution time and selectivity (ratio of input-to-output size) at various stages of a query (we give more insight on how these stages are defined in the algorithm below). This information is used when a new query is submitted to generate the following estimates for each stage of the query, denoted by i:

Q_i^E – time estimate for executing stage i over encrypted data.

Q_i^P – time estimate for executing the rest of the query starting from stage i over plaintext data.

$V_{i,f}$ – size estimate (bytes) of field f at stage i.

E_f – time estimate to encrypt 1 byte of field f.

D_f – time estimate to decrypt 1 byte of field f.

Steps. Once the directed acyclic graph (DAG) that captures the query computation is generated, Cuttlefish greedily considers each data flow branch of the DAG and identifies all operations that are not supported in the cloud $i = 1 \ldots N$ where operation N is the first unsupported operation and operation 1 is the last unsupported operation in the query, which in turn defines stage i to be the portion of the query between unsupported operation i and $i - 1$.

(1) For each unsupported operation i, generate two sets: $RencSet_i$ represents all fields that need to be sent to the re-encryption service in case of re-encryption, $SplitSet_i$ all fields that need to be sent to the client in case of split execution.

(2) For each unsupported operation i, estimate the cost of re-encryption step R_i as:

$$R_i = \sum_{f=1}^{||RencSet_i||} V_{i,f}(D_f + E_f)$$

or $R_i = \infty$ if re-encryption $D_f \rightarrow E_f$ is restricted, and the cost of split execution step S_i as:

$$S_i = \sum_{f=1}^{||SplitSet_i||} V_{i,f} D_f$$

(3) For each unsupported operation i, decide whether to use re-encryption or split execution based on:

$$C_i = Min(C_i^R, C_i^C)$$

where

$$C_i^R = R_i + Q_i^E + C_{i-1},\ C_i^C = S_i + Q_i^P$$

and $C_0 = S_0 + Q_0^P$ is the cost of sending the final results to the client and decrypting them.

C_i^R is the overall cost of performing a re-encryption at position i and is calculated as the cost of the re-encryption step R_i added to the cost of executing stage i over encrypted data and the cost of the previous position of C_{i-1}. Similarly, C_i^C is the cost of using split execution at position i, which includes the split execution step S_i added to Q_i^P which is the cost of performing the rest of the query over plaintext data at the client. The algorithm recursively estimates the cost of the query at each stage (C_i), starting from the end of the query and moving backwards. For each unsupported operation i, Cuttlefish chooses re-encryption if $C_i^R < C_i^C$, otherwise it uses split execution.

5.2 Re-encryption Service

Cuttlefish supports two different implementations of the re-encryption service.

Client-side re-encryption. In the absence of specialized hardware in the cloud, Cuttlefish uses a remote client-side re-encryption service. In this case, data is shipped to the remote re-encryption service for re-encryption before sent back to the cloud.

SGX-based re-encryption. If specialized hardware such as Intel SGX is available in the cloud, Cuttlefish can employ an in-enclave SGX-based re-encryption service to remove the need to ship data in a remote location. Importantly, not all cloud nodes need to contain specialized hardware. Since re-encryption (when needed) constitutes only a fraction of a query, it can easily be offloaded to a small set of nodes that support Intel SGX, while the bulk of the computation can be handled by non-specialized nodes. This aligns well with the gradual adaptation of specialized hardware in public clouds. Furthermore, the well-defined function of re-encryption fits Intel's programming guide suggestions [5] on how to securely use SGX well. Specifically, it allows for a very small and easily verifiable enclave code (cf. Section 6), which keeps the trusted component and the number of interfaces between the trusted and untrusted components small while remaining application independent.

5.3 Re-encryption Optimizations

Since encrypting and decrypting data can be costly operations, Cuttlefish uses a set of optimizations aiming to reduce the overall cost of re-encryption.

Caching. After a value is decrypted from or encrypted to a deterministic scheme, the ciphertext-plaintext pair can be cached as proposed in previous work [53]. Subsequent requests to encrypt or decrypt a cached value are then completed much faster by simply referring to the cached data. This optimization only applies to deterministic crypto systems such as DET and OPE.

