PRACTICAL CONFIDENTIALITY-PRESERVING DATA ANALYTICS IN UNTRUSTED CLOUDS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Savvas Savvides

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2020

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

To my friends and family who made this work possible.

ACKNOWLEDGMENTS

I wish to express my gratitude to my supervisor, Prof. Patrick Eugster for his continuous support and guidance, as well as Prof. Jan Vitek, Prof. Cristina Nita-Rotaru and Prof. Byoungyoung Lee who I had the honor of collaborating with throughout my Ph.D. journey.

I would also like to thank my thesis committee members, Prof. Xiangyu Zhang, Prof. Pedro Fonseca, and Prof. Sonia Fahmy for their insightful comments and feedback.

Last but not least, I would like to thank my collaborators Julian Stephen, Malte Viering, Seema Kumar, Marc Arndt, Darshika Khandelwal, Masoud Ardekani, Vinai Sundaram, and Denis Ulybyshev for their invaluable help as well as my fellow Ph.D. students Janos Horvath, Bradley Fitzgerald, Bara Abusalah, Adil Ahmad, and Aveek Dutta for some interesting discussions, work related and non-work related.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Savvides, Savvas Ph.D., Purdue University, August 2020. Practical Confidentiality-Preserving Data Analytics in Untrusted Clouds. Major Professor: Patrick Eugster.

Cloud computing offers a cost-efficient data analytics platform. This is enabled by constant innovations in tools and technologies for analyzing large volumes of data through distributed batch processing systems and real-time data through distributed stream processing systems. However, due to the sensitive nature of data, many organizations are reluctant to analyze their data in public clouds. To address this stalemate, both software-based and hardware-based solutions have been proposed yet all have substantial limitations in terms of efficiency, expressiveness, and security. In this thesis, we present solutions that enable practical and expressive confidentiality-preserving batch and stream-based analytics. We achieve this by performing computations over encrypted data using Partially Homomorphic Encryption (PHE) and Property-Preserving Encryption (PPE) in novel ways, and by utilizing remote or Trusted Execution Environment (TEE) based trusted services where needed.

We introduce a set of extensions and optimizations to PHE and PPE schemes and propose the novel abstraction of *Secure Data Types (SDTs)* which enables the application of PHE and PPE schemes in ways that improve performance and security. These abstractions are leveraged to enable a set of compilation techniques making data analytics over encrypted data more practical. When PHE alone is not expressive enough to perform analytics over encrypted data, we use a novel *planner engine* to decide the most efficient way of utilizing *client side completion*, *remote re-encryption*, or *trusted hardware re-encryption* based on Intel Software Guard eXtensions (SGX) to overcome the limitations of PHE. We also introduce two novel *symmetric PHE* schemes that allow arithmetic operations over encrypted data. Being symmetric, our schemes are more efficient than the state-of-the-art asymmetric PHE schemes without compromising the level of security or the range of homomorphic operations they support. We apply the aforementioned techniques in the context of batch data analytics and demonstrate the improvements over previous systems. Finally, we present techniques designed to enable the use of PHE and PPE in resource-constrained *Internet of Things (IoT) devices* and demonstrate the practicality of stream processing over encrypted data.

1. INTRODUCTION

The advent of cloud computing has decisively changed the landscape of computing. By making practically infinite amounts of computation and storage resources available on-demand — to individuals, small and medium size enterprises, but also large organizations — the cloud computing paradigm has had a profound economic impact, in addition to a substantial technical one. While driving the big data evolution, the offloading of computation and storage to public clouds bears increased security concerns [1, 2]. Hardware and software bugs can be exploited by malicious actors who can cause applications to fail and reveal confidential information. The risks of confidentiality breaches are only exacerbated by the multi-tenant nature of public clouds, with benign and malicious actors sharing the same cloud resources. Even in the absence of malicious actors, curious administrators can still leak confidential information. In the presence of such threats, innovative solutions are needed that allow us to utilize cloud resources without sacrificing confidentiality of data.

1.1 Preserving Confidentiality

To ensure confidentiality, data can be encrypted before sent to the cloud. Traditional data encryption preserves confidentiality of data *at rest*. Securing data at rest allows users to utilize cloud storage without exposing sensitive information, but it prevents users from making use of the compute capabilities of the cloud to carry out data analytics. To preserve the confidentiality of data *in use* existing cryptographic systems (cryptosystems) that allow meaningful operations to be performed directly on encrypted data can be used. These cryptosystems are called *homomorphic encryption schemes*. Fully Homomorphic Encryption (FHE) schemes allow arbitrary computations over encrypted data. Unfortunately, despite regular advancements, FHE schemes still exhibit high overhead for complex analytical queries. Instead, in this work we use Partially Homomorphic Encryption (PHE) and Property-Preserving Encryption (PPE) schemes that allow computations over encrypted data with respect to some specific operations and have practical performance. PHE and PPE schemes preserve the confidentiality of data at rest, in motion, and in use, allowing users to utilize both the storage and the compute capabilities of the cloud, without exposing sensitive information.

1.2 Challenges

Several confidentiality-preserving systems exist that use PHE and PPE schemes to perform operations over encrypted data. Yet these systems suffer from a set of drawbacks.

Firstly, existing systems largely ignore properties of individual PHE and PPE schemes such as a) the level of security they provide, b) the performance of their encryption, decryption, and homomorphic operations, and c) their expressiveness in terms of the range of homomorphic operations they support. By not considering these properties, existing systems deploy queries that are less secure and less performant.

Secondly, existing systems utilize asymmetric PHE schemes to carry out arithmetic operations over encrypted data. The security of these schemes relies on mathematical problems (e.g., integer factorization) which are hard to solve for large numbers. Because of this fact, the ciphertext size of these schemes is large and the homomorphic operations they support are computationally expensive. The asymmetric nature of these schemes offers no benefits when computing over encrypted data, yet it leads to less performant confidentiality-preserving data analytics.

Lastly, existing systems have mostly considered computations over encrypted data in a database or a distributed batch processing setting but not in a stream processing setting. Stream processing poses unique challenges when computing over encrypted data. Furthermore, stream processing applications often use data generated by memory constrained and computationally constrained Internet of Things (IoT) devices. A straightforward application of PHE and PPE schemes on such end devices is unlikely to be practical.

1.3 Contributions

In this thesis we address the aforementioned challenges and investigate ways of performing both batch and stream cloud-based data analytics in a practical and confidentiality-preserving manner.

We begin by introducing a set of extensions and optimizations to existing PHE and PPE schemes (Chapter 3) making these schemes more expressive, secure, and performant.

We then introduce a data type abstraction (Chapter 4) that captures properties of various PHE and PPE schemes and exposes a set of programming abstractions that can express data flow style computations that capture simple but fine-grained information relevant to security and performance. We then describe a compiler that leverages the above abstractions to substantially accelerate PHE- and PPE-based query execution. We also introduce a planner engine that utilizes trusted resources smartly to improve the performance of query execution.

Recognizing that a major source of overhead when executing queries over encrypted data is due to the use of asymmetric PHE schemes to carry out arithmetic operations, we introduce two novel symmetric PHE schemes. Our proposed schemes are more performant than their asymmetric counterparts and they retain the full range of homomorphic operations that asymmetric schemes support while offering the same level of security (semantic security, (IND-CPA)).

Finally, we provide implementations to existing PHE and PPE schemes designed for use in resource-constrained IoT devices and present a secure stream abstraction through which confidentiality-preserving stream analytics can be expressed. We demonstrate the practicality of computations over encrypted data in applications that use small IoT devices as the end devices of a larger stream processing system.

1.4 Thesis Statement

In this thesis we introduce methodologies for practical, confidentiality-preserving in-cloud data analytics over encrypted data. *We propose a new data type abstraction that captures security constraints and the structure of input data and allows for more optimized batch processing analytics, two symmetric PHE schemes that replace slower state-of-the-art asymmetric PHE schemes, and a set of techniques that make PHE and PPE practical on resource-constrained IoT devices and in stream processing analytics.*

1.5 Thesis Roadmap

This thesis contains eight chapters. Chapter 2 presents background information on PHE, PPE as well as on batch and stream processing data analytics frameworks. Chapter 3 introduces a set of extensions and optimizations to existing PHE and PPE schemes to make them more performant and more expressive. Chapter 4 proposes the Secure Data Types (SDTs) abstraction and a distributed data analytics system that leverages this abstraction to improve the security and performance of data analytics over encrypted data. Chapter 5 introduces two novel symmetric PHE schemes and a distributed data analytics system that uses these schemes to achieve efficient arithmetic operations over encrypted data. Chapter 6 demonstrates the practicality of PHE and PPE on IoT devices and presents a stream data analytics system that preserves confidentiality. Chapter 7 concludes this thesis and presents plans for future work.

2. BACKGROUND

To preserve confidentiality of sensitive data while utilizing the compute capabilities of the cloud, we utilize security mechanisms such as Partially Homomorphic Encryption (PHE), Property-Preserving Encryption (PPE), and Trusted Execution Environments (TEEs). In this chapter we present background information about these mechanisms (as well as relevant background about Fully Homomorphic Encryption (FHE) for comparison) and put emphasis on the parts pertaining to subsequent chapters of this thesis. In addition, we present background information about distributed batch processing frameworks and distributed stream processing frameworks, such as Apache Spark and Apache Storm.

2.1 Fully Homomorphic Encryption

FHE allows arbitrary operations to be performed directly over encrypted data and therefore preserves the confidentiality of data throughout computations. Gentry introduced the approach of FHE and a first scheme [3] that provably achieves it. Even though FHE has been becoming more practical [4], it still exhibits cost which is prohibitive for many complex computations. In this thesis we steer away from FHE schemes to keep execution overheads low. We instead focus on using PHE, PPE, and TEEs in novel ways, to improve expressiveness, performance, and security of data analytics.

2.2 Partially Homomorphic Encryption

Unlike FHE which allows arbitrary computations over encrypted data, PHE allows computations over encrypted data with respect to specific operations. More formally, a cryptosystem is said to be *partially homomorphic* with respect to certain operations if it allows computations consisting in such operations on encrypted data. If $E(x)$ and $D(x)$ denote the encryption and decryption functions for input data x respectively (omitting keys for simplicity), then an encryption scheme is said to be homomorphic with respect to operation ϕ if $\exists$ operation ψ such that

$$D(E(x_1)\ \psi\ E(x_2)) = x_1\ \phi\ x_2 \tag{2.1}$$

An encryption scheme is said to be an Additive Homomorphic Encryption (AHE) scheme when ϕ is addition "+". Examples of AHE schemes include the Paillier [5], Benaloh [6], Okamoto–Uchiyama [7], Naccache-Stern [8], and Damgård-Jurik [9] cryptosystems. Similarly, an encryption scheme is said to be a Multiplicative Homomorphic Encryption (MHE) scheme when ϕ is multiplication "$\times$". Examples of MHE schemes include the ElGamal [10], and the unpadded RSA [11] cryptosystems. Other PHE schemes include the Goldwasser-Micali cryptosystem [12] which is homomorphic with respect to exclusive-OR (addition modulo 2) and the Sander-Young-Yung [13] cryptosystem which is homomorphic with respect to AND (multiplication modulo 2). Another noteworthy PHE scheme is the Boneh-Goh-Nissim cryptosystem [14] which allows any number of additions over encrypted data and a single multiplication over encrypted data.

Example: Paillier cryptosystem The Paillier cryptosystem [5] is a probabilistic asymmetric AHE scheme the security of which is based on the decisional composite residuosity assumption. Given a security parameter ϵ (at the time of this writing, NIST recommends that ϵ is at least 2048 for "factoring modulus" related assumptions [15]), we generate the public and private keys of Paillier by choosing two distinct prime numbers p and q of size $\frac{\epsilon}{2}$ bits each and setting the modulus N to $N = pq$. Then we set $\lambda = lcm(p-1, q-1)$ where lcm denotes the least common multiplier function. Lastly, we choose a generator $g \in \mathbb{Z}^*_{N^2}$ such that $gcd(L(g^\lambda \bmod N^2), N) = 1$ where gcd denotes the greatest common divisor function and L is defined as:

$$L(x) := \frac{x-1}{N}$$

We set the public key to $pk = (g, N)$ and the corresponding private key to $sk = (g, N, \lambda)$. Given a public key pk, and a plaintext message m, the encryption function of Paillier is defined as follows:

$$E_{pail}(pk, m) := g^m r^N \bmod N^2 \tag{2.2}$$

where r is a uniform random value. Given a private key sk, and a ciphertext value c the decryption function of Paillier is defined as follows:

$$D_{pail}(sk, c) := \frac{L(c^\lambda \bmod N^2)}{L(g^\lambda \bmod N^2)} \bmod N \tag{2.3}$$

Given two Paillier ciphertexts $c_1 = g^{m_1} r_1^N \bmod N^2$ and $c_2 = g^{m_2} r_2^N \bmod N^2$, homomorphic addition ($\mathsf{add}$) can be achieved by:

$$\begin{aligned} \mathsf{add}(pk, c_1, c_2) &:= c_1 c_2 \bmod N^2 \\ &= g^{m_1} r_1^N g^{m_2} r_2^N \bmod N^2 \\ &= g^{m_1+m_2} (r_1 r_2)^N \bmod N^2 \end{aligned} \tag{2.4}$$

which is a valid encryption of $m_1 + m_2$. Note that the above homomorphic operation requires the public key, pk, but not the private key, sk.

Example: ElGamal cryptosystem The ElGamal cryptosystem [10] is a probabilistic asymmetric MHE scheme the security of which is based on the decisional diffie hellman assumption. Given a security parameter ϵ (at the time of this writing, NIST recommends that ϵ is at least 2048 for "discrete logarithm" related assumptions), we select a generator g of an efficient cyclic group $\mathbb{G}$ of order N where N has a bit-length of ϵ. We then select a random integer $x \in \mathbb{Z}_N$ and set $h = g^x$. We set the public key to $pk = (\mathbb{G}, N, g, h)$ and the corresponding private key to $sk = (\mathbb{G}, N, g, x)$. Given a public key pk and a plaintext message m, the encryption function of ElGamal is defined as follows:

$$\begin{aligned} E_{elg}(pk, m) &:= (c_1, c_2) \\ &= (g^r \bmod N, m h^r \bmod N) \end{aligned} \tag{2.5}$$

where r is a uniform random value. Given a private key sk and a ciphertext value $c = (c_1, c_2)$, the decryption function of ElGamal is defined as follows:

$$\begin{aligned} D_{elg}(sk, c) :=& \frac{c_2}{c_1^x} \bmod N \\ =& \frac{mh^r}{g^{rx}} \bmod N \\ =& \frac{mg^{xr}}{g^{rx}} \bmod N \\ =& m \end{aligned} \tag{2.6}$$

Given two ElGamal ciphertexts $c_1 = (c_1^1, c_2^1)$ and $c_2 = (c_1^2, c_2^2)$, homomorphic multiplication (mul) can be achieved by:

$$\begin{aligned} \mathsf{mul}(pk, c_1, c_2) :=& (c_1^1 c_1^2, c_2^1 c_2^2) \\ =& (g^{r_1} g^{r_2}, m_1 h^{r_1} m_2 h^{r_2}) \\ =& (g^{r_1+r_2}, m_1 m_2 h^{r_1+r_2}) \end{aligned} \tag{2.7}$$

which is a valid encryption of $m_1 m_2$. Note that, just like with Paillier homomorphic addition, the above homomorphic multiplication requires the public key, pk, but not the private key, sk.

Secondary homomorphic operations The "primary" operations supported by each encryption scheme require that their operands are both encrypted under the same encryption scheme. In addition to these operations, some cryptosystems support "secondary" operations as long as one of the operands is in plaintext form. The Paillier cryptosystem, for example, primarily supports addition between two encrypted values as shown in Equation 2.4, but beyond this operation, Paillier also supports addition between an encrypted value, c_1, and a plaintext value, m_2 (adp: $\mathsf{add\ plaintext}$):

$$\begin{aligned} \mathsf{adp}(pk, c_1, m_2) :=& c_1 g^{m_2} \bmod N^2 \\ =& g^{m_1} r_1^N g^{m_2} \bmod N^2 \\ =& g^{m_1+m_2} r_1^N \bmod N^2 \end{aligned} \tag{2.8}$$

which is a valid encryption of $m_1 + m_2$. Paillier also supports multiplication between an encrypted value, c_1, and a plaintext value, m_2 (mlp: **mul**tiply **p**laintext):

$$\begin{aligned} \mathsf{mlp}(pk, c_1, m_2) :=&\ c_1^{m_2} \bmod N^2 \\ =&\ (g^{m_1} r_1^N)^{m_2} \bmod N^2 \\ =&\ g^{m_1 m_2} (r_1^{m_2})^N \bmod N^2 \end{aligned} \tag{2.9}$$

which is a valid encryption of $m_1 m_2$. Homomorphic subtraction (sub) can, therefore, be achieved by performing an addition between the first operand and the additive inverse (by performing a multiplication by -1) of the second operand:

$$\begin{aligned} \mathsf{sub}(pk, c_1, c_2) :=&\ \mathsf{add}(pk, c_1, \mathsf{mlp}(pk, c_2, -1)) \\ =&\ g^{m_1} r_1^N g^{-m_2} (r_2^{-1})^N \bmod N^2 \\ =&\ g^{m_1 - m_2} (r_1 r_2^{-1})^N \bmod N^2 \end{aligned} \tag{2.10}$$

which is a valid encryption of $m_1 - m_2$. Similar to Paillier, the ElGamal cryptosystem supports secondary homomorphic operations. Specifically, ElGamal supports multiplication and division (multiplication with the multiplicative inverse) between two encrypted values and multiplication/exponentiation between an encrypted and a plaintext value.