Speculative Re-encryption. Cuttlefish can further optimize re-encryption by speculating what values are more likely to be re-encrypted. Cuttlefish achieves this by using SDT `range` information to predict the range of values that need to be re-encrypted. When this range of values is bounded, Cuttlefish generates a small map of pre-encrypted values containing the values most likely to be requested for re-encryption. Populating this map occurs when the re-encryption service is idle, to avoid slowing down other re-encryption requests. Unlike caching, speculative re-encryption applies to all crypto systems.

Random Number Pre-computation. Some crypto systems require large random numbers as part of encrypting data (e.g., Paillier and ElGamal). Cuttlefish pre-computes and stores a small amount of such random numbers whenever the re-encryption service is idle, so that re-encryption requests can be served faster.

6 IMPLEMENTATION

Our prototype implementation of Cuttlefish is made up of several components as shown in Figure 1. The Cuttlefish compiler is implemented in 5500 lines of Scala code and it includes a parser, a transformation module and a metadata module. The parser parses queries written in Apache Spark that may optionally contain secure data type information. The transformation module is responsible for generating the *remote query* that is deployed in the cloud and operates over encrypted data and the *local query* that will be executed on the client side to decrypt and generate the final results. The metadata module stores information about the state of the encrypted database (e.g., encrypted table schemas, encryption key ids, enum maps etc...) in the form of XML files, which is then used by the transformation module when transforming queries.

The cloud service runs an unmodified Apache Spark service. Operations on encrypted data are implemented through the use of UDFs written in 2200 lines of Scala code, by extending the *UserDefinedFunction* class.

The client-side re-encryption service is implemented as a Java server using Java sockets. Requests for re-encryption are made through special UDFs that first establish a connection with the re-encryption service and then send data for re-encryption. Each Spark executor uses a dedicated connection with the server and in turn, the re-encryption server uses multiple threads to handle these connections.

SGX-based re-encryption is implemented as an Intel SGX enclave using 1100 lines of C code, where the majority of the code (over 900 lines) accounts for the implementation of the crypto systems and therefore this code has been well tested over the years. To implement these crypto systems we port a small multiple-precision arithmetic library into SGX called *BigDigits* [2] (version 2.6) that allows us to do big number computations. Similarly to the client-side re-encryption, re-encryption requests to SGX are made through UDFs which use JNI to access an interface written in C which then calls the appropriate function in the enclave.

7 EVALUATION

In this section, we empirically assess the benefits of our proposed abstractions and techniques. We evaluate Cuttlefish on three different aspects. We first evaluate how valuable our compilation techniques are in improving expressiveness and performance by examining how frequently they apply in analytical queries (Section 7.2). We then compare Cuttlefish to other systems to observe relative performance benefits (Section 7.3). Finally, to understand the benefits of our proposed abstractions and techniques individually, we compare the execution time of Cuttlefish to the execution on plaintext and we examine how effective each of our compilation techniques and re-encryption heuristic are in improving performance (Section 7.4).

7.1 Experimental Setup

To evaluate Cuttlefish we use the following two standard industry adopted-benchmarks:

(1) TPC-H is a decision support benchmark comprising a set of 22 queries. These queries are designed to have broad industry-wide relevance and be representative of realistic decision support queries with a high degree of complexity that give answers to critical business questions.
(2) TPC-DS is an industry standard benchmark for big data decision solutions comprising a set of 100 queries. This benchmark models a retail product supplier, with multiple users running interactive and data mining queries. We use a subset of 19 queries that is used by an existing industry benchmark (Cloudera's Impala benchmark [20]) and by recent work on studying performance bottlenecks in distributed computation frameworks [40].

We compared Cuttlefish with other systems and evaluated individual techniques on both benchmarks. Due to lack of space, we show the comparison with other systems on TPC-H and the evaluation of individual techniques on TPC-DS.

We performed all our experiments using Amazon EC2 instances. All reported execution times in the experiments are the average of 5 runs. We used a cluster of 20 *m4.xlarge* instances to represent the untrusted cloud, each with 4 virtual CPUs and 16GB of memory. We also used a single *c4.2xlarge* EC2 instance with 8 virtual CPUs and 15GB of memory as the trusted client-side node, where both the client-side re-encryption service was hosted, and the local query is executed to decrypt and generate the final results. To account for network latencies, we deployed the untrusted cloud cluster in Amazon's N. Virginia datacenter and deployed the trusted client-side node in Amazon's Ohio datacenter. We retain the default EC2 network throughput for all nodes within the cloud (750Mbps for *m4.xlarge* instances). We use HDFS as our storage medium with a replication factor of 3. We built the enclave running the SGX-based re-encryption using Linux SGX SDK version 1.8.100.37739 and due to the unavailability of Intel SGX hardware in Amazon EC2, we set the SGX mode of the SDK to simulation. Decryption keys are passed to the enclave via a secure channel after the enclave is initialized and remotely attested.

Similar to other PHE-based systems [44, 50, 53], we assume that data is already encrypted and securely stored in the cloud, and hence do not include encryption latency in our evaluations. Similarly, setting up decryption keys happens once at the start of the cloud service and therefore our evaluation does not include enclave initialization and attestation times.

7.2 Compilation Techniques Applicability

We first analyze how applicable our proposed compilation techniques are in TPC-H and TPC-DS benchmarks. Table 7 shows the number of queries each compilation technique was applicable to for both benchmarks. Expression rewriting allows for 4 queries of TPC-H and 2 queries in TPC-DS to be executed over encrypted data by using SDT `composite` types to replace substring operations. In addition, `composite` types allowed Cuttlefish to simplify expressions as discussed in Section 4.1 in 12 TPC-H queries and 1 TPC-DS query. PHE-aware algebraic expression simplification applies to 10 TPC-H queries and 4 TPC-DS queries. Overall, expression rewriting applies to 15 TPC-H queries and 6 TPC-DS queries. Condition expansion applies only to 2 queries of TPC-DS but in those cases, the performance benefits are substantial since it reduces the amount of data the rest of the query is computed over. Selective encryption applies to 7 TPC-H and 7 TPC-DS queries. Crucially, selective encryption in TPC-H has the effect of reducing the number of re-encryptions required. Before selective encryption is applied, 9 TPC-H queries (Q03, Q05, Q10, Q11, Q15, Q17, Q18, Q20 and Q22) require re-encryption to complete whereas after selective encryption is applied, only 8 queries require re-encryption (Q17 no longer needs re-encryption). This leads to substantial improvements in latency. Efficient encryption is a widely applicable technique and it applies to all TPC-H and TPC-DS queries, reducing the encrypted data size overhead. Furthermore, efficient encryption in combination with expression rewriting, allows for 2 TPC-H fields that would otherwise require OPE, to be replaced with SRCH and 1 field that would require DET to be replaced with SRCH, improving the security of 12 queries that involve those fields. Similarly, 1 TPC-DS field is changed from OPE to SRCH and another from DET to SRCH.

Compilation technique	Number of queries	
	TPC-H	TPC-DS
Expression rewriting	15	6
Condition expansion	0	2
Selective encryption	7	7
Efficient encryption	22	19

Table 7: Compilation technique applicability out of a total of 22 TPC-H queries and 19 TPC-DS queries.

7.3 System Performance

To evaluate the performance overhead of our system, we run TPC-H and compare the execution time of Cuttlefish with the plaintext execution and the execution time of other PHE-based systems. We run TPC-H at scale factor 100, i.e., 100GB of data before encryption. TPC-H data is divided across 8 tables with a total of 61 fields. We represent 5 of these fields as `enum` types (ship-instruction, nation-name, region-name, order-status and ship-mode) and another 5 fields as `composite` types (customer-phone, shipdate, receiptdate, commitdate and orderdate). In addition, we mark 14 fields as positive numbers (`+`) and another 7 with a more strict `range`. Lastly, we mark 2 fields as non-sensitive (quantity and discount). We capped the execution time to 30 minutes for each query execution and marked queries not completed in that time as timed out.

Plaintext represents the execution time of TPC-H over plaintext data. Cuttlefish-TH is the execution time of Cuttlefish when trusted hardware is available in the cloud and therefore the re-encryption service used is the SGX-based re-encryption. Cuttlefish-CS shows the execution time of Cuttlefish but in this case a Client-Side re-encryption is utilized. Next, we include the execution time of the Monomi [53] system, which we implemented in Spark as opposed