2.3 Property-Preserving Encryption

Another category of cryptosystems that allows computations over encrypted data is Property-Preserving Encryption (PPE). As the name suggests, PPE schemes preserve some property of the underlying plaintext which in turn can be used to perform operations over encrypted values.

For example, deterministic (DET) encryption schemes such as AES in ECB mode or AES in CMC mode with a zero initialization vector [16] preserve plaintext uniqueness and hence enable equality comparisons over encrypted data. Other PPE schemes preserve the order of underlying plaintext values and enable order comparisons such as "<" and ">" over encrypted data. These schemes are called Order-Preserving

Encryption (OPE) schemes, and an example of a such scheme is the order preserving symmetric encryption scheme introduced by Boldyreava et al [17] which also happens to be a deterministic scheme. Lastly, some schemes enable text searches over encrypted data by applying "pattern matching", enabled by searchable encryption (SRCH) schemes, such as the SWP scheme [18].

In the past, PPE schemes have been criticized for having low-security guarantees [19–22], but recent work has introduced new PPE schemes that provide semantic security [23].

2.4 Trusted Execution Environments

Trusted Execution Environments (TEEs) have recently gained traction for performing secure computations in the cloud. TEEs are secure areas inside a CPU that runs separately of the operating system, in an isolated environment and ensure that the code and data loaded in their environment remain confidential and integrity protected. These security guarantees are enabled through both software and hardware extensions in recent CPUs.

Intel Software Guard eXtensions Intel SGX [24] is a popular TEE supported in the latest iterations of Intel's CPUs. SGX is a set of x86-64 ISA extensions that enable the creation of isolated areas of execution, called enclaves. Within enclaves, operations can be performed on sensitive data in a way protected against accesses from the outside environment, including those from the operating system and hypervisor, since the CPU oversees accesses to the enclave memory. Attempts to access the memory assigned to an enclave are rejected by the CPU, and enclave cache lines are encrypted using authenticated encryption before leaving the CPU package, e.g., before written to the main memory. This security model is stronger than earlier security models such as ones based on trusted hypervisors because it entails a smaller TCB. More specifically, the TCB of SGX is limited to the CPU package and the user code that runs inside the enclave. Under this security model, the operating system and the hypervisor don't need to be trusted but also a wide range of hardware attacks, such as memory probing, are covered. Importantly, SGX supports sealing which allows for enclave data to be stored to an untrusted memory after the data has been encrypted using a CPU and enclave-identity specific key. SGX also supports local attestation which allows an enclave to prove its identity to another enclave on the same machine, after which a secure channel can be established between the enclaves, via which the enclaves can securely communicate. Furthermore, SGX supports remote attestation through which an enclave can prove its identity to a remote party.

2.5 Distributed Batch Processing

Distributed batch processing systems utilize multiple nodes in a cloud to enable the processing of high-volume, repetitive data jobs with little or no user interaction. Distributed batch processing systems are critical in most organizations because they allow efficient execution of complex queries which are used for the organization's decision support and to give answers to critical business questions.

Apache Spark and Spark SQL Apache Spark is an example of a general-purpose cloud computing engine which uses distributed collections called resilient distributed datasets (RDDs) that can be manipulated through operations like map, filter, and reduce. RDDs are fault-tolerant, in that the system can recover lost data using the lineage graph of the RDDs to re-compute the lost data. They can also be explicitly cached in memory to speed up computation. Spark also includes the DataFrame primitive which is equivalent to a table in a relational database. DataFrames keep track of their schema and support various relational operations that lead to more optimized execution. Using DataFrames, Spark SQL extends Spark with a declarative DataFrame API to allow relational processing, offering benefits such as automatic optimization, and letting users write complex pipelines that mix relational and complex analytics. Spark SQL includes a highly extensible optimizer, called Catalyst, that makes it easy to add composable rules, control code generation, and define extension points.

2.6 Distributed Stream Processing

Distributed stream processing systems are designed to support applications that require timely processing of high-volume data streams. Stream processing gives organizations the ability to quickly process large amounts of data from multiple sources, in real-time. Oftentimes, stream data is generated through Internet of Things (IoT) devices which are resource-constrained.

Apache Storm Apache Storm is an open-source distributed real-time stream processing system used for processing data streams. Apache Storm processes unbounded streams of data in a reliable manner and delivers high-end applications. Storm uses the stream abstractions of spouts, bolts, and topologies. Spouts are input data streams than need to be processed. Bolts are compute units that process input streams and produce output streams. Topologies are graphs that capture the data flow with bolts representing nodes on the graph and spouts representing edges.

3. PRACTICAL COMPUTATIONS OVER ENCRYPTED DATA

Many of the systems described in this thesis rely on Partially Homomorphic Encryption (PHE) and Property-Preserving Encryption (PPE) schemes to preserve the confidentiality of sensitive information. Despite the fact that these schemes are more practical than Fully Homomorphic Encryption (FHE) schemes, a straightforward application of PHE and PPE for confidential data analytics is unlikely to be practical. In this chapter we introduce a set of extensions to the schemes used to improve their expressiveness, hence providing better support for computations over encrypted data. We then explore intricate characteristics of these schemes and leverage them to enable the application of a number of optimizations. These optimizations make the schemes more performant in terms of the cost of their encryption and decryption functions as well as the cost of computations over encrypted data they support. The extensions and optimizations introduced in this chapter are then used by systems described in subsequent chapters to make data analytics more practical and more expressive.

3.1 Synopsis

We introduce several extensions to both PHE and PPE schemes to improve the expressiveness of operations they support and improve their performance. Here, we first give an overview of these extensions and the purpose they serve and we describe each in more detail in subsequent sections.

Support for negative numbers: By default, PHE schemes that allow arithmetic operations over encrypted data do not support negative numbers. We extend Additive Homomorphic Encryption (AHE) and Multiplicative Homomorphic

Encryption (MHE) schemes to support encryption, decryption, and homomorphic operations that involve negative numbers.

Secure search constructions: String search can vary from simple `exists` requests that check whether a token exists in a string to complex pattern matching. We extend existing searchable encryption (SRCH) schemes to offer support to a wide range of search operations over encrypted data.

Ciphertext packing: We extend AHE and MHE schemes to allow packing multiple plaintext values into a single ciphertext, thereby, amortizing the cost of encryption and decryption functions as well as the cost of homomorphic operations they support.

PRN pre-computation: We pre-compute and store a small number of PRNs to speed up encryption of probabilistic encryption schemes.

Encryption caching: We enable caching of ciphertexts for deterministic encryption schemes to speed up encryption.

Speculative encryption: We speculatively encrypt a small number of plaintext values that are likely to be needed to speed up future encryption requests.

Format-preserving encryption: Where possible, we make use of deterministic schemes that preserve the format and size of the plaintext data when encrypting to minimize the ciphertext size overhead and thereby, improve the performance of operations over encrypted data.

3.2 Support for Negative Numbers

AHE schemes such as Paillier [5] and MHE schemes such as ElGamal [10] are used in PHE-based systems to carry out arithmetic operations homomorphically over encrypted data. By default, these encryption schemes, as well as existing PHE-based systems that use these schemes [25–28], do not support operations that involve negative numbers. Attempting to encrypt or decrypt a negative number under such a scheme will lead to unexpected results. In addition, performing homomorphic operations on positive numbers that produce negative numbers, e.g., homomorphic subtraction, and then decrypting the result will also lead to incorrect results.

To allow these schemes to support negative numbers we introduce alternative implementations for the encryption and decryption functions for both AHE and MHE schemes. To achieve this, we leverage the fact that most asymmetric homomorphic encryption schemes, including Paillier and ElGamal, operate on large plaintext and ciphertext spaces. This is because the security of these cryptosystems depends on problems that are hard to solve for large numbers such as the decisional composite residuosity assumption for Paillier and the decisional Diffie-Hellman assumption for ElGamal. At the time of this writing, technical standard institutions such as NIST, recommend that cryptosystems based on these mathematical problems use a security parameter ϵ of at least 2048 [15]. This results in a 2048-bit modulus, N, which corresponds to a 2048-bit plaintext space (see Section 2.2 for more details). Furthermore, since both Paillier and ElGamal have an expansion factor of 2, their resulting ciphertext space is 4096 bits long. In comparison, the actual space required by applications is much smaller, e.g., 32 bits for `int` values or 64 bits for `long` values.

We define a configurable parameter $\delta \geq 1$ that divides the plaintext space into two distinct parts. We treat values in the range $[0, \lfloor \frac{N}{\delta} \rfloor)$ as positive numbers and values in the range $[\lfloor \frac{N}{\delta} \rfloor, N)$ as negative numbers. When $\delta = 1$ all values are treated as positive. By default, we set $\delta = 2$ which divides the plaintext space in two roughly equal parts. For both Paillier and ElGamal we define encryption that can handle negative numbers as:

$$E^{neg}(pk, x) = \begin{cases} E(pk, x \bmod N) & \text{if } -\lceil N(1 - \frac{1}{\delta}) \rceil \leq x < \lfloor \frac{N}{\delta} \rfloor \\ \varnothing & \text{otherwise} \end{cases} \tag{3.1}$$

where $E(pk, x)$ denotes the original definition of encryption that does not support negative numbers, e.g., Equation 2.2 or Equation 2.5. Here, $x \bmod N$ always returns

Table 3.1.: Search constructions. NoDup: no duplicate tokens in ciphertext. Shuffle: token order is randomized

Constr.	**Parameters**		**Match**				
	NoDup	**Shuffle**	**Exists**	**Count**	**StartsWith**	**EndsWith**	**Pattern**
A	1	1	✓	✗	✗	✗	✗
B	1	0	✓	✗	✓	✗	✗
C	0	1	✓	✓	✗	✗	✗
D	0	0	✓	✓	✓	✓	✓

a non-negative value even for $x < 0$ and $\varnothing$ indicates an invalid ciphertext was returned because input x lies outside the supported range. For both Paillier and ElGamal we define decryption that can handle negative numbers as:

$$D^{neg}(sk, x) = ((D(sk, x) + \lceil N(1 - \frac{1}{\delta})\rceil) \bmod N) - \lceil N(1 - \frac{1}{\delta})\rceil$$

or equivalently, and avoiding expensive modulo computations:

$$D^{neg}(sk, x) = \begin{cases} D(sk, x) - N & \text{if } D(sk, x) \geq \lfloor \frac{N}{\delta} \rfloor \\ D(sk, x) & \text{otherwise.} \end{cases} \tag{3.2}$$

where $D(sk, x)$ denotes the original definition of decryption that does not support negative numbers, e.g., Equation 2.3 or Equation 2.6. With the encryption and decryption functions described above, all homomorphic operations can remain unchanged and negative numbers will be supported.

3.3 Secure Search Constructions

SRCH schemes such as the SWP cryptosystem [18] are used to perform string search over encrypted text. The SWP cryptosystem works by first splitting strings into tokens, usually using whitespace as delimiter, and then encrypting each token separately before constructing the ciphertext that contains all encrypted tokens. We extend this cryptosystem to accept two boolean parameters that control what kind of match operations can be performed on ciphertexts and also affect the amount of information that a ciphertext reveals. The first parameter, *NoDup*, indicates that the resulting ciphertext should not contain duplicate tokens. When this parameter is enabled, all duplicate tokens are removed from the plaintext before encrypting. The second parameter, *Shuffle*, indicates that the order of the tokens should be shuffled before encrypting. These parameters lead to four different search constructions as shown in Table 3.1. The most secure construction is construction A which removes duplicate tokens and randomizes the order of the remaining tokens before encrypting. This construction offers near random security (leaks only the number of encrypted tokens within the text) but only allows to check whether a token exists in the encrypted text. Construction B removes duplicates but does not shuffle tokens before encryption. This retains the order of tokens after duplicates have been removed and allows for search operations involving the order of tokens such as the StartsWith operation. Because when removing duplicates only the first instance of a token is retained, EndsWith is not supported in construction B. Search construction C randomizes the order of the tokens but does not remove duplicate tokens. This retains the number of instances of each token within the encrypted text and thus enables the use of count. Lastly, construction D neither removes duplicate tokens nor randomizes their order, allowing matching on any pattern including any number of tokens and involving the regex operators ".*" (any text) and "|" (or).

3.4 Ciphertext Packing

As mentioned above, the plaintext space of asymmetric encryption schemes such as Paillier and ElGamal is significantly larger than the space needed to represent numbers in data analytics applications. For example, encrypting a 32-bit integer value under Paillier or ElGamal with security parameter $\epsilon = 2048$ will produce a 4096-bit ciphertext which has a 128× ciphertext size overhead. To reduce ciphertext size overhead, we use a technique introduced by Ge et al. [29] to pack multiple plaintext values into a single ciphertext. Ciphertext packing works by concatenating multiple messages into a single plaintext value before encrypting. For example, values $a_1, a_2, ..., a_n$ can be concatenated into $a_1 \circ a_2 \circ ... \circ a_n$ before being encrypted, where $\circ$ indicates bit-string concatenation. As homomorphic operations are performed on ciphertexts, the operations are carried out on the underlying packed values separately.

A potential issue when carrying out operations over ciphertexts that pack multiple values are overflows, where the result of one set of packed values overflows into the preceding one, which would lead to incorrect results. Ge et al. demonstrate packing for AHE schemes and solve overflows by using multiple groups and keeping partial sums per group, careful to only pack values in a way that cannot overflow. This approach works well in a database setting assumed by Ge et al. but does not work in a data analytics setting, including steam processing because values are generated dynamically and cannot be known beforehand. Instead, in this work we introduce another approach where before each packed value we include a series P of 0 bits, so that in case of overflow, the preceding value will not be affected: $P \circ a_1 \circ P \circ a_2 \circ ... \circ P \circ a_n$. In the next paragraphs we demonstrate a novel approach of calculating padding bits P for AHE and MHE schemes, we demonstrate a novel way of ciphertext packing for MHE schemes, and introduce post-encryption packing for AHE schemes which allows us to pack values together after they have been encrypted.

AHE packing Homomorphic addition can be carried out when ciphertexts contain packed values because arithmetically $(a_1 \circ ... \circ a_n) + (b_1 \circ ... \circ b_n) = (a_1 + b_1) \circ ... \circ (a_n + b_n)$. To avoid overflows we calculate the total number of bits, T, that need to be allocated for each packed item and the number of items that can be packed in a single plaintext, I, before encrypting as follows:

$$\begin{aligned} T &= P + M \\ &= \lfloor log_2(R(2^M - 1)) \rfloor + 1, \end{aligned} \tag{3.3}$$

$$\begin{aligned} I &= \left\lfloor \frac{\epsilon}{T} \right\rfloor \\ &= \left\lfloor \frac{\epsilon}{\lfloor log_2(R(2^M-1)) \rfloor + 1} \right\rfloor \end{aligned} \tag{3.4}$$

In the above equations P is the number of padding bits needed to capture overflows, M is the size of each message in bits (e.g., $M = 32$ for `int` values), R is the maximum number of tuples containing packed ciphertexts that can be aggregated before padding bits are exceeded, and ϵ is the security parameter of the scheme which, as discussed in Section 2.2, dictates the size of the plaintext space (e.g., when $\epsilon = 2048$ the modulus N is a 2048 bits long value). Here, $2^M - 1$ is the largest possible number that can be represented in M bits, $R(2^M - 1)$ is the largest possible number after R additions, and $\lfloor log_2(x) \rfloor + 1$ is the number of bits needed to represent number x. By default we set $R = 2^{30}$ which means when packing 32-bit integers into a single 2048-bit plaintext we can fit 33 items before encrypting and can perform over 1 billion operations without exceeding the allotted padding bits. Using T as calculated above, we define encryption of a vector containing n messages $\vec{x} = (x_1, x_2, ..., x_n)$ that applies AHE packing as:

$$\begin{aligned} E_{ahe}^{pack}(pk, \vec{x}) &:= E(pk, pack(T, \vec{x})) \\ &= E(pk, \sum_{i=1}^{n} 2^{(i-1)T} x_i) \end{aligned} \tag{3.5}$$

where $E(pk, x)$ denotes the original definition of the encryption function that accepts a single integer value, e.g., Equation 2.2. We define decryption for AHE schemes that applies unpacking as:

$$\begin{aligned} D_{ahe}^{pack}(sk, x) &:= unpack(T, D(sk, x)) \\ &= (x_1, x_2, ..., x_n) \\ &= \vec{x} \end{aligned} \tag{3.6}$$

where $D(sk, x)$ denotes the original definition of the decryption function that returns a single integer value, e.g., Equation 2.3 with the *unpack* function being the inverse of the *pack* function ($unpack(T, pack(T, \vec{x})) = \vec{x}$) that returns a vector of messages. With the encryption and decryption functions defined above, AHE operations over packed ciphertexts can be performed without changing the definitions of the homomorphic operations.

MHE packing We also allow packing for MHE. Packing for MHE is more limited because in multiplication each packed item of a ciphertext is multiplied with all packed items of the other ciphertext. We, therefore, fix the number of items packed in a plaintext to 2 ($I = 2$). Now, arithmetically we have $(a_1 \circ a_2) \times (b_1 \circ b_2) = (a_1 \times b_1) \circ (a_1 \times b_2 + a_2 \times b_1) \circ (a_2 \times b_2)$ which includes the intermediate term $(a_1 \times b_2 + a_2 \times b_1)$. By ignoring the intermediate term after decryption, we get the required $(a_1 \times b_1) \circ (a_2 \times b_2)$. We compute the total number of bits, T, allocated for each packed item as follows:

$$\begin{aligned} T = & \ P + M \\ = & \ \lfloor log_2(2^M - 1)^R \rfloor + 1 \\ \approx & \ log_2(2^M)^R \\ = & \ MR, \end{aligned} \tag{3.7}$$

where $(2^M - 1)^R$ is the maximum possible number after R items of M bit-length are multiplied together. We calculate the number of tuples, R, that can be aggregated before overflows exceed the padding bits as follows:

$$\begin{aligned} R = & \left\lfloor \frac{\epsilon}{T} \right\rfloor - 1 \\ \approx & \left\lfloor \frac{\epsilon}{MR} \right\rfloor - 1 \Rightarrow \\ R + 1 \approx & \ \frac{\epsilon}{MR} \Rightarrow \\ R(R+1) \approx & \ \frac{\epsilon}{M} \Rightarrow \\ R \approx & \left\lfloor \frac{\sqrt{M^2 + 4M\epsilon} - M}{2M} \right\rfloor \end{aligned} \tag{3.8}$$

In the above equation, -1 accounts for the intermediate terms discussed above. By replacing T with the approximation MR and solving for the positive root of the quadratic equation we get the final term. The above equations show that when packing 32-bit integers into a plaintext space of 2048 bits a total of $R = 7$ packed ciphertexts can be multiplied together before an overflow exceeds the padding bits.

To continue performing multiplications after this, the packed ciphertext needs to be refreshed by being re-encrypted using a trusted node. This suggests that MHE packing is mostly useful in applications that need to perform multiplications infrequently or when there are frequent uses of a trusted service(see Section 4.5) or frequent key changes (see Section 6.5) as part of which the packed ciphertext can be refreshed. We note that every homomorphic multiplication operation generates an additional intermediate term in the ciphertext. In order to ignore these terms after decrypting we extend the ciphertext of our MHE scheme to include a counter indicating how many multiplications have been performed to generate that ciphertext. When decrypting, this counter is used to identify how many intermediate terms need to be ignored to get the correct result. Concretely we define encryption of MHE schemes that apply packing of a vector $\vec{x} = (x_1, x_2)$ (as noted above, we pack exactly two items before encryption) as:

$$\begin{aligned} E_{mhe}^{pack}(pk, \vec{x}) &:= (c, ctr) \\ &= (E(pk, pack(T, \vec{x})), 0) \\ &= (E(pk, \sum_{i=1}^{n} 2^{(i-1)T} x_i), 0) \end{aligned} \tag{3.9}$$

where $E(pk, x)$ denotes the original definition of the encryption function, e.g., Equation 2.5 and *ctr* is the counter of how many intermediate terms the ciphertext contains that is included in the ciphertext in plaintext form, and is initialy set to 0. We define decryption that applies MHE unpacking as:

$$\begin{aligned} D_{mhe}^{pack}(sk, x) &:= trunc(unpack(T, D(sk, c)), ctr) \\ &= trunc((x_1, x_2, ..., x_n), ctr) \\ &= (x_1, x_{ctr+1}) \\ &= \vec{x} \end{aligned} \tag{3.10}$$

In the above encryption and decryption functions, *pack* and *unpack* functions are the same as the ones used in Equation 3.5 and Equation 3.6. The *trunc* function takes as input a vector of values $\vec{x}$ and the *ctr* value and returns a vector containing the first and the (ctr+1)th item in the given vector, by ignoring *ctr* intermediate terms. We also define homomorphic multiplication that performs multiplication of two packed ciphertext values $x_1 = (c_1, ctr_1)$ and $x_2 = (c_2, ctr_2)$ as:

$$\mathsf{mul}^{pack}(pk, x_1, x_2) := \ (\mathsf{mul}(pk, c_1, c_2), ctr_1 + 1) \tag{3.11}$$

The above equation shows that the original definition of mul is used on the c_1 and c_2 components and the counter of the first ciphertext, ctr_1, is incremented and included in the resulting ciphertext. The second ciphertext x_1 is expected to contain no intermediate terms, i.e., $ctr_2 = 0$. Other homomorphic operations that MHE schemes support are similarly adjusted.

Post-encryption packing The packing techniques described above require that multiple messages are packed together into a single plaintext before encrypted. It is oftentimes not possible to pack messages before they are encrypted. For example, these messages might be generated over time and not available at the moment of encryption. It is still beneficial to pack values after they have been encrypted through post-encryption packing, which allows multiple ciphertext to be packed together and reduces ciphertext size and decryption times. We support post-encryption packing for AHE schemes. To pack a vector of ciphertexts $\vec{c} = (c_1, c_2, ..., c_n)$ into a single ciphertext, we first homomorphically shift the ciphertexts appropriately and then homomorphically add them together:

$$pack^{post}(pk, T, \vec{c}) := \sum_{i=1}^{n} \mathsf{mlp}(pk, c_i, 2^{(i-1)T}) \tag{3.12}$$

In the above equation, mlp denotes homomorphic multiplication between a ciphertext and a plaintext value (see Equation 2.9), T indicates the total number of bits required per packed item, including padding bits, as described in Equation 3.3, $n \leq I$ is the number of ciphertexts in the given vector, $\vec{c}$, that need to be packed together, and $\sum$ denotes homomorphic summation.

3.5 PRN Pre-computation

Encryption functions of probabilistic (PHE) schemes such as Paillier [5] and ElGamal [10] work by first generating a large pseudorandom number (PRN) and then carrying out computations involving the PRN and the plaintext value. As shown in Equation 2.2 for Paillier and Equation 2.5 for ElGamal, a random value r is firstly generated before any other computation occurs. Generating this PRN is oftentimes the most expensive operation of the encryption function, but luckily, PRNs can be generated independently of encryption requests and stored for later use. Furthermore, the bulk of the computations required to perform encryption can be done before having access to the message m. To utilize PRN pre-computation and improve the performance of encryption, we divide computations involved in encryption in two parts. We first perform the pre-computation step, P, that computes all components of the ciphertext that don't require access to the message m and then define a new encryption function E^{pre} that utilizes the outcome of pre-computation to perform fast encryption. For Paillier these functions are defined as follows:

$$\begin{aligned} P_{pail}(pk) :=&\ p \\ =&\ r^N \bmod N^2 \end{aligned} \tag{3.13}$$

$$E^{pre}_{pail}(pk, m) := g^m p \bmod N^2 \tag{3.14}$$

where r is a uniform random value and p is the result of pre-computation. We note that P_{pail} does not require access to the plaintext message m and, therefore, can be computed and stored for later use. With pre-computation, the resulting encryption function E^{pre}_{pail} requires only a modular exponentiation and a modular multiplication, which is much faster than the original definition of Paillier encryption in Equation 2.2. Similarly, for ElGamal these functions are defined as follows:

$$\begin{aligned} P_{elg}(pk) :=&\ (p_1, p_2) \\ =&\ (g^r \bmod N, h^r \bmod N) \end{aligned} \tag{3.15}$$

$$\begin{aligned} E_{elg}^{pre}(pk, m) :=& \ (c_1, c_2) \\ =& \ (p_1, mp_2 \bmod N) \end{aligned} \tag{3.16}$$

where r is a uniform random value and p_1, p_2 are the results of pre-computation. With pre-computation, the resulting encryption function E_{elg}^{pre} requires only a modular multiplication, which is much faster than the original definition of ElGamal encryption in Equation 2.5.

Importantly, the PRN pre-computation optimization does not conflict with other extensions or optimizations introduced above, e.g., support for negative numbers, and ciphertext packing. Therefore, PRN pre-computation can be combined with other optimizations to further improve performance.

3.6 Encryption Caching

Deterministic encryption schemes such as deterministic (DET) schemes and Order-Preserving Encryption (OPE) schemes produce the same ciphertext for a fixed plaintext value. To speed up encryption and decryption of deterministic encryption schemes, we use a map that stores a fixed number of plaintext-ciphertext value pairs and we impose an LRU policy to remove pairs once the map is full. We extend the definitions of encryption and decryption functions for DET and OPE schemes to first search this map to see if the plaintext-to-ciphertext key-value pair exists, and if so return the corresponding value. Otherwise, we perform encryption/decryption as usual.

3.7 Speculative Encryption

To further reduce encryption time overhead, we use speculative encryption, by attempting to predict what values will need to be encrypted next. Similar to the encryption caching optimization, we encrypt a small number of plaintext-ciphertext value pairs proactively and store them in a fixed-sized map with an LRU eviction policy. Speculative encryption is mostly useful in scenarios where the range of possible values is small, or for low entropy values. Importantly though, speculative encryption can be used for both deterministic and probabilistic encryption schemes.

3.8 Format-Preserving Encryption

Oftentimes, data analytics queries include equality comparisons involving values of a fixed format such as dates, timestamps, or phone numbers. To perform equality comparisons these values need to be encrypted under a deterministic scheme. Naïvely encrypting fixed-format values under a deterministic scheme such as AES will result in a ciphertext which is at least the size of the block of the cryptosystem used. To reduce ciphertext overhead, we employ format-preserving encryption. Format-preserving encryption generates n-bits of ciphertext for n-bits of input plaintext as long as the plaintext is smaller than the block size of the format-preserving encryption scheme used, e.g., 128 bits. This optimization allows us to significantly reduce the ciphertext size overhead for values that need to be encrypted under a deterministic scheme.

4. CONFIDENTIAL DATA ANALYTICS USING SECURE DATA TYPES

As public clouds become more popular, there is a need to allow cloud-based data analytics that preserve confidentiality of sensitive information and have practical overhead. In this chapter we describe our novel abstraction of *Secure Data Types (SDTs)* which can be used to conveniently denote constraints relevant to security as well as fine-grain data-type information about the input data. These abstractions are then leveraged by novel *compilation techniques* implemented in our system, `Cuttlefish`, to execute 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 (PHE) and Property-Preserving Encryption (PPE) schemes to perform operations securely, resorting however to *client-side completion*, *remote re-encryption*, or *secure hardware-based re-encryption* based on Intel SGX when available and uses a novel *planner engine* to decide the most efficient way of utilizing these trusted resources.

4.1 Overview

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. [30, 31]); and corresponding explicit regulations in certain sectors (e.g., healthcare and finance [32]).

4.1.1 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, and (2) hardware-based trusted computing solutions such as Intel Software Guard eXtensions (SGX) [24].

With the first approach, data is encrypted with specific cryptosystems 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 [3]. These support arbitrary computations on ciphertexts, but incur up to $10^9\times$ slowdowns [33] compared to executing computations on plaintext, thus annulling the benefits of computing in a cloud. Despite advancements in this area [4], FHE schemes still incur high overheads for complex computations and as a consequence they have not gained a great momentum in practice yet.

Instead, a lot of pragmatic research has been focused on using PHE and PPE schemes, which allow computations over encrypted data with respect to specific operations as opposed to arbitrary computations, but are more practical (see Chapter 2 for background details on PHE and PPE and Chapter 3 for a set of techniques making PHE and PPE more expressive and practical). Several systems (e.g., [25–27, 34–38]) have been proposed based on PHE and PPE. 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 an Additive Homomorphic Encryption (AHE) scheme; 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 a Multiplicative Homomorphic Encryption (MHE). Existing PHE-based systems handle such limitations in one of the following three ways: (i) *abort* by rejecting a corresponding query (e.g., [25, 35]); (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., [26,34,38]); or (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., [27,28,37]). 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 [39, 40]. 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 [41, 42]. 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 expressiveness 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 [25,26,34], MapReduce [35], PigLatin [27,36], Spark [38]), 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 sub-optimal performance but also confounds security guarantees of different cryptosystems 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.

4.1.2 Contributions

In this work 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.

The `Cuttlefish` 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.

Table 4.1 shows how `Cuttlefish` compares with various state-of-the-art systems. By definition, FHE [33] can express any computation but it exhibits high overhead for complex computations. Opaque [43] can express any computation within secure enclaves, but requires specialized hardware. CryptDB [25] and MrCrypt [35] have limited expressiveness since they cannot handle queries for which PHE and PPE alone can execute in full. Monomi [26] and Seabed [38] use split execution to execute part of the query in a trusted node. While this allows for any query to be executed, the

Table 4.1.: 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

System	**Expressiveness**	**Data Constraints**	**Specialized Hardware**
FHE [33]	Unlimited[1]	No	N/a
Opaque [43]	Unlimited[2]	No	Required
CryptDB [25]	Limited[3]	No	N/a
MrCrypt [35]	Limited[3]	No	N/a
Monomi [26]	Limited[4]	No	N/a
Seabed [38]	Limited[4]	No	N/a
Crypsis [36]	Limited[5]	No	N/a
Autocrypt [37]	Limited[5]	No	N/a
`Cuttlefish`	Unlimited[6]	Yes	Optional

expressiveness of the part of the query deployed in the cloud is limited. Crypsis [36] and Autocrypt [37] use re-encryption to convert from one PHE scheme to another when the current PHE scheme does not support the operation needed. Re-encryption alone limits expressiveness since it is computationally expensive but also because not all conversions are secure. In comparison, `Cuttlefish` uses a novel planner engine to overcome limitations of PHE without compromising performance, by more flexibly combining spit execution as well as re-encryption as explained in Section 4.5

Concretely, `Cuttlefish` makes the following contributions:

Programming abstractions: `Cuttlefish` introduces 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 cryptosystems such as secondary homomorphic properties. Several of our query rewriting techniques go against traditional compilation 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 [44] 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 [26] and Crypsis [36].

4.1.3 Roadmap

The rest of this chapter is structured as follows. Section 4.2 gives an overview of the design of `Cuttlefish` and Section 4.3 introduces SDTs. Section 4.4 presents our novel compilation techniques. Section 4.5 describes our planner engine and the heuristics for re-encryption. Section 4.6 discusses the implementation of `Cuttlefish` and Section 4.7 empirically evaluates `Cuttlefish`. Section 4.8 presents related work and Section 4.9 concludes with final remarks.

4.2 `Cuttlefish` Design

In this section, we first explain the threat model of our system and then provide information about the system design of `Cuttlefish`.

4.2.1 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, but 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 Trusted Execution Environment (TEE)-based nodes in the cloud such as Intel SGX [24].

`Cuttlefish` achieves this goal by utilizing a set of encryption schemes to encrypt sensitive data as shown in Table 4.2. The security guarantees offered by each of these encryption schemes vary from strong guarantees offered by probabilistic cryptosystems, to relatively weaker guarantees such as deterministic (DET) encryption schemes which reveal duplicate values, and Order-Preserving Encryption (OPE) schemes which reveal order of values.

Despite improvements [17, 47–51], DET and OPE schemes remain susceptible to inference attacks such as frequency analysis attacks [22, 52]. 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 cryptosystem that offers the required security guarantees. Furthermore, in Section 4.4.6 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 encryption schemes.