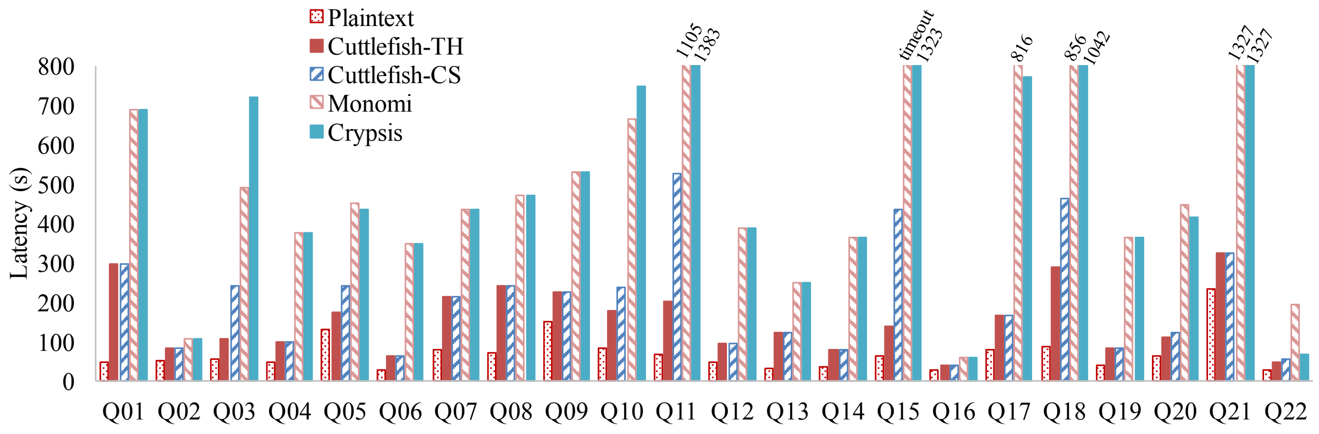

Figure 3: TPC-H latency of Cuttlefish compared to plaintext and other PHE-based systems

to the proposed centralized database design (Postgres). Monomi does not support an MHE scheme or a SRCH scheme that can handle all TPC-H queries (originally, Monomi could not handle Q13, Q15 and Q16). Thus, we extend Monomi to use MHE and our implementation of the SRCH scheme. Furthermore, Monomi requires pre-computation to handle complex arithmetic expressions. We allow Monomi to use pre-computation only in cases where doing otherwise would cause split execution to fail (i.e., split at the very beginning of the query). For a fair comparison, we allow the same pre-computation for all evaluated systems. Lastly, we compare Cuttlefish to Crypsis [50] which we also implemented in Spark (originally implemented in Apache Pig [1, 39]). Neither Monomi or Crypsis can utilize SDTs or apply any of our proposed compilation techniques, nor can they use our planner engine. Therefore, Monomi is limited to using split execution to overcome limitations of PHE and Crypsis is limited to naïve re-encryption. As shown in Figure 3, the average overheads of Cuttlefish-TH, Cuttlefish-CS, Monomi and Crypsis across all 22 queries, normalized to Plaintext execution time, are 2.34×, 3.05×, 7.83×, and 8.68× respectively. Cuttlefish-TH has lower overhead than Cuttlefish-CS because the former, re-encrypts data locally using SGX enclaves in the cloud, and does not incur network latency. The significant lower overhead of both Cuttlefish approaches compared to Monomi and Crypsis is due to the compilation techniques enabled by the use of SDTs which led to substantially reduced encrypted data size, less use of re-encryption and overall more optimal query plans.

7.4 Compilation Technique Performance

To evaluate the performance improvement due to each individual compilation technique, we run 19 TPC-DS queries with different compilation techniques enabled each time. We use a scale factor of 100 resulting to 100GB of data before encryption, divided into 9 tables with a total of 166 fields across all tables. We selected 3 fields (tax, coupon-amount and sale-price) that based on their name seem to not hold sensitive information and marked them with `NONE` to indicate that they can remain in plaintext form. We defined 6 fields as `enum` types and defined 14 fields that hold dates, zipcodes or addresses as `composite` types. In addition, 22 fields were marked as positive integers and 16 fields were assigned a more specific `range`, based on the information provided in the TPC-DS data generator.

Figure 4 shows the results of this evaluation. The Plaintext bar indicates the execution time when executing over plaintext data. Cuttlefish-TH shows the execution time of Cuttlefish with SGX-based re-encryption and all compilation techniques enabled. Subsequent bars show execution times with one less compilation technique applied. For example, -Expression rewriting indicates that expression rewriting is disabled, -Condition expansion indicates that *in addition* to expression rewriting, condition expansion is also disabled and so on.