Table 4.2.: `Cuttlefish` cryptosystems and operations they support over encrypted data. "Type" column shows whether the cryptosystem is symmetric or asymmetric. "Operations" column lists operations between two encrypted values and "Secondary Operations" column lists operations between an encrypted value and a plaintext value. "$\div$" denotes multiplicative inverse and "$\wedge$" denotes exponentiation.

Cryptosystem	**Property**	**Type**	**Security**	**Operations**	**Secondary Operations**
AES (CBC mode)	RND	Sym.	Probabilistic	–	–
AES (CMC mode [16])	DET	Sym.	Deterministic	=	–
FNR [45]	DET	Sym.	Deterministic	=	–
Boldyreva et al. [46]	OPE	Sym.	Deterministic	$<, >$	–
SWP [18]	SRCH	Sym.	Probabilistic	`match`	–
ASHE [38]	AHE	Sym.	Probabilistic	$+$	–
Paillier [5]	AHE	Asym.	Probabilistic	$+, -$	$+, \times$
ElGamal [10]	MHE	Asym.	Probabilistic	$\times, \div$	$\times, \wedge$

4.2.2 `Cuttlefish` Overview

Figure 4.1 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 the untrusted cloud. Once the remote query completes, its 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.

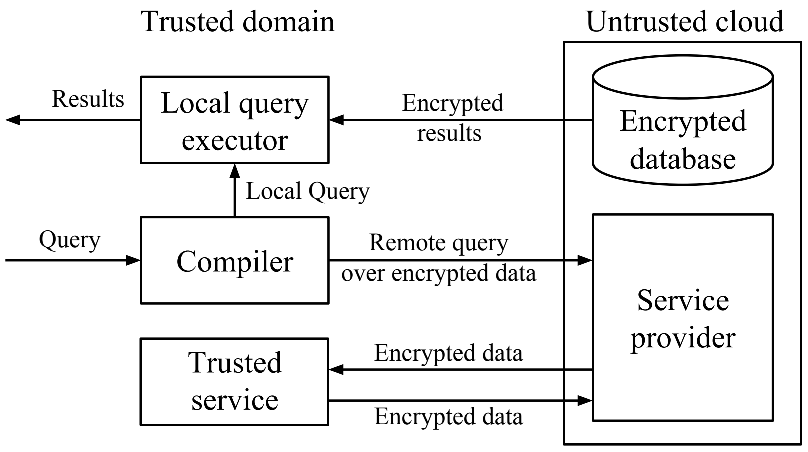


Fig. 4.1.: `Cuttlefish` architecture

Table 4.3.: `Cuttlefish` compilation techniques and high level goals they achieve

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

Since PPE and 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 cryptosystem 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 TEE (e.g., Intel SGX) in the cloud.

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 called *Secure Data Types (SDTs)*. The `Cuttlefish` compiler then uses this information to apply a set of *compilation techniques*. Table 4.3 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 encryption schemes 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 encryption schemes 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. `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 $x > 0 \mid\mid y > 0$. 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.

Table 4.4.: Secure Data Types and corresponding compilation techniques they enable

Secure Data Types	Expression Rewriting	Condition Expansion	Selective Encryption	Efficient Encryption
Sensitivity level	✓		✓	✓
Data range	✓	✓		
Decimal accuracy	✓			
Uniqueness	✓			✓
Tokenization	✓			
Enumerated type				✓
Composite type	✓			

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 cryptosystem. Inversely, when the same operation can be performed by multiple cryptosystems `Cuttlefish` selects the cryptosystem leading to more efficient execution.

4.3 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.

Table 4.5.: Sensitivity levels and associated encryption schemes. DET^{U} denotes deterministic encryption on fields with unique values

Sensitivity Level	**Encryption Scheme**
`HIGH`	RND, AHE, MHE, SRCH, DET^{U}
`LOW`	DET, OPE
`NONE`	plaintext

These are captured by our proposed SDTs and leveraged by the `Cuttlefish` compiler to substantially optimize analytical queries. Table 4.4 cross-references our SDTs with the compilation techniques that leverage them.

4.3.1 Sensitivity Level

As previously mentioned (see Chapter 2 and Table 4.2), cryptosystems used by `Cuttlefish` offer different levels of security. `Cuttlefish` captures this difference by categorizing cryptosystems under the categories `HIGH` and `LOW` (and `NONE`) as shown in Table 4.5. Category `HIGH` captures cryptosystems 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 cryptosystem 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. These fields

Table 4.6.: SDTs annotations

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

must be encrypted under cryptosystems 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 4.4.

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 cryptosystem 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 cryptosystem 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.

4.3.2 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 4.6 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, the SDT `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 cryptosystems 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.

4.3.3 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, This information allows `Cuttlefish` to encrypt that field under a DET cryptosystem while preserving high security guarantees, as described in Table 4.5. In addition, for fields that hold string values the programmer can specify a `delimiter` that separates tokens. This is particularly important when encrypting under a searchable encryption (SRCH) cryptosystem such as SWP, since the latter requires to split the string into individual tokens before encrypting.

4.3.4 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 after their order has been randomized, 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.

4.3.5 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 encryption schemes 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:

$$c := \texttt{composite[}\ t\ s\ t\ \texttt{]}$$

$$t := \texttt{(}num : \alpha\texttt{)} \mid \texttt{(}num : \alpha\texttt{)}\ s\ t$$

$$s := \texttt{-} \mid \texttt{/} \mid \texttt{,} \mid \ldots$$

$$num := \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \ldots$$

where s is a delimiter and α refers to a type such as `int`, `long`, etc. with an optional annotation as shown in Table 4.6 (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` is a 2 digit number 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.

4.3.6 SDTs Example

Listing 4.1 shows the table definition for a subset of fields of the `Lineitem` table used in TPC-H benchmark, defined using SDTs. 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 6 declares the field to be of `enum` type with four possible values, `IN-PERSON`, `MAIL`, `RETURN` and `COLLECT`. Finally, line 7 declares `quantity` to be of type `long` and sets it as a field of

```
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 4.1: `Cuttlefish` table definition with SDTs

`HIGH` sensitivity. Fields for which sensitivity level is not explicitly given are treated as having a sensitivity level of no less than `LOW`.

4.4 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 [36] to carry out this transformation. In this section we focus on describing the set of compilation techniques our compiler applies to the query that leverage SDTs 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.

4.4.1 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 enabling the execution of 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 encryption scheme 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 area code of the phone number 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 4.4.6.

The CES compilation 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 4.4.5, 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 [53, 54].

More generally, the compiler first applies well known expression simplification techniques, namely, Constant Folding (CF), Algebraic Simplification (AS) and Common Sub-expression 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 each operand 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.

4.4.2 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 data 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 or more expensive homomorphic operations. 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.

4.4.3 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 operations of encryption schemes (see Section 2.2 and Table 4.2) 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 operation, 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.

4.4.4 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 cryptosystems. When doing so, `Cuttlefish` minimizes the number of cryptosystems a field is encrypted under by recognizing that some cryptosystems 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 4.3.4. In situations where an enumerated type is involved in deterministic 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 cryptosystems. For example, addition can be achieved by both the Paillier and the ASHE cryptosystem, multiplication can be achieved by ElGamal and Paillier (if one operand is in plaintext) and equality comparison can be achieved by FNR, AES-DET

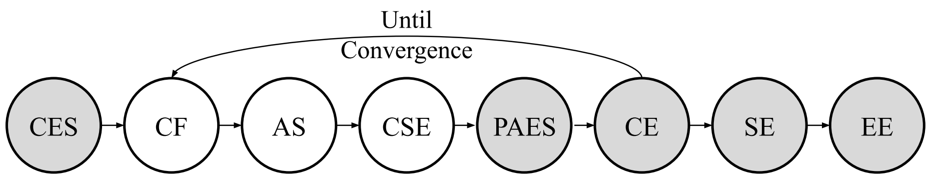


Fig. 4.2.: `Cuttlefish` compilation techniques order. Shaded components indicate novel compilation techniques proposed in this work.

and SWP (if one operand is a fixed constant). In such cases, `Cuttlefish` uses SDT information to select the cryptosystem 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 operation 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 cryptosystem, it leads to a smaller encrypted data size overhead and more efficient query execution.

Lastly, `Cuttlefish` follows the approach of previous work [26] and uses FNR [45], 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, `Cuttlefish` uses AES in CMC mode as used in CryptDB [25].

4.4.5 Order of Compilation Techniques

Figure 4.2 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 cryptosystems to use.

4.4.6 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 4.7.2. Section 4.4.1 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 cryptosystem, 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* [49, 55, 56], 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* [57] 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 once their implementation becomes publicly available.

4.5 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.

4.5.1 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 4.7):

1. Queries start with a large volume of input data but the results of the query are much smaller in size.
2. 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.
3. 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 (e.g., using the secondary homomorphic operation), 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. [58]. 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 $i = 1...N$ 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 $D_f \to E_f$ 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

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

   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.

4.5.2 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 [41] on how to securely use SGX well. Specifically, it allows for a very small and easily verifiable enclave code (see Section 4.6), which keeps the trusted component and the number of interfaces between the trusted and untrusted components small while remaining application independent.

4.5.3 Re-encryption Optimizations

Since encrypting and decrypting data can be costly operations, `Cuttlefish` utilizes a set of optimizations described in Chapter 3 with the aim of reducing the overall cost of re-encryption. `Cuttlefish` uses encryption caching (Section 3.6) to speed up re-encryption to or from deterministic schemes such as DET and OPE, by storing a map of previously re-encrypted values that contains plaintext-ciphertext value pairs. `Cuttlefish` can further optimize re-encryption by speculating what values are more likely to be re-encrypted (Section 3.7). `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. Lastly, `Cuttlefish` uses pseudorandom number (PRN) pre-computation (Section 3.5) and stores a small amount of pre-computed values for Paillier (Equation 3.13) and ElGamal (Equation 3.15) whenever the re-encryption service is idle, so that re-encryption requests can be served faster.

4.6 Implementation

Our prototype implementation of `Cuttlefish`[1] is made up of several components as shown in Figure 4.1. In this section we provide some details about the implementation of these components.

4.6.1 Compiler

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 infor-

[1] `https://github.com/ssavvides/cuttlefish`

mation. 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., schema of encrypted tables, encryption key ids, enum maps etc...) in the form of XML files, which is then used by the transformation module when transforming queries.

4.6.2 Cloud Service

The cloud service runs an unmodified Apache Spark service. Operations on encrypted data are implemented through the use of user-defined functions (UDFs) written in 2200 lines of Scala code, by extending the `UserDefinedFunction` class.

4.6.3 Client Side Re-encryption

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 re-encryption server and in turn, the re-encryption server uses multiple threads to handle these connections.

4.6.4 SGX-based Re-encryption

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 cryptosystems and therefore this code has been well tested over the years. To implement these cryptosystems we port a small multiple-precision arithmetic library into SGX called *BigDigits* [59] (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 Java Native Interface (JNI) to access an interface written in C which then calls the appropriate function in the enclave.

4.7 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 4.7.2). We then compare `Cuttlefish` to other systems to observe relative performance benefits (Section 4.7.3). 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 4.7.4).

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 [60]) and by recent work on studying performance bottlenecks in distributed computation frameworks [61].

We compared `Cuttlefish` with other systems and evaluated individual techniques on both benchmarks. We use TPC-H to show a comparison with other systems and TPC-DS to show the evaluation of individual techniques.

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 [25, 26, 36], 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.

4.7.2 Compilation Techniques Applicability

We first analyze how applicable our proposed compilation techniques are in TPC-H and TPC-DS benchmarks. Table 4.7 shows the number of queries each compilation technique was applicable to for both benchmarks. Expression rewriting allows for 4

Table 4.7.: Compilation technique applicability out of a total of 22 TPC-H queries and 19 TPC-DS queries.

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

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 allows `Cuttlefish` to simplify expressions as discussed in Section 4.4.1 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

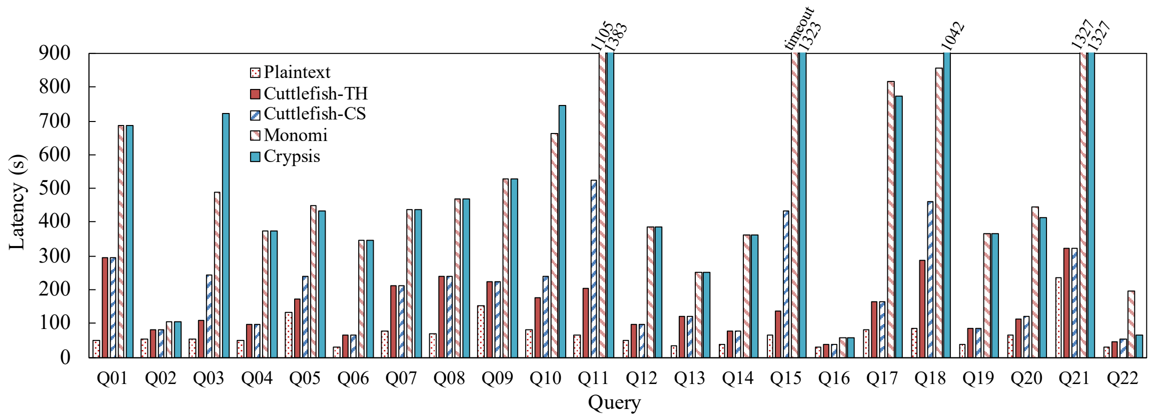


Fig. 4.3.: `Cuttlefish` latency on the TPC-H benchmark compared to plaintext and other PHE-based systems

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.

4.7.3 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. The `Cuttlefish`-TH label denotes 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 [26] system, which we implemented in Spark as opposed 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 [36] which we also implemented in Spark (originally implemented in Apache Pig [62, 63]). 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 4.3, 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.

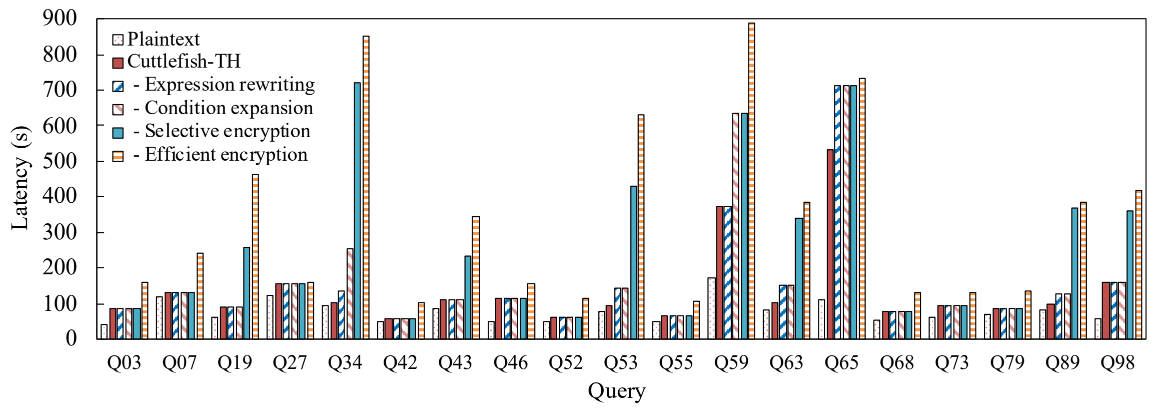


Fig. 4.4.: `Cuttlefish` latency on the TPC-DS benchmark with different compilation techniques enabled

4.7.4 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 4.4 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.

4.8 Related Work

In this section we present an overview on closely related existing systems that preserve confidentiality by using client side computation, PHE, or TEEs. We also present work related to capturing data constraints and multiple sensitivity levels.

4.8.1 Solutions Based on Client Side Computation

SUNDR [64] is a network file system designed to store data securely on untrusted servers. Clients can detect unauthorized file modifications by cryptographically verifying the file system's state. A SUNDR server is kept blind to the private keys of all of its filesystems, essentially minimizing attacks from a compromised server. SUNDR requires the users of a filesystem to be aware of the last operation they performed and verify that the system is in a fork-consistent state (the committed operations history is consistent across users) before any new operation is performed. In essence, failures can be detected after communication between users. While SUNDR performs comparably to NFS, it doesn't scale well in case of large number of users.

SPORC [65] is a framework for building collaborative applications with untrusted servers on the cloud. SPORC's cloud servers see only encrypted data and clients can detect any deviation from correct operation. Similar to SUNDR, SPORC adopts the idea of fork consistency to ensure that the fetch-modify views are consistent across users. On top of that, SPORC applies Operational Transformation (OT) for clients to execute concurrent operations without locking as well as for conflict resolution. The system also supports dynamic admission control where users could modify the access control list, even when they are disconnected to servers.

Depot [66] is a cloud storage system that minimizes trust assumptions while ensuring durability and high availability. Depot enforces fork-join-causal (FJC) consistency (nodes see updates in the right order) to guarantee data integrity. Similar to SUNDR and SPORC, Depot is based on the assumption that the users are aware of their last performed operation. It increases reliability by replicating both data and meta-data associated with update history and overcomes the low availability part of the other two systems using FJC. In all 3 systems, the client nodes can detect a failure by detecting divergent information. Depot and SPORC can even repair such failures by joining these forks to a sensible history. While depot performs well for reads, it doesn't scale well with updates.

The systems above limit the extend to which the cloud can be utilized. Cloud is used as merely a storage system for data since computing resources cannot be utilized without compromising security. For performing any computation involving the data, all required data values need to be download on the client first and all computation is performed on the client. When dealing with large volumes of data, this download and compute model puts enormous load on the client nodes, making it an inefficient solution.

4.8.2 Computing Over Encrypted Data

In their pioneering work, Popa et al. [25] proposed CryptDB, a system that provides confidentiality for applications using a SQL database back-end. 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 [26] 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.

AutoCrypt [37] is a tool that automatically transforms applications written in C to applications that operate over encrypted data. With AutoCrypt, the programmer first designates the input sources that are sensitive and then AutoCrypt uses information flow analysis to propagate dependencies on sensitive fields and infer all sensitive variables. Then AutoCrypt uses type inference on all operations involving sensitive variables and assigns an encryption scheme to each such variable. AutoCrypt utilizes a trusted hypervisor to provide a set of hypercalls that allow converting from one homomorphic scheme to another. AutoCrypt, Uses only three crypto systems (Paillier, ElGamal, Search) and therefore has limited expressiveness. Trusted hypervisor is only used for conversions from one scheme to another. It does not allow partial computations and therefore does not support arbitrary workloads.

Unlike `Cuttlefish`, the above systems do not allow Spark-style parallelization of queries. Furthermore, they depend on data definitions provided by MySQL or Postgres databases and do not utilize properties of encryption schemes to speed up query execution. Instead, `Cuttlefish` generates more efficient queries that operate over encrypted data by leveraging optimizations relying on specifying data ranges, precision, or sensitivity levels.

MrCrypt [35] is a system that statically analyzes MapReduce programs to identify the operation each data column is involved in and transform the program into one that operates over encrypted data, by selecting an appropriate encryption scheme for each column. When a data column is involved in multiple operations or when sequences of operations are applied to a same column, MrCrypt concludes that an FHE scheme is required to carry out the computation in the cloud, noting that the system does not currently execute such jobs at all due to lack of practical FHE schemes.

JCrypt [67] is a system similar to MrCrypt and transforms Java and MapReduce programs into equivalent programs which perform computations over encrypted data. JCrypt uses information flow analysis to split the variables into sensitive and non sensitive ones and tries to keep the sensitive variables minimal for optimal performance. For better expressiveness, if the same variable is used in multiple operations, the variable is re-encrypted using a different scheme when possible.

SecureScala [68] is a domain-specific language in Scala that allows expressing secure programs without requiring any cryptographic knowledge from the programmer.

Crypsis [27, 36] 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 4.7.

4.8.3 Solutions Based on Trusted Hardware

A number of recent approaches rely on specialized hardware and TEE, e.g., Intel SGX [69] that provides a trusted area of execution to preserve data confidentiality while computing. Haven [70] relies on hardware encrypted and integrity protected physical memory provided by SGX to ensure application security. Cipherbase [71] 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 [72] preserves confidentiality by using a server-hosted, tamper-proof trusted hardware in critical query processing stages.

VC3 [73] is a system that uses SGX to provide confidential and integrity protected general purpose data processing. It focuses on MapReduce computations and retains a small TCB by keeping most of the MapReduce framework outside the TCB. Map and Reduce logic is executed inside enclaves while scheduling, load balancing fault tolerance and other services run unchanged in the normal world. Map and Reduce can optionally run with read or write integrity which is a set of run-time checks that ensure code running inside enclaves does not read or write memory locations that leak information or can potentially compromise the enclave isolation, e.g., writes outside the enclave memory.

Opaque [43] is a data analytics engine built on top of Spark that uses SGX to carry out secure computations. Opaque ensures confidentiality and integrity of computations, and prevents information leakage from access patterns by introducing a set of oblivious operations. These operations also work in a distributed and parallel setting. Opaque also introduces optimizations based on new query planning techniques for a cost model that captures the behavior of these oblivious operations.

Iron [74] provides a practical implementation of functional encryption (FE) by using SGX enclaves to achieve the same effect as FE. Iron first uses a key manager enclave to generate a private and public key for an elliptic curve ElGamal cryptosystem which is used to encrypt sensitive data. The private key is then sent to a decryption enclave through a secure channel after the decryption enclave has been attested. When a client is authorized to access a function f of the sensitive data, a function enclave is initialized with a binary that applies f on the sensitive data. Once the function enclave is attested and proves its authenticity to the decryption enclave, the latter sends the secret key to the function enclave, which decrypts the sensitive data first, applies f on the sensitive data and returns the result to the client.

Gollamudi et al. [75] propose a calculus called $\mathrm{IMP_E}$ that can be used to express programs that leverage SGX-like enclave mechanisms. $\mathrm{IMP_E}$ includes a security-type system that is used to enforce confidentiality policies in the presence of a passive or an active attacker by automatically partitioning non-enclave enabled programs into semantically equivalent programs that execute sensitive code inside enclaves. $\mathrm{IMP_E}$ defines confidentiality restrictions in terms of three security levels, namely, L, H and $\top$. L is for public information that can be learned by the adversary, H is for sensitive information that only trusted parties can access and $\top$ is for information that should not exist in the system at all.

Moat [76] is a tool which verifies the confidentiality properties of applications running on SGX. This is useful in case of vulnerabilities in the application itself, such as the incorrect use of SGX instructions or memory safety errors, which can be exploited to reveal information. Moat employs a flow and path-sensitive type checking algorithm for automatically verifying whether an enclave program leaks a secret to the adversary-visible, non-enclave memory. They also create a formal semantic model of the interface between the programmer and the implementation of SGX along with a relevant adversary model to better understand the contract between hardware and software.

Sinha et al. [77] also present a methodology for designing applications that enables certifying their confidentiality. The main difference between the two is that while Moat works for all code, this system requires the application to perform all communications through a narrowly constrained interface, compiling it with runtime checks that aid verification, and linking it with a small runtime library that implements the interface. These constraints on user code are formalized as Information Release Confinement (IRC), and a modular automatic verifier to check IRC is presented.