The average overheads of Cuttlefish-TH, -Expression rewriting, -Condition expansion, -Selective Encryption and -Efficient Encryption across all 19 queries, normalized to Plaintext execution time, are 1.69×, 1.87×, 2.02×, 3.16× and 4.23× respectively. Condition expansion applies only to Q34 and Q59. Q65 has an overhead of 4.88× for Cuttlefish-TH compared to Plaintext. This is because Q65 makes heavy use of re-encryption even after all compilation techniques are applied. Q19, Q34, Q53, Q63, Q89 and Q98 benefit significantly when Selective encryption is enabled. This indicates that even when a very small number of fields is marked as non-sensitive (e.g., 3 out of a total of 166, as is the case in this evaluation), Cuttlefish enables compilation techniques that improve performance significantly.

8 RELATED WORK

8.1 Computing over Encrypted Data

In their pioneering work, Popa et al. [44] proposed CryptDB, a system that provides confidentiality for applications using a SQL database backend. CryptDB works by executing SQL queries over encrypted data using a collection of encryption schemes and also uses a trusted proxy to analyze queries. Monomi [53] builds off of CryptDB's design and introduces split client/server query execution to support more queries. In addition, Monomi proposes techniques that improve performance for different workloads and adds a designer to automatically choose an efficient design suitable for each workload. Talos [46] encrypts and stores data specific to the *Internet of Things* in a cloud database while still allowing query processing. Unlike Cuttlefish, these systems do not allow Spark-style parallelization of queries. Furthermore, they depend on data definitions provided by MySQL or Postgres databases which makes

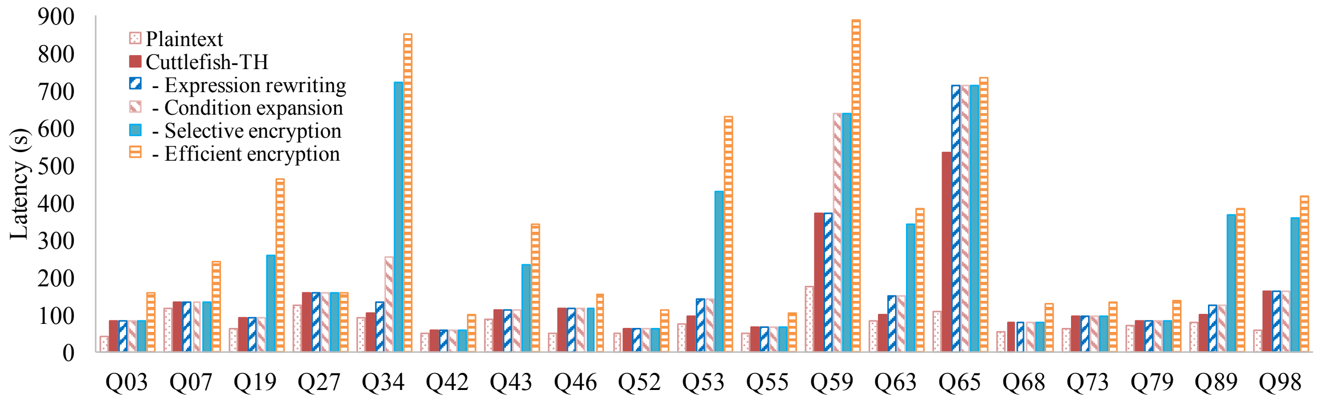

Figure 4: TPC-DS latency with different compilation techniques enabled

it difficult to leverage optimizations that Cuttlefish uses that rely on specifying data ranges, precisions, or sensitivity levels.

Crypsis [50] is a modified Pig Latin runtime that performs data flow analysis and query transformations for Pig Latin scripts, automatically and transparently enabling their execution on encrypted data. Unlike Cuttlefish, Crypsis does not propose novel abstractions to allow compile-time and runtime optimization techniques such as expression rewriting, caching/speculative re-encryption, etc. Furthermore, Crypsis solely uses re-encryption to overcome limitations of PHE, which can lead to increased execution times. Instead, Cuttlefish uses a planner engine to overcome PHE limitations without compromising performance. The performance improvements we have over Crypsis because of these techniques are evident from Section 7.