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 [39,40]. 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 [78], 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 4.7) which are too big to run directly in SGX, thereby benefiting from in-memory computations [44], and uses SGX only for re-encryption of small amounts of data.

4.8.4 Data Constraints

Managing information flows with multiple sensitivity levels within programs has been widely studied. Denning's seminal work [79, 80] in the field was further formulated as a type system by Volpano et.al. [81]. 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 [82, 83]. The primary focus in these approaches are type inference, checking and correctness. In the context of encrypted query processing, CryptDB [25] 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.

4.9 Conclusion

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 chapter presented `Cuttlefish`, a system for confidentiality-preserving data analytics in the cloud which relies on PHE and PPE, combined with smart re-encryption exploiting SGX when available to overcome limitations of computing over encrypted data. 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.

5. CONFIDENTIAL DATA ANALYTICS USING SYMMETRIC PARTIALLY HOMOMORPHIC ENCRYPTION

In the two previous chapters of this thesis, we highlighted the need for data confidentiality in cloud analytics and demonstrated how Partially Homomorphic Encryption (PHE) combined with other technologies can be used to achieve that. However, despite the optimizations we presented in Chapter 3 and the techniques we introduced in Chapter 4 which enable us to use PHE schemes in clever and more efficient ways, the overhead of asymmetric PHE schemes remains non-trivial.

In this chapter, we present two novel PHE schemes, an *Additive* and a *Multiplicative Homomorphic Encryption* scheme, which, unlike previous schemes, are *symmetric*. We prove the security of our schemes and show they are more efficient than state-of-the-art asymmetric PHE schemes, without compromising the expressiveness of homomorphic operations they support. The main intuition behind our schemes is to trade strict ciphertext compactness for good "relative" compactness in practice, while in turn reaping improved performance. We build a prototype system called `Symmetria` that uses our proposed schemes and demonstrate its performance improvements over previous work. `Symmetria` achieves up to 7× average speedups on standard benchmarks compared to asymmetric PHE-based systems.

5.1 Overview

Cloud computing has become ubiquitous due to its economical and practical paradigm. Third-party clouds are nowadays used by both corporations and governments to perform cost-effective computations. Oftentimes, these computations require sensitive data to be moved to the cloud which places trust on the cloud provider. Resource sharing among multiple tenants only adds to the problem of ensuring data confidentiality. Due to this fact, many organizations are still reluctant to use third party clouds.

One approach being pursued to alleviate the confidentiality concerns is *homomorphic encryption.* Fully Homomorphic Encryption (FHE) allows arbitrary operations to be performed directly over encrypted data and thus preserves the confidentiality of data throughout computations. Gentry introduced FHE and a first cryptosystem [3, 33] that provably achieves it. Though FHE has been becoming more practical [4], it still exhibits performance costs that are prohibitive for many computations. An alternative to FHE is PHE. PHE denotes schemes that allow individual operations over encrypted data, e.g., addition, multiplication. By using multiple PHE schemes side-by-side, many operations can be supported with practical overhead. This is particularly true when Property-Preserving Encryption (PPE) schemes are used, in addition to PHE. Ciphertexts of PPE schemes preserve some property of the underlying plaintext and can be used to carry out comparisons.

Although PHE is much faster compared to FHE, its overhead remains non-trivial. This is because most existing PHE schemes such as ElGamal [10], Benaloh [6], Paillier [5], and RSA [11] are asymmetric. Asymmetric schemes use a public key to encrypt messages and another private key to decrypt messages and tend to fall behind symmetric schemes in terms of performance since they have large ciphertext spaces leading to large ciphertext size overheads. Furthermore, the homomorphic operations of asymmetric schemes require complex computations involving arbitrary-precision arithmetic operations.

Previous attempts to propose symmetric PHE schemes that are more performant than their asymmetric counterparts have done so at the expense of *expressiveness.* The Additively Symmetric Homomorphic Encryption (ASHE) [38] for example is a symmetric Additive Homomorphic Encryption (AHE) scheme that enables addition of two encrypted values. ASHE is much faster than asymmetric AHE schemes such as Paillier but has limited expressiveness. Specifically, ASHE only supports addition of two ciphertexts, whereas other, asymmetric, AHE schemes support addition and subtraction between two ciphertexts or between a ciphertext and a plaintext value, negation of a ciphertext, as well as multiplication between a ciphertext and a plaintext.

The paucity of PHE schemes that are both performant (symmetric) and expressive in terms of the homomorphic operations they support, forced previous PHE-based systems to utilize multiple schemes for the same type of homomorphism. For example, Cuttlefish [84] switches between the symmetric, more performant, ASHE scheme when only additions are required, and falls back to the asymmetric, more computationally expensive, Paillier scheme when additional homomorphic computations (such as the ones described above) are required. Similarly, Cipherbase [71] uses a dual hardware and software co-design, switching between PHE schemes and a trusted co-processor for secure computations. While switching between schemes allows these systems to execute queries more efficiently, having to utilize multiple schemes makes their design less flexible due to switching overhead.

In this chapter, we thus introduce two novel symmetric encryption schemes: 1) *Symmetric Additive Homomorphic Encryption (SAHE)* and 2) *Symmetric Multiplicative Homomorphic Encryption (SMHE)* designed specifically to retain the expressiveness of state-of-the-art PHE schemes while providing improved performance compared to previously used asymmetric PHE schemes. The main intuition behind the design of our schemes is to trade ciphertext *compactness* (size remains the same as homomorphic operations are performed) in the strict sense (i.e., qualitatively) for good "relative" compactness in practice (quantitatively), while in turn reaping improved performance in practice. We observe that:

1. In many scenarios ciphertext compactness (or lack thereof) has little to no effect on performance. Many applications require arithmetic operations between a few, fixed number of ciphertext values – in the case of relational data, one can simply think of computations across *columns* for individual rows. Here

the ciphertext size overhead added due to the lack of compactness guarantee is bounded.

2. By applying a set of compaction techniques (see Section 5.2.4) we can limit the ciphertext expansion in unbounded sequences of operations such as aggregations over à priori unbounded sets of values. Examples include aggregations across *rows* in relational data. As we show empirically in Section 5.5, our schemes achieve practical overhead despite not being compact.

3. Many of the homomorphic operations our schemes support, by their nature, do not increase the ciphertext size.

We provide formal proofs of security of our schemes and show that they satisfy standard notions of security, i.e., semantic security under Chosen Plaintext Attack (CPA). To the best of our knowledge, there is no symmetric AHE scheme that preserves the expressiveness of existing asymmetric AHE schemes and there is no symmetric MHE scheme at all. Here we make a distinction between PHE and somewhat homomorphic encryption (SWHE) [85] schemes. Symmetric SWHE schemes that support multiplication do exist but, unlike our schemes, only allow a limited number of operations.

We present a prototype system called `Symmetria` that uses our proposed schemes SAHE and SMHE to perform arithmetic operations over encrypted data. We further extend `Symmetria` to utilize a number of existing PPE schemes to allow comparisons over encrypted data and therefore support a wider range of queries. We compare our symmetric schemes against asymmetric schemes such as Paillier and ElGamal. We evaluate `Symmetria` and demonstrate its low overhead over plaintext computations. `Symmetria` achieves average speedups of 3.8× and 7× over state-of-the-art asymmetric PHE-based systems on the standard TPC-H and TPC-DS benchmarks respectively. In summary, in this chapter we make the following contributions:

- We propose a novel semantically secure AHE scheme called SAHE that supports addition as well as other homomorphic operations over encrypted data.

- We propose a novel semantically secure MHE scheme called SMHE that supports multiplication as well as other homomorphic operations over encrypted data.
- We introduce a set of compaction techniques to limit the overhead of ciphertext size for our proposed schemes.
- We show the design and evaluation of a system called `Symmetria` that uses our proposed schemes and employs a set of query optimizations to execute queries over encrypted data with little overhead.

In the rest of this chapter, we first describe the design of SAHE and SMHE in Section 5.2 and then in Section 5.3 we provide an overview of `Symmetria`. In Section 5.4 we discuss implementation details of `Symmetria`. In Section 5.5 we empirically evaluate `Symmetria` and our proposed schemes. In Section 5.6 we present work related to ours and in Section 5.7 we conclude with final remarks.

5.2 Symmetric PHE

Section 2.2 describes a set of AHE and MHE schemes such as the Paillier and the ElGamal cryptosystems. These cryptosystems are asymmetric and their security is based on mathematical problems that are hard to solve for large numbers. For example, if the modulus, N, of the public key of Paillier can be factored, the security of the cryptosystem no longer holds. For small values of N, solving this problem is trivial and therefore sufficiently large numbers should be used. For this reason, NIST recommends N to be at least 2048-bits long [15], making the $\bmod N^2$ homomorphic computations in Paillier 4096-bits long (see Equation 2.2 for details). Furthermore, NIST recommends that 2048-bits keys should not be used beyond the year 2030, with other organizations suggesting that 3072-bit long keys are already needed today [86, 87]. Oftentimes, these cryptosystems and the associated homomorphic computations are implemented using an arbitrary-precision arithmetic library such as the GMP library, or the BigInteger immutable arbitrary-precision arithmetic primitive in Java. These libraries, though highly optimized, can exhibit high overheads for computations involving large numbers, which leads to slow homomorphic operations.

ASHE [38] is a AHE scheme which is *symmetric* and therefore more performant than asymmetric PHE schemes. Unfortunately, the performance improvements of ASHE come at the expense of expressiveness since ASHE only supports additions between two ciphertexts, unlike the asymmetric schemes described in Section 2.2 that support a wide range of operations over encrypted data, including secondary homomorphic operations between a ciphertext and a plaintext value.

In this section, we tackle both performance and expressiveness limitations of previous PHE schemes and we describe two novel *symmetric* PHE schemes that can replace existing asymmetric schemes such as Paillier and ElGamal used for arithmetic operations in state-of-the-art PHE-based systems [25, 28, 84, 88, 89]. Our schemes trade strict compactness for quantitatively good compactness in practice, which is supported by a set of compaction techniques. These techniques allow our schemes to achieve better performance than asymmetric schemes without sacrificing expressiveness, as they retain the full range of homomorphic operations of traditional PHE schemes.

5.2.1 Symmetric AHE

Symmetric Additive Homomorphic Encryption (SAHE) is a symmetric Additive Homomorphic Encryption scheme that allows homomorphic additions.

Scheme description SAHE is defined in the abelian additive group $\mathbb{Z}_N$ of order $N > 1$. Let $F : \{0,1\}^n \times \{0,1\}^n \rightarrow \mathbb{Z}_N$ defined as $F_k(x) = F(k, x)$ be a pseudorandom function (PRF) mapping strings of length n to elements of $\mathbb{Z}_N$, where n is the security parameter which dictates the length of the key and must be set according to the application needs. Below we formally define the scheme $\Pi_{SAHE} = (\mathsf{gen}, \mathsf{enc}, \mathsf{dec})$.

$\mathsf{gen}(1^n)$: output uniform $k \in \{0,1\}^n$ as the symmetric key.

$\mathsf{enc}(k, m)$: on input key $k \in \{0,1\}^n$ and message $m \in \mathbb{Z}_N$ choose uniform $r \in \{0,1\}^n$ and output the ciphertext

$$c := \langle (m + F_k(r)) \ \mathsf{mod}\ N, [r], \varnothing \rangle$$

The resulting ciphertext c is a triplet $\langle v, l_p, l_n \rangle$ where v is the obfuscated value, l_p is a list of identifiers (ids) each of which is used to generate a random element in $\mathbb{Z}_N$ and added to m, and, l_n is a list of ids used to generate a random element in $\mathbb{Z}_N$ and subtracted from m. When initially encrypting a value m, there is only one id added to m to get $v = (m + F_k(r))\ \mathsf{mod}\ N$ and therefore $l_p = [r]$ and $l_n = \varnothing$ (empty list).

$\mathsf{dec}(k, c)$: on input key $k \in \{0,1\}^n$ and ciphertext $c = \langle v, l_p, l_n \rangle$ output the plaintext message

$$m := (v - \sum_{r_1 \in l_p} F_k(r_1) + \sum_{r_2 \in l_n} F_k(r_2))\ \mathsf{mod}\ N$$

Proof of security We provide proof of the security of SAHE and show that it is secure under the assumption that F_k is a secure PRF. Note that lists l_p and l_n hold ids that act as inputs to F_k and can remain in the clear as long as the key k is kept secret.

Theorem 5.2.1 *If F is a PRF, the cryptographic construction Π_{SAHE} is a semantically secure under CPA (IND-CPA) private-key encryption scheme for messages of $\mathbb{Z}_N$.*

Proof We assume a Probabilistic Polynomial-Time (PPT) adversary A that attempts to break the semantic security of $\Pi_{SAHE} = (\mathsf{gen}, \mathsf{enc}, \mathsf{dec})$, for the rest of the proof simply Π. We denote the probability of A succeeding as $Pr[CPA_\Pi^A = 1]$ where CPA_Π^A is the CPA indistinguishability experiment. We show that if A can break the semantic security of Π we can construct an attacker that can distinguish a PRF F from a truly random function f.

We proceed by a proof by reduction by using adversary A as a subroutine to construct a distinguisher D. D has oracle access to a function $\mathcal{R} : \{0,1\}^n \to \mathbb{Z}_N$ which can either be a PRF or a truly random function, and the goal of D is to determine which is the case. D emulates CPA^A and when A requests the encryption of a message $m \in \mathbb{Z}_N$, D generates a uniform string $r \in \{0,1\}^n$, uses the oracle to get $x := \mathcal{R}(r)$, and returns the ciphertext $\langle (m+x) \bmod N, [r], \varnothing \rangle$ to A. Recall that n is chosen such that adversary A can issue a polynomial number of such encryption requests which we denote as $poly(n)$. This process is repeated until A outputs messages $m_0, m_1 \in \mathbb{Z}_N$. In return, D chooses a uniform bit $b \in \{0,1\}$, a uniform string $r \in \{0,1\}^n$, uses the oracle to get $x := \mathcal{R}(r)$, and returns the challenge ciphertext $\langle (m_b+x) \bmod N, [r], \varnothing \rangle$ to A. Finally, when A outputs a bit b' indicating a guess on what message was encrypted, D outputs 1 if $b = b'$, and 0 otherwise.

Let $\widetilde{\Pi}$ be a cryptographic construction same as Π but where the PRF F is replaced with a truly random function f. Notice that if $\mathcal{R}$ is a PRF then $x := F_k(r)$ and the emulation of CPA^A matches the experiment with construction Π, i.e., CPA^A_Π. Similarly, when $\mathcal{R}$ is a truly random function then $x := f(r)$ and the emulation of CPA^A matches the experiment with construction $\widetilde{\Pi}$, i.e., $CPA^A_{\widetilde{\Pi}}$. Therefore, by the assumption that F_k is a secure PRF, there exists negligible probability ϵ s.t.

$$|\ Pr[CPA^A_\Pi = 1] - Pr[CPA^A_{\widetilde{\Pi}} = 1]\ |\ \leq \epsilon \tag{5.1}$$

In the case of $\mathcal{R}$ being a truly random function, we consider the probability of the adversary A having received a ciphertext encrypted using the same random string $r \in \{0,1\}^n$ as the one used to generate the challenge ciphertext $c = \langle (m + f(r)) \bmod N, [r], \varnothing \rangle$. Note that in such event, A can infer $f(r) = (c - m) \bmod N$ and use that to determine whether the challenge cipher was generated from m_1 or m_2, winning the indistinguishability game. Since A can perform $poly(n)$ number of encryption requests, and each encryption requires one out of a total of 2^n possible strings, such an event can occur with probability $\frac{poly(n)}{2^n}$. Therefore we have

$$Pr[CPA^A_{\widetilde{\Pi}} = 1] \leq \frac{1}{2} + \frac{poly(n)}{2^n}. \tag{5.2}$$

Using Equation 5.1 and Equation 5.2 we get

$$Pr[CPA_{\Pi}^{A} = 1] \leq \frac{1}{2} + \frac{poly(n)}{2^n} + \epsilon$$

where $\frac{poly(n)}{2^n} + \epsilon$ is negligible and therefore the advantage of adversary A is negligible. ■

Homomorphic operations Next, we describe the set of homomorphic operations $\Phi_{SAHE} = (\mathsf{add}, \mathsf{adp}, \mathsf{mlp}, \mathsf{neg}, \mathsf{sub})$ of SAHE. In what follows, we use c to denote a ciphertext ($c = \langle v, l_p, l_n \rangle$), c_1 and c_2 to denote a pair of ciphertexts ($c_1 = \langle v_1, l_{p_1}, l_{n_1} \rangle$ and $c_2 = \langle v_2, l_{p_2}, l_{n_2} \rangle$) and m to denote a plaintext value. As previously mentioned, ciphertexts are randomized and none of the operations in Φ_{SAHE} require the use of trapdoors.

add: This operation homomorphically adds two ciphertexts:

$$\mathsf{add}(c_1, c_2) := \langle (v_1 + v_2) \bmod N, l_{p_1} \cup l_{p_2}, l_{n_1} \cup l_{n_2} \rangle$$

Here $\cup$ indicates list concatenation.

adp: The add plaintext operation expresses a homomorphic addition between a ciphertext and a plaintext value:

$$\mathsf{adp}(c, m) := \langle (v + m) \bmod N, l_p, l_n \rangle$$

mlp: Even though SAHE is an AHE scheme, it supports multiplication between a ciphertext and a plaintext value through the mlp (multiply plaintext) operation:

$$\mathsf{mlp}(c, m) := \langle (v \times m) \bmod N, l_1, l_2 \rangle$$

where

$$l_1, l_2 = \begin{cases} l_n^{|m|}, l_p^{|m|} & \text{if } m < 0 \\ \varnothing, \varnothing & \text{if } m = 0 \\ l_p^{|m|}, l_n^{|m|} & \text{if } m > 0 \end{cases} \tag{5.3}$$

In the above, $|m|$ denotes the absolute value of m and the list operation $l^{|m|}$ denotes $|m-1|$-fold concatenation of list l with itself. For example, if $l = [r_1, r_2]$ and $m = 3$ then $l^{|m|} = [r_1, r_2, r_1, r_2, r_1, r_2]$. This example shows that the lists can grow quickly as more homomorphic operations are performed. In Section 5.2.4 we introduce a set of compaction techniques to alleviate this concern.

neg: We define neg, the unary operation of negating an encrypted value based on the mlp function:

$$\begin{aligned} \mathsf{neg}(c) &:= \mathsf{mlp}(c, -1) \\ &= \langle -v \ \mathsf{mod}\ N, l_n, l_p \rangle \end{aligned}$$

sub: We define the subtraction operation sub based on the homomorphic operations add and neg:

$$\begin{aligned} \mathsf{sub}(c_1, c_2) &:= \mathsf{add}(c_1, \mathsf{neg}(c_2)) \\ &= \langle (v_1 - v_2) \ \mathsf{mod}\ N, l_{p_1} \cup l_{n_2}, l_{n_1} \cup l_{p_2} \rangle \end{aligned}$$

5.2.2 Symmetric MHE

Symmetric Multiplicative Homomorphic Encryption (SMHE) is a symmetric Multiplicative Homomorphic Encryption scheme that allows homomorphic operations with respect to multiplication.

Scheme description SMHE is defined in the abelian multiplicative group $\mathbb{Z}_N^*$ of order $N > 1$. Just like with SAHE, we assume there exists a PRF $F_k(x) = F(k, x)$ generating elements of $\mathbb{Z}_N^*$, and use a fixed generator g for the selected group order. Below we formally define the scheme $\Pi_{SMHE} = (\mathsf{gen}, \mathsf{enc}, \mathsf{dec})$.

$\mathsf{gen}(1^n)$: same as SAHE gen function.

$\mathsf{enc}(k, m)$: on input key $k \in \{0,1\}^n$ and message $m \in \mathbb{Z}_N^*$ choose uniform $r \in \{0,1\}^n$ and output the ciphertext

$$c := \langle (m \times g^{F_k(r)}) \bmod N, [r], \varnothing \rangle$$

Ciphertext c is a triplet $\langle v, l_p, l_n \rangle$ but for SMHE l_p denotes the list of ids each of which is used to generate a random element in $\mathbb{Z}_N^*$ which is then raised to the power of g and multiplied by m to generate the obfuscated value v. l_n is similar except that the generated value $g^{F_k(r)}$ for a given r is inverted before being multiplied by m.

$\mathsf{dec}(k, c)$: on input key $k \in \{0,1\}^n$ and ciphertext $c = \langle v, l_p, l_n \rangle$ output the plaintext message

$$m := (v \times \prod_{r_1 \in l_p} g^{-F_k(r_1)} \times \prod_{r_2 \in l_n} g^{F_k(r_2)}) \bmod N$$

Proof of security Here, we prove that SMHE is secure under the assumption that F is a secure PRF.

Theorem 5.2.2 *If F is a PRF, the cryptographic construction Π_{SMHE} is a semantically secure under CPA (IND-CPA) private-key encryption scheme for messages of $\mathbb{Z}_N^*$.*

Proof As the proof is similar to the one for SAHE, we only provide the parts specific to SMHE. First, we note that since g is a generator of the group $\mathbb{Z}_N^*$ every element in the group can be expressed via g^x for some x. Thus, assuming a truly random function f, for uniform r, $g^{f(r)}$ is uniformly distributed in the group. Since g is a generator and $m \in \mathbb{Z}_N^*$ then $m = g^x$ for some x. Therefore $m \times g^{f(r)} = g^x \times g^{f(r)} = g^{x+f(r)} \bmod N$ is uniform as well. The rest of the proof proceeds in three steps like the proof for SAHE by 1. proving that replacing f with the PRF F_k only gives the adversary negligible advantage, 2. showing that a modified Π_{SMHE} scheme where $F_k(r)$ is replaced with f is semantically secure and 3. showing that $poly(n)$ is the upper bound on the number of queries that the adversary can send to the oracle of the indistinguishability game, and the only way the adversary has an advantage is when the same r is used more than once which happens only with negligible probability. ■

Homomorphic operations In this section, we describe the set of homomorphic operations $\Phi_{SMHE} = (\mathsf{mul}, \mathsf{mlp}, \mathsf{pow}, \mathsf{inv}, \mathsf{div})$ of SMHE.

mul: This operation denotes homomorphic multiplication:

$$\mathsf{mul}(c_1, c_2) := \langle (v_1 \times v_2) \ \mathsf{mod}\ N, l_{p_1} \cup l_{p_2}, l_{n_1} \cup l_{n_2} \rangle$$

mlp: The multiply plaintext operation denotes homomorphic multiplication between a ciphertext and a plaintext:

$$\mathsf{mlp}(c, m) := \langle (v \times m) \ \mathsf{mod}\ N, l_p, l_n \rangle$$

pow: SMHE supports exponentiation between a ciphertext and a plaintext value through the pow operation:

$$\mathsf{pow}(c, m) := \langle (v^m) \ \mathsf{mod}\ N, l_1, l_2 \rangle$$

Here l_1 and l_2 are computed as per Equation 5.3.

inv: The multiplicative inverse of a ciphertext can be homomorphically computed as follows:

$$\mathsf{inv}(c) := \mathsf{pow}(c, -1)$$

div: Finally, homomorphic division (multiplication with the inverse) is defined as:

$$\mathsf{div}(c_1, c_2) := \mathsf{mul}(c_1, \mathsf{inv}(c_2))$$

5.2.3 Security Properties

In this section, we discuss some security properties of our symmetric schemes SAHE and SMHE.

Statefulness So far the description of SAHE and SMHE required that we generate a uniform id $r \in \{0,1\}^l$ when encrypting, although the homomorphic operations and the security of our schemes do not require r to be uniform. It is sufficient for r to be unique across all values encrypted under the same key k. We can thus change the enc algorithm of both of our schemes from being randomized to being stateful. By keeping track of the last id used (for a given key k), ids can be chosen in an incremental fashion. By default, `Symmetria` uses the stateful implementation of enc for both SAHE and SMHE which leads to more regular lists of ids and make the application of the compactness techniques described in Section 5.2.4 more effective.

Malleability and null values Similar to other PHE schemes, homomorphic operations of our schemes are a result of the schemes being malleable and, therefore, allow some limited alterations to ciphertexts which result in predictable changes to the underlying plaintexts. As a result, stronger notions of security such as IND-CCA2 are not satisfied by our (or any previous) homomorphic schemes. Asymmetric schemes allow the creation of any ciphertext without access to the private key since encryption only requires the public key. So far the description of our schemes permits the creation of arbitrary ciphertexts without having access to the secret key k, despite our schemes being symmetric. For example, for either SAHE or SMHE and for any message $0 \leq m < N$, one can construct the ciphertext $c^* = \langle m, \varnothing, \varnothing \rangle$ where the obfuscated value of the ciphertext is set to m itself and both id lists l_p and l_n are empty. As per discussion so far, c^* is a legitimate ciphertext that once decrypted will return the message m, i.e., dec $(c^*, k) = m$. Ciphertexts with both id lists being empty can also occur by applying operations on a non-empty ciphertext. For instance, given a SAHE ciphertext c one can perform the operation $c_0 = \mathsf{mlp}(c, 0)$ to get a valid (per discussion so far) ciphertext c_0 of the value 0. Then, by repeated application of $\mathsf{adp}(c_0, 1)$ one can create a valid ciphertext for any value desired. To address this issue we treat ciphertexts with both id lists being empty in a special way. Given a ciphertext triplet $\langle v, l_p, l_n \rangle$, if both lists are empty ($l_p = \varnothing$ and $l_n = \varnothing$) the ciphertext is considered as a null ciphertext regardless of the value v. We denote the null ciphertext of SAHE and SMHE by $\varnothing^A$ and $\varnothing^M$ respectively. Decryption considers such ciphertexts as a special case. For both SAHE $\mathsf{dec}(\varnothing^A, k) = 0$ (identity element of $\mathbb{Z}_N$) and SMHE $\mathsf{dec}(\varnothing^M, k) = 1$ (identity element of $\mathbb{Z}_N^*$), the obfuscated value v is ignored.

5.2.4 Compaction Techniques

Ciphertexts of SAHE and SMHE are made up of three elements $\langle v, l_p, l_n \rangle$. The size of v depends on the choice of N and is fixed. The size of id lists l_p and l_n however can vary as homomorphic operations are performed. As l_p and l_n increase in size, the memory footprint of `Symmetria` grows. Moreover, the performance of homomorphic operations and decryption of ciphertexts degrades as these lists grow. To reduce these overheads we propose a set of techniques to compact these lists. We also define alternatives for enc and dec that leverage the techniques for improved performance.

List aggregation Ids in lists l_p and l_n are used to generate a random value using a PRF F_k which in turn is used to apply an operation during encryption or decryption. For both SAHE and SMHE, lists l_p and l_n describe inverse operations. For instance, in SAHE, list l_p holds ids used to generate a random value *added* to the message m, and l_n holds ids used to generate a random value *subtracted* from m. These operations can cancel each other out, meaning that ids that appear in both lists can be safely removed. For example, the two lists $l_p = [r_1, r_2, r_3]$ and $l_n = [r_1, r_4]$ can be reduced to $l_p = [r_2, r_3]$ and $l_n = [r_4]$.

Id grouping Homomorphic operations of both SAHE and SMHE can lead to ids appearing multiple times in the same id list. For example, after an mlp operation using the SAHE scheme, each id will by copied multiple times in both id lists l_p and l_n, as per Equation 5.3. To keep id lists more concise, same ids within a list can be grouped. For example, the list of ids $[r_1, r_1, r_1, r_2]$ can be represented as $[3 : r_1, r_2]$.

Here "3 :" indicates the cardinality of the id r_1. Furthermore, subsequent additions of the id r_1 to the list do not change the list size or the overall ciphertext size and instead, the cardinality of the id r_1 is increased. This technique not only reduces ciphertext footprint but also improves decryption performance as we describe next. We define a function $\mathcal{C} : \{0,1\}^l \rightarrow \mathbb{N}$ which, given an id $r \in \{0,1\}^l$, returns the cardinality of that id in the list, i.e., how many times that id appears in the list. Using this function, we can change the definitions of decryption functions (dec) for SAHE and SMHE respectively as follows:

$$m := (v - \sum_{r_1 \in l_p} \mathcal{C}(r_1) \times F_k(r_1) + \sum_{r_2 \in l_n} \mathcal{C}(r_2) \times F_k(r_2)) \bmod N$$

$$m := (v \times \prod_{r_1 \in l_p} g^{-\mathcal{C}(r_1) \times F_k(r_1)} \times \prod_{r_2 \in l_n} g^{\mathcal{C}(r_2) \times F_k(r_2)}) \bmod N$$

With this technique enabled, list concatenation (Equation 5.3) can be efficiently implemented simply by updating id cardinalities.

Range folding Sequences of consecutive ids is another pattern appearing in id lists, especially when ids are selected sequentially at encryption (see Section 5.2.3). These sequences can be folded into a *range* of ids. E.g., consider the id list $[2, 3, 4, 5, 8]$. This list can be compacted to $[2 - 5, 8]$. In this example, subsequent additions of the ids 1 or 6 do not increase the size of the list as these ids will simply extend the existing range. To make this technique more effective, we generalize range folding to fold sequences of ids of any step and not just consecutive sequences, i.e., sequences of step 1. We have observed that in practice such patterns appear frequently, for instance when summing over a column, even after some rows are filtered.

Telescoping To enable efficient execution of aggregation functions we use the telescoping technique. Consider the case where a sizable number of ciphertexts need to be summed. The ids of these ciphertexts can be consecutive in which case the resulting ciphertext will be compact due to the range folding technique discussed above. Even if this is the case, while decrypting, $F_k(r)$ needs to be generated for each id r even if

r is folded in a range, making decryption impractical. To avoid this, we change the encryption functions for SAHE and SMHE respectively as follows:

$$c := \langle (m + F_k(r) - F_k(r+1)) \bmod N, [r], [r+1] \rangle$$

$$c := \langle (m \times g^{F_k(r)} \times g^{-F_k(r+1)}) \bmod N, [r], [r+1] \rangle$$

Here, for each id r we generate $F_k(r)$ and $F_k(r+1)$ and apply inverse operations based on these, storing r in l_p and $r+1$ in l_n. For consecutive values of r, these operations cancel each other out based on our list aggregation technique, making decryption practical, at the expense of having to generate an additional pseudorandom number (PRN) at encryption.

Integer list compression Our schemes can also benefit from compaction techniques proposed in previous work. In particular, the problem of efficiently compressing arrays of integers has been studied extensively [90, 91]. A requirement of such algorithms is that the integers of the array are not random. Our lists of ids satisfy this requirement by storing ids in non-decreasing order. Another requirement for these algorithms to be effective is that most integers in the array are small or gaps between them are small. These requirements are met by our schemes by choosing ids in an incremental manner (Section 5.2.3).

5.2.5 Scheme Compactness

As SAHE and SMHE are not strictly compact each application of a homomorphic operation might yield a ciphertext with increased size. Here we discuss how the compactness techniques proposed above can be applied to achieve homomorphic operations that are (quantitatively) compact in practice. Table 5.1 outlines the conditions for respective operations to retain compactness. Any homomorphic operation is compact if it involves (a) adding an id to the l_p list and the same id exists in the l_n list or vice-versa (Section 5.2.4); (b) adding an id to a list that already contains the id (Section 5.2.4); (c) adding an id to a list that falls within an existing range

Table 5.1.: Scheme operation compactness.

SAHE	SMHE	Compactness Condition
add	mul	Techniques Section 5.2.4, Section 5.2.4, and Section 5.2.4
adp	mlp	By definition
mlp	pow	Technique Section 5.2.4
neg	inv	By definition
sub	div	Techniques Section 5.2.4, Section 5.2.4, and Section 5.2.4

(Section 5.2.4). Some homomorphic operations of our schemes are also compact by definition. In particular, operations involving a single ciphertext (e.g., neg, inv) as well as operations involving a ciphertext and a plaintext value (e.g. SAHE operations adp and mlp, SMHE operations mlp and pow) are always compact. We evaluate the effects of (a) scheme compactness in terms of ciphertext size overhead and execution time in Section 5.5.4 and (b) individual compaction techniques in Section 5.5.6.

5.3 Symmetria: System Design

We first explain the threat model underlying our Symmetria system and then give an overview of its design, followed by details on how it transforms queries to operate over encrypted data using our schemes. Lastly, we introduce query optimizations based on the properties of our schemes applied by Symmetria at query transformation to improve performance.

5.3.1 Symmetria Threat Model

Symmetria aims to preserve data confidentiality in the presence of a semi-honest adversary. We assume the adversary has access to all cloud nodes and can observe data and query executions, including any intermediate results. In addition, the adversary can observe data in transit between the cloud nodes and between the cloud nodes and the trusted application driver (described below). We assume the adversary cannot make changes to the queries, results, or data stored in the cloud, and consider integrity and availability attacks as well as access patterns and side-channel attacks out of scope. These attacks are orthogonal to our work and have been studied extensively in previous works. We plan to extend `Symmetria` to handle such attacks as part of our future work, using ideas from existing works [43, 73, 92–95].

`Symmetria` uses a set of PPE schemes proposed in previous work for comparisons over encrypted data. Some of these schemes, e.g., Order-Preserving Encryption (OPE) and deterministic (DET), have been shown to provide lower security guarantees and even reveal partial plaintext [19–22]. We plan to replace these schemes with more recent ones that offer semantic security [23] when their implementations become public.

Another concern specific to our encryption schemes SAHE and SMHE is the information leakage due to lack of strict compactness. Specifically, an adversary that observes the result of a query encrypted under SAHE or SMHE can infer the number of rows used to generate the result, e.g., how many items were added together to generate an aggregated sum. We note that an adversary that can observe the execution of queries, as well as any intermediate results (such as the adversary assumed in this work and by many previous PHE-based systems [25–27]), can infer the above information even when the homomorphic encryption schemes used are compact. But for a weaker adversary that can only observe the final results of the queries, the ciphertext size of our schemes can reveal the number of rows used to generate the final result. Under this weaker adversarial model, compact homomorphic schemes such as Paillier and ElGamal hide this information. Note that in some cases, our compaction techniques can hide the exact number of items used to generate a ciphertext, such as through the application of list aggregation (Section 5.2.4) and telescoping (Section 5.2.4).

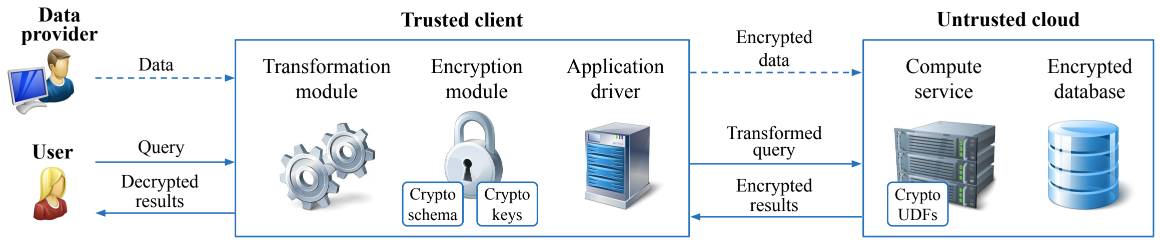


Fig. 5.1.: `Symmetria` system architecture. Dashed arrows indicate setup phase. Solid arrows indicate query execution phase.

5.3.2 `Symmetria` Design Overview

Figure 5.1 shows the high-level design of `Symmetria` which includes several components divided between the trusted client and the untrusted cloud:

Transformation module: transforms a query designed to operate over plaintext data to a query that operates on encrypted data to preserve data confidentiality.

Encryption module: holds the cryptographic keys, encrypts traffic sent to the cloud and decrypts traffic received from the cloud. Also stores the crypto schema which contains the available encryptions of each column in the database.

Application driver: deploys user-submitted queries to the cloud after they have been transformed. Receives encrypted results from the cloud, decrypts them using the encryption module and returns the results to the user.

Compute service: a distributed computing framework that executes the submitted queries in the untrusted cloud. Utilizes a set of cryptographic user-defined functions (UDFs) to perform operations over encrypted data.

Encrypted database: holds the encrypted data.

`Symmetria` operates in two phases. During the setup phase when `Symmetria` is deployed, the data provider submits the plaintext data along with associated schema to the trusted client. The data provider also identifies operations expected to be performed on each column, following the approach used in previous work [96]. Using this information, the encryption module (1) encrypts each column under one or more encryption schemes according to the operations expected for it, (2) replaces the original column name with a random string for anonymization, and (3) uploads the encrypted data to the cloud. The encryption module also generates and stores a crypto schema with a mapping from each of the original column names to the resulting anonymized ones, along with the encryption scheme used to encrypt the column.

The second phase is the query execution phase. Users of `Symmetria` are oblivious of the underlying encryption strategy used to preserve data confidentiality and do not know the structure of the encrypted database. Users, therefore, submit queries designed to operate over plaintext data and assume that the database structure follows the original schema as submitted by the data provider during setup. When a query is submitted, the transformation module intercepts and transforms it into one that operates over encrypted data. To transform queries `Symmetria` follows the approach of previous works [27,36,68]. Firstly, the query is parsed into a logical plan which has the form of a directed acyclic graph (DAG) with nodes representing operations, column names, or literal values, and edges representing the flow of data in the query. For each operation in the logical plan, the transformation module considers the available schemes the involved column(s) are encrypted under. A column can either be a data column that exists in the encrypted database, in which case the available encryption schemes are kept in the crypto schema, or the column is derived from previous operations, in which case the lineage of the logical plan is used to derive the available encryption schemes of the column. If one of the available encryption schemes supports the operation currently considered, then (i) the operation in the logical plan is replaced by the equivalent homomorphic operation, (ii) the involved column names are replaced with the (anonymized) names of the appropriately encrypted columns, and (iii) literal values are encrypted using the same scheme that supports the operation. If none of the available encryption schemes support the operation, the query is split (split execution [26]) and all remaining computation in the query is performed on the trusted client, after intermediate results are decrypted.