8.2 Data Constraints

Managing information flows with multiple sensitivity levels within programs has been widely studied. Denning's seminal work [23, 24] in the field was further formulated as a type system by Volpano et.al. [55]. These systems leverage type information with additional policy specifications to perform information flow analysis at compile or run time to ensure that programs do not leak sensitive data. Unlike Cuttlefish, these techniques do not use sensitivity levels to optimize query processing time. In programming language research, similar fine grained type specifications have been explored under refinement type systems [22, 27]. The primary focus in these approaches are type inference, checking and correctness. In the context of encrypted query processing, CryptDB [44] allows application developers to annotate the database schema, define principals and specify access permission of each principal. Unlike Cuttlefish, CryptDB does not utilize fine grained data type information (e.g., `range`, `unique`, etc.) to perform optimizations.

8.3 Trusted Hardware

A number of recent approaches rely on specialized hardware, e.g., Intel SGX [37] that provides a trusted area of execution to preserve data confidentiality while computing. Haven [12] relies on hardware encrypted and integrity protected physical memory provided by SGX to ensure application security. Cipherbase [8] provides an FPGA-based implementation of a trusted hardware that can be used to run a commercial SQL database system without sacrificing data confidentiality. TrustedDB [11] preserves confidentiality by using a server-hosted, tamper-proof trusted hardware in critical query processing stages. Iron [13] provides a practical implementation of functional encryption using SGX enclaves. Opaque [57] is a data oblivious distributed data analytics platform built using SGX enclaves for secure computation.

While trusted hardware can enable secure computations at practical performance overheads, it suffers from portability deficiencies since it relies on specific hardware. Instead, Cuttlefish can preserve confidentiality even if no specialized hardware is available in the cloud. In addition, SGX enclaves can only (efficiently) access a limited amount of trusted memory [10, 17]. This problem is likely exacerbated in multi-tenant cloud environments where SGX needs to be virtualized and securely shared among users. Currently, only static memory allocation is supported [4], meaning pages are statically allocated and mapped to a VM when it is created, and only released when the VM is destroyed, making trusted memory, even more so, a scarce resource. Unlike hardware-only approaches, Cuttlefish can operate directly on encrypted data in untrusted memory and can handle large workloads (e.g., 100s of GBs as shown in Section 7) which are too big to run directly in SGX, thereby benefiting from in-memory computations [56], and uses SGX only for re-encryption of small amounts of data.

9 CONCLUSIONS

Both software- and hardware-based approaches have been proposed to support data analytics in the public cloud while preserving data confidentiality. Both have clear limitations, for instance in terms of expressiveness for PHE in the former case, and in terms of lack of portability of Intel SGX in the latter case.

This paper presents Cuttlefish, a system for confidentiality preserving data analytics in the cloud which essentially relies on partially homomorphic encryption, combined with smart re-encryption exploiting SGX when available to overcome limitations of PHE. To this end, Cuttlefish leverages three key concepts: the novel abstraction of secure data types, a set of compilation techniques, and a planner engine for efficient execution. We have demonstrated how these contributions in combination allow for practical performance.

REFERENCES

[1] Apache Pig. http://pig.apache.org.

[2] BigDigits multiple-precision arithmetic library. http://www.di-mgt.com.au/bigdigits.html.

[3] Intel SGX. https://software.intel.com/en-us/isa-extensions/intel-sgx.

[4] SGX Virtualization. https://01.org/intel-software-guard-extensions/sgx-virtualization.

[5] 2014. Intel Software Guard Extensions Programming Reference. (2014). https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[6] 2014. Synopsys, Inc., Open Source Report 2014. (2014). http://go.coverity.com/rs/157-LQW-289/images/2014-Coverity-Scan-Report.pdf.

[7] George J Annas. 2003. HIPAA regulations-a new era of medical-record privacy? *New England Journal of Medicine* 348, 15 (2003), 1486–1490.

[8] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. 2013. Orthogonal Security with Cipherbase. In *Biennial Conf. on Innovative DataSystems Research (CIDR)*.

[9] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuellar, and Llanos Tobarra. 2008. Formal Analysis of SAML 2.0 Web Browser Single Sign-on: Breaking the SAML-based Single Sign-on for Google Apps. In *W. on Formal Methods in Security Engineering (FMSE)*. 1–10.

[10] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *Symp. on Op. Sys. Design and Implementation (OSDI)*. 689–703.

[11] Sumeet Bajaj and Radu Sion. 2014. TrustedDB: A Trusted Hardware-Based Database with Privacy and Data Confidentiality. *IEEE Trans. Knowl. Data Eng.* 26, 3 (2014), 752–765.

[12] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *Symp. on Op. Sys. Design and Implementation (OSDI)*. 267–283.

[13] Dan Boneh Ben A. Fisch, Dhinakaran Vinayagamurthy and Sergey Gorbunov. 2016. Iron: Functional Encryption using Intel SGX. Cryptology ePrint Archive, Report 2016/1071. (2016). http://eprint.iacr.org/2016/1071.

[14] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. 2009. Order-Preserving Symmetric Encryption. In *Int. Conf. on The Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. 224–241.

[15] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. 2011. Order-preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions. In *Annual Int. Cryptology Conf. (CRYPTO)*. Springer-Verlag, 578–595.

[16] Dan Boneh, Kevin Lewi, Mariana Raykova, Amit Sahai, Mark Zhandry, and Joe Zimmerman. 2015. Semantically Secure Order-Revealing Encryption: Multi-input Functional Encryption Without Obfuscation. In *Int. Conf. on The Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. 563–594.

[17] Stefan Brenner, Colin Wulf, Matthias Lorenz, Nico Weichbrodt, David Goltzsche, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. 2016. SecureKeeper: Confidential ZooKeeper using Intel SGX. In *Int. Conf. on Middleware (MIDDLEWARE)*. 14:1–14:13.

[18] Florian Cajori. 1911. Horner's method of approximation anticipated by Ruffini. *Bull. Amer. Math. Soc.* 17, 8 (05 1911), 409–414.

[19] Nathan Chenette, Kevin Lewi, Stephen A. Weis, and David J. Wu. 2016. Practical Order-Revealing Encryption with Limited Leakage. In *Int. Conf. on Fast Software Encryption (FSE) (FSE 2016)*. 474–493.

[20] Cloudera. A TPC-DS like benchmark for Cloudera Impala. https://github.com/cloudera/impala-tpcds-kit.

[21] Sashank Dara and Scott R. Fluhrer. 2014. FNR: Arbitrary Length Small Domain Block Cipher Proposal. In *Int. Conf. on Security, Privacy, and Applied Cryptography Engineering (SPACE)*. 146–154.

[22] Rowan Davies. 2005. *Practical Refinement-type Checking*. Ph.D. Dissertation. AAI3168521.

[23] Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Commun. ACM* (1976), 236–243.

[24] Dorothy E. Denning and Peter J. Denning. 1977. Certification of Programs for Secure Information Flow. *Commun. ACM* (1977), 504–513.

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

[26] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel-Catalin Rosu, and Michael Steiner. 2015. Rich Queries on Encrypted Data: Beyond Exact Matches. In *Symp. on Research in Computer Security (ESORICS)*. 123–145.

[27] Tim Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Conf. on Prog. Lang. Design and Implementation (PLDI)*. ACM, 268–277.

[28] Craig Gentry. 2009. *A Fully Homomorphic Encryption Scheme*. Ph.D. Dissertation. Advisor(s) Boneh, Dan. AAI3382729.

[29] Craig Gentry, Shai Halevi, and Nigel P. Smart. 2012. Homomorphic Evaluation of the AES Circuit. *IACR Cryptology ePrint Archive* (2012). Informal publication.

[30] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. 2017. Leakage-Abuse Attacks against Order-Revealing Encryption. In *Symp. on Security and Privacy (S&P)*. 655–672.

[31] Hakan Hacigümüs, Balakrishna R. Iyer, Chen Li, and Sharad Mehrotra. 2002. Executing SQL over encrypted data in the database-service-provider model. In *Int. Conf. on the Mgt. of Data (SIGMOD)*. 216–227.

[32] Alon Y. Halevy. 2001. Answering Queries Using Views: A Survey. *The VLDB Journal* 270–294.

[33] A. Hosangadi, F. Fallah, and R. Kastner. 2006. Optimizing Polynomial Expressions by Algebraic Factorization and Common Subexpression Elimination. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2006), 2012–2022.

[34] Florian Kerschbaum. 2015. Frequency-Hiding Order-Preserving Encryption. In *Int. Conf. on on Computer and Communications Security (CCS)*. ACM, 656–667.

[35] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. 2016. Embark: Securely Outsourcing Middleboxes to the Cloud. In *Networked Sys. Design and Implem. (NSDI)*. 255–273.

[36] Kevin Lewi and David J. Wu. 2016. Order-Revealing Encryption: New Constructions, Applications, and Lower Bounds. In *Int. Conf. on on Computer and Communications Security (CCS)*. 1167–1178.

[37] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative instructions and software model for isolated execution. In *W. on Hardware and Architectural Support for Security and Privacy (HASP)*. 10.

[38] Muhammad Naveed, Seny Kamara, and Charles V. Wright. 2015. Inference Attacks on Property-Preserving Encrypted Databases. In *Int. Conf. on on Computer and Communications Security (CCS)*. ACM, 644–655.

[39] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig Latin: A Not-so-foreign Language for Data Processing. In *Int. Conf. on the Mgt. of Data (SIGMOD)*. 1099–1110.

[40] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In *Networked Sys. Design and Implem. (NSDI)*. 293–307.

[41] Pascal Paillier. 1999. Public-key Cryptosystems Based on Composite Degree Residuosity Classes. In *Int. Conf. on The Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. 223–238.

[42] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. 2016. Big Data Analytics over Encrypted Datasets with Seabed. In *Symp. on Op. Sys. Design and Implementation (OSDI)*.

[43] Raluca Ada Popa, Frank H. Li, and Nickolai Zeldovich. 2013. An Ideal-Security Protocol for Order-Preserving Encoding. In *Symp. on Security and Privacy (S&P)*. 463–477.

[44] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. Cryptdb: protecting confidentiality with encrypted query processing. In *Symp. on Op. Sys. Principles (SOSP)*. 85–100.

[45] Thomas Ristenpart and Eran Tromer. 2009. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Int. Conf. on on Computer and Communications Security (CCS)*. 199–212.

[46] Hossein Shafagh, Anwar Hithnawi, Andreas DroÌĹscher, Simon Duquennoy, and Wen Hu. 2015. Talos: Encrypted Query Processing for the Internet of Things. In *Conf. on Embedded Networked Sensor Sys. (SenSys)*.

[47] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. 2015. BlindBox: Deep Packet Inspection over Encrypted Traffic. In *Int. Conf. on Data Communication (SIGCOMM)*. 213–226.

[48] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. 2000. Practical Techniques for Searches on Encrypted Data. In *Symp. on Security and Privacy (S&P)*. 44–55.

[49] Julian James Stephen, Savvas Savvides, Russell Seidel, and Patrick Eugster. 2014. Practical Confidentiality Preserving Big Data Analysis. In *W. on Hot Topics in Cloud Computing (HotCloud)*.

[50] Julian James Stephen, Savvas Savvides, Russell Seidel, and Patrick Th. Eugster. 2014. Program analysis for secure big data processing. In *Int. Conf. on Automated Software Engineering (ASE)*. 277–288.

[51] Julian James Stephen, Savvas Savvides, Vinaitheerthan Sundaram, Masoud Saeida Ardekani, and Patrick Eugster. 2016. STYX: Stream Processing with Trustworthy Cloud-based Execution. In *Symp. on Cloud Computing (SoCC)*. 348–360.

[52] Sai Deep Tetali, Mohsen Lesani, Rupak Majumdar, and Todd D. Millstein. 2013. MrCrypt: Static analysis for secure cloud computations. In *Conf. on Object-Oriented Prog. Sys., Lang. and Applications (OOPSLA)*. 271–286.

[53] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. 2013. Processing Analytical Queries over Encrypted Data. *Proc. VLDB Endow.* 6, 5 (2013), 289–300.

[54] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. 2011. ARIA: Automatic Resource Inference and Allocation for Mapreduce Environments. In *Int. Conf. on Autonomic Computing (ICAC)*. 235–244.

[55] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security* 167–188.

[56] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Networked Sys. Design and Implem. (NSDI)*. 15–28.

[57] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *Networked Sys. Design and Implem. (NSDI)*. 283–298.