Once query transformation completes, the transformed query is passed to the application driver which deploys it in the cloud. The compute service in the cloud is an unmodified distributed computing framework able to perform homomorphic operations on encrypted data by means of cryptographic UDFs. These UDFs do not contain any sensitive information and are submitted to the compute service as part of the deployed application. Once the query execution completes, the (intermediate) encrypted results are sent back to the trusted client, decrypted, any remaining computation is performed, and the plaintext results are returned to the user.

5.3.3 Query Optimizations

We describe a set of optimizations, specific to SAHE and SMHE, applied to queries by the transformation module.

Expression re-writing To improve performance for expressions involving multiple SAHE or SMHE homomorphic operations we introduce a set of re-writing rules (see Table 5.2 for SAHE and Table 5.3 for SMHE). The null operations category shows rules that replace expressions with null values during query transformation. Identity operation rules replace expressions that have no effect on the input ciphertext with the ciphertext itself. The operation replacing category replaces one expression with another equivalent expression with the goal of generating more uniform expressions that can be further optimized. Addition re-ordering and multiplication re-ordering rules re-arrange expressions to enable the application of other re-writing rules. Constant folding and constant factoring rules simplify operations involving constants. Term factoring rules replace an add operation with an mlp operation or remove an add operation altogether and similarly, term simplification rules remove or replace a mul operation with a pow operation.

Our re-writing rules go against traditional compilation. As, in general, plaintext multiplication is more expensive than addition [97], compilers replace integer multiplication with constants with consecutive additions or shift operations. E.g., $2 \times x$ is replaced by $x + x$ (verified using gcc v7.3.0, Ubuntu v18.04). In contrast, in SAHE mlp is preferred over add as can be seen in rule category term factoring (Table 5.2); similarly for SMHE pow is preferred over mul as shown in the rule category term simplification (Table 5.3).

Operation pipelining A query can include expressions involving multiple homomorphic operations. For each operation on ciphertexts of the form $c = \langle v, l_p, l_n \rangle$, a modulo arithmetic computation is performed on v and, if the operation involves two ciphertexts, the two sets of id lists need to be merged, creating larger lists. Furthermore, any associated data-structures needed to enable compaction techniques (Section 5.2.4), e.g., data-structures used to hold cardinalities of ids, also need to be initialized and populated to create the resulting ciphertext. Finally, depending on the use case, the resulting ciphertext for each operation in the expression might have to be serialized.

To avoid repeated memory allocation and the creation of objects that hold intermediate results `Symmetria` pipelines all homomorphic operations in expressions, thereby amortizing their cost. All ciphertexts in an expression are collected and all homomorphic operations are applied in one go so that only data-structures for final ciphertexts need to be created. E.g., in the expression $c = \mathsf{add}(\mathsf{add}(\mathsf{add}(c_1, c_2), c_3), c_4)$, instead of creating intermediate ciphertexts for each add operation, all three operations are pipelined. Enough memory is pessimistically (assuming no compaction techniques apply) allocated to hold the data of all 4 input ciphertexts c_1–c_4, and only the final ciphertext object c is created and serialized.

Operation pipelining is particularly effective in aggregation functions. Instead of deserializing and aggregating a single ciphertext value at a time into the intermediate aggregated result, a small number of serialized ciphertexts are kept in a cache. Once

Table 5.2.: Re-writing rules for SAHE. † indicates that order of operands is not relevant.

Category	**Before**	**After**
Null operations	$\mathsf{add}(\varnothing^A,\varnothing^A)$, $\mathsf{adp}(\varnothing^A, m)$, $\mathsf{mlp}(\varnothing^A, m)$, $\mathsf{mlp}(c, 0)$	$\varnothing^A$
Identity operations	$\mathsf{add}(c,\varnothing^A)^\dagger$, $\mathsf{adp}(c, 0)$, $\mathsf{mlp}(c, 1)$	c
Operation replacing	$\mathsf{neg}(c)$	$\mathsf{mlp}(c, -1)$
	$\mathsf{sub}(c_1, c_2)$	$\mathsf{add}(c_1, \mathsf{neg}(c_2))$
Addition re-ordering	$\mathsf{add}(\mathsf{add}(c_1, c_2), c_1)^\dagger$	$\mathsf{add}(\mathsf{add}(c_1, c_1), c_2)$
	$\mathsf{add}(\mathsf{adp}(c_1, m), c_2)^\dagger$	$\mathsf{adp}(\mathsf{add}(c_1, c_2), m)$
Constant folding	$\mathsf{adp}(\mathsf{adp}(c, m_1), m_2)$	$\mathsf{adp}(c, m_1 + m_2)$
	$\mathsf{mlp}(\mathsf{mlp}(c, m_1), m_2)$	$\mathsf{mlp}(c, m_1 \times m_2)$
Constant factoring	$\mathsf{add}(\mathsf{mlp}(c_1, m), \mathsf{mlp}(c_2, m))$	$\mathsf{mlp}(\mathsf{add}(c_1, c_2), m)$
Term factoring	$\mathsf{add}(c, c)$	$\mathsf{mlp}(c, 2)$
	$\mathsf{add}(\mathsf{mlp}(c, m), c)^\dagger$	$\mathsf{mlp}(c, m + 1)$
	$\mathsf{add}(\mathsf{mlp}(c, m_1), \mathsf{mlp}(c, m_2))$	$\mathsf{mlp}(c, m_1 + m_2)$

a threshold is reached, all are deserialized, aggregated together in one go as shown in the example above, and then added to the intermediate aggregation result.

Pre-computing PRNs To encrypt a plaintext value or decrypt a ciphertext, both SAHE and SMHE require to generate one or more PRNs using $F_k(r)$ for a given r. When encryption is stateful (Section 5.2.3), r is chosen in an incremental fashion and

Table 5.3.: Re-writing rules for SMHE. † indicates that order of operands is not relevant.

Category	Before	After
Null operations	$\mathsf{mul}(\varnothing^M,\varnothing^M)$, $\mathsf{mlp}(\varnothing^M, m)$, $\mathsf{pow}(\varnothing^M, m)$, $\mathsf{pow}(c, 0)$	$\varnothing^M$
Identity operations	$\mathsf{mul}(c,\varnothing^M)^\dagger$, $\mathsf{mlp}(c, 1)$, $\mathsf{pow}(c, 1)$	c
Operation replacing	$\mathsf{inv}(c)$	$\mathsf{pow}(c, -1)$
	$\mathsf{div}(c_1, c_2)$	$\mathsf{mul}(c_1, \mathsf{inv}(c_2))$
Multiplication re-ordering	$\mathsf{mul}(\mathsf{mul}(c_1, c_2), c_1)^\dagger$	$\mathsf{mul}(\mathsf{mul}(c_1, c_1), c_2)$
	$\mathsf{mul}(\mathsf{mlp}(c_1, m), c_2)^\dagger$	$\mathsf{mlp}(\mathsf{mul}(c_1, c_2), m)$
Constant folding	$\mathsf{mlp}(\mathsf{mlp}(c, m_1), m_2)$	$\mathsf{mlp}(c, m_1 \times m_2)$
	$\mathsf{pow}(\mathsf{pow}(c, m_1), m_2)$	$\mathsf{pow}(c, m_1 \times m_2)$
Term simplification	$\mathsf{mul}(c, c)$	$\mathsf{pow}(c, 2)$
	$\mathsf{mul}(\mathsf{pow}(c, m), c)^\dagger$	$\mathsf{pow}(c, m + 1)$
	$\mathsf{mul}(\mathsf{pow}(c, m_1), \mathsf{pow}(c, m_2))$	$\mathsf{pow}(c, m_1 + m_2)$

is therefore known before a request to encrypt a value is made. In this case, to speed up encryption, `Symmetria` pre-computes and stores a small amount of such PRNs and uses them when a value needs to be encrypted. Similarly, decryption can be sped up by pre-computing $F_k(r)$ if the value r is known.

Plaintext multiplication The mlp operation that allows multiplication between a ciphertext and a plaintext value is supported by both SAHE and SMHE. At query transformation, `Symmetria` chooses the most appropriate scheme for mlp depending on what other operations the corresponding column is involved in. This allows

Symmetria to perform as much computation as possible over encrypted data in the cloud and prevents returning to the trusted client before strictly necessary. E.g., in the expression $2\times(a+b)$ SAHE is used to perform the multiplication with the constant 2 as $a+b$ involves an addition that is only supported by SAHE. On the other hand, in the expression $2 \times a \times b$ SMHE is used so the entire expression can be executed in the cloud. If there are no conflicting operations and either of the schemes can be used, to perform mlp, Symmetria uses SMHE as it is more performant (Section 5.5.3).

5.4 Implementation

In this section, we provide implementation details on our schemes SAHE and SMHE, as well as our Symmetria[1] system.

5.4.1 Encryption Schemes

We implemented SAHE and SMHE and all associated compaction techniques in Java and using AES as PRF F_k. Each ciphertext triplet $\langle v, l_p, l_n \rangle$ includes an 8-byte integer value v and two id lists l_p and l_n which are implemented using immutable arrays. We use a separate hashmap to map ids to their cardinalities. To keep this map small, we assume a default cardinality of 1 and only keep track of higher cardinalities. To compress the list of ids we implemented a custom compression function that implements the compaction technique discussed in Section 5.2.4. We further compress the arrays of ids as described in Section 5.2.4 using the JavaFastPFOR[2] library v0.1.12 and enable the compression codecs for differential coding, variable byte and fast PFOR (patched frame-of-reference). To speed up serialization/deserialization we implemented custom serialization functions for both SAHE and SMHE ciphertext objects. We also incorporate 3 PPE schemes, namely deterministic AES (ECB mode), OPE [17, 46], and SWP [18], as well as two asymmetric PHE schemes, namely Pail-

[1]https://github.com/ssavvides/symmetria

[2]https://github.com/lemire/JavaFastPFOR.

lier [5] and ElGamal [10], which we use to compare against SAHE and SMHE. For Paillier and ElGamal we use 2048-bit long keys and use the Java BigInteger class for arbitrary-precision arithmetic computations.

5.4.2 Trusted Client

The trusted client contains the transformation module, encryption module, and application driver, and is therefore deployed in a trusted node. We implemented the transformation module by extending Apache Spark's Catalyst optimizer. By default, Catalyst applies a set of transformation rules to create a logical plan out of the query, optimize it, and turn it into an executable physical plan [98]. We created a set of transformation rules by extending the Spark Rule[LogicalPlan] class and use these rules to carry out the transformation of the query to one that operates over encrypted data. The query optimizations proposed in Section 5.3.3 are also implemented as transformation rules. We register these rules with the Catalyst optimizer externally, i.e., without modifications to Spark, so that they are recursively applied to all nodes of the logical plan. We implement the application driver by extending the Spark driver. The application driver deploys the transformed query in the cloud and receives the encrypted results. The driver loads these results into a Scala parallel collection with default parallelization. The results are then decrypted in parallel and any remaining computation (see Section 5.3.2) is performed in parallel before final results are returned.

5.4.3 Untrusted Cloud

The compute service runs an unmodified Apache Spark service. To enable the use of homomorphic operations of the schemes employed we created a Spark UDF function for each operation. For non-aggregation operations, we created a Spark UDF by extending the Spark UserDefinedFunction class; for aggregation functions such as summation and product we extend the UserDefinedAggregateFunction class. These functions are again externally registered.

5.5 Evaluation

In this section, we evaluate the expressiveness and performance of our proposed schemes SAHE and SMHE as well as our prototype system `Symmetria`. We first assess the expressiveness of our schemes compared to previous schemes (Section 5.5.2). Then, we evaluate the performance of individual operations of our schemes (Section 5.5.3) and quantify the effect of scheme compactness compared to other compact and non-compact schemes (Section 5.5.4). We then examine the encryption time and ciphertext size overhead of `Symmetria` compared to previous systems (Section 5.5.5). Finally, we assess the performance benefits of our proposed compaction techniques and query optimizations (Section 5.5.6) and the overall efficiency of `Symmetria` by comparing its end-to-end latency with other systems (Section 5.5.7).

5.5.1 Evaluation Setup

We conduct all our experiments on Amazon EC2. We use a cluster of 10 m5.2xlarge (8 vCPUs, 32 GB RAM) slave nodes running Apache Spark v2.4.0 and deploy the Spark driver on a separate m5.4xlarge (16 vCPUs, 64 GB RAM) node. The Spark driver is assumed to be trusted and handles the decryption of the results. Decryption is parallelized and utilizes all available CPUs on the driver. We set up Spark to deploy a single executor per slave node with 8 tasks each (1 for each CPU) and a total of 28 GB memory. We load data into Hadoop HDFS (Hadoop v2.9.2) with replication factor 3 and use the Apache Parquet storage format to compress data. We carry out comparisons of primarily 3 system setups:

`Plaintext` setup does not use any encryption and does not offer any confidentiality guarantees,

Table 5.4.: Expressiveness comparison. *Type* indicates whether a scheme is symmetric (sym) or asymmetric (asym).

(a) AHE

	Paillier	**ASHE**	**SAHE**
Type	asym	sym	sym
add	✓	✓	✓
adp	✓	✗	✓
mlp	✓	✗	✓
neg	✓	✗	✓
sub	✓	✗	✓

(b) MHE

	ElGamal	**SMHE**
Type	asym	sym
mul	✓	✓
mlp	✓	✓
pow	✓	✓
inv	✓	✓
div	✓	✓

`Symmetria` utilizes our proposed schemes SAHE and SMHE to carry out arithmetic operations over encrypted data, and

`Asym` utilizes the asymmetric schemes Paillier and ElGamal to carry out arithmetic operations over encrypted data.

We use two industry-adopted benchmarks for our evaluations. 1) TPC-H is a decision support benchmark comprising 22 queries. We use TPC-H v2.18 at a scale of 100 which uses over 100 GB of input data. 2) TPC-DS is a benchmark for big data decision solutions comprising 100 queries. Due to lack of space, we show the results of 19 of the TPC-DS queries as done in previous work [60]. We use TPC-DS v2.10 at a scale of 100 which uses 38.6 GB of plaintext input data.

Unless stated otherwise, reported execution times in our experiments are the average of 5 executions.

Table 5.5.: Operation execution times of SAHE compared to asymmetric schemes. All reported times are given in *nanoseconds* followed by the *relative standard error*. Values in parentheses indicate pre-computation.

	Paillier	Packed Paillier	SAHE
enc	17285376 ± 0.13%	880921 ± 0.11%	1321 (63) ± 1.43%
dec	16390295 ± 0.01%	781727 ± 0.01%	1202 (153) ± 4.18%
add	34807 ± 1.37%	1666 ± 1.21%	457 ± 3.10%
adp	917141 ± 2.38%	104775 ± 0.95%	71 ± 0.37%
mlp	857943 ± 2.54%	–	406 ± 0.18%
neg	1370859 ± 0.07%	–	397 ± 0.11%
sub	1408870 ± 0.08%	–	819 ± 3.88%

5.5.2 Scheme Expressiveness

Before conducting an empirical evaluation, we first show a simple comparison of our schemes with previous homomorphic schemes in terms of expressiveness. Table 5.4 shows the supported operations of various (a) AHE and (b) MHE schemes. As can be seen from the table, SAHE supports all homomorphic operations that Paillier supports though being symmetric. ASHE [38] is another AHE scheme that is symmetric like SAHE, but unlike SAHE, it has limited expressiveness and only supports additions between two ciphertexts (add). SMHE supports all homomorphic operations that ElGamal supports though being symmetric. We know of no other symmetric MHE scheme to compare against SMHE.

5.5.3 Operation Execution Times

We now examine the execution times of individual operations in the sets Π_{SAHE}, Φ_{SAHE}, Π_{SMHE}, and Φ_{SMHE}. We compare these against the equivalent operations

Table 5.6.: Operation execution times of SMHE compared to asymmetric schemes. All reported times are given in *nanoseconds* followed by the *relative standard error*. Values in parentheses indicate pre-computation.

	ElGamal	SMHE
enc	8700278 ± 0.04%	2974 (752) ± 0.29%
dec	4768193 ± 0.02%	3090 (1420) ± 0.23%
mul	25803 ± 0.16%	419 ± 0.92%
mlp	678 ± 1.17%	371 ± 0.11%
pow	505675 ± 2.53%	2856 ± 0.37%
inv	809711 ± 0.09%	3529 ± 0.24%
div	841260 ± 0.14%	4172 ± 0.25%

of the state-of-the-art asymmetric schemes Paillier and ElGamal. Paillier encryption and its associated homomorphic operations require arbitrary-precision arithmetic operations and modulo operations on the square of the public key. This means that the length of Paillier ciphertexts are 4096 bits long. ElGamal ciphertexts are made of two parts, each 2048 bits long. The increased ciphertext size and arbitrary-precision arithmetic operations required to achieve the homomorphic operations of Paillier and ElGamal lead to significant overhead. We also include a comparison against "Packed Paillier" which implements the packing technique discussed in Section 3.4. We pack 21 plaintext values in a ciphertext and include 32 bits of padding to account for overflows.

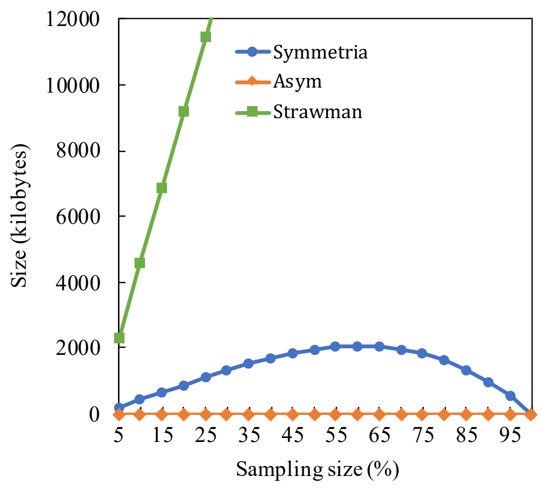
Table 5.5 and Table 5.6 show the results of this evaluation. This experiment was run on a single m5.2xlarge node. We use a random 8-byte integer value as input for the encryption operation. We use a ciphertext generated by encrypting an 8-byte integer value for the decryption and for unary homomorphic operations. Two such ciphertexts are used for binary homomorphic operations. Each reported execution time is the average of 1 million executions after 100 warm-up executions. We observe that SAHE's encryption and decryption are 4 orders of magnitude faster than Paillier's. These can be made even faster by pre-computing the required PRNs as explained in Section 5.3.3.

Packed Paillier improves the performance of encryption and decryption significantly, but SAHE remains 2 orders of magnitude faster than Packed Paillier. Performing a homomorphic addition using Paillier requires a multiplication of the two ciphertexts modulo the square of the public key which is a 4096-bit value. This makes homomorphic addition of SAHE 76× faster than Paillier or 4× faster than Packed Paillier. The performance gains of other operations in Φ_{SAHE} compared to Paillier are even more dramatic because in Paillier these operations require exponentiation modulo the square of the public key which is a very expensive operation. We note that our current implementation of Packed Paillier does not support the operations mlp, neg, and sub. Due to this limitation, we do not use Packed Paillier for the rest of our evaluations.

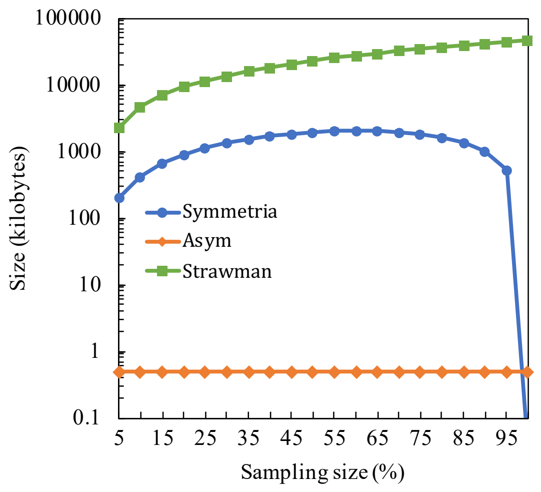
We observe similar results when comparing SMHE to ElGamal. Encryption and decryption functions of SMHE are 3 orders of magnitude faster than ElGamal. Arithmetic operations in Φ_{SMHE} range from 2× faster to 230× faster compared to the equivalent operations with ElGamal.

5.5.4 Effect of Non-Compactness

Next, we examine how the lack of compactness in our schemes affects ciphertext size and execution time and how effective our compaction techniques are in mitigating these effects. We perform a summation involving 1 million rows and to examine how our schemes behave when some of these rows are filtered out, we randomly sample the rows and perform a summation over the selected rows. We measure the size of the resulting ciphertext and the overall time taken to perform the summation as we change the sampling size from 5% to 100%.



(a) Linear scale



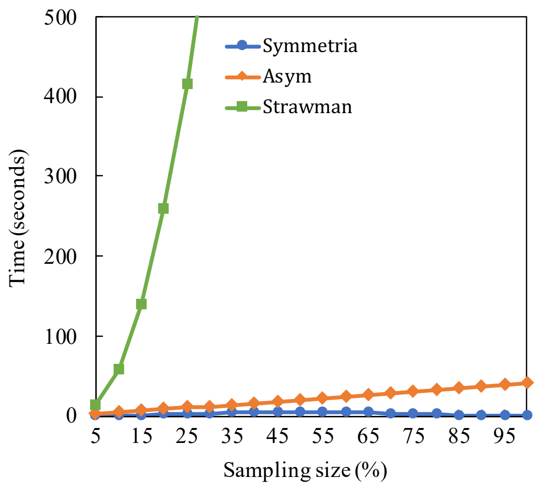
(b) Log scale

Fig. 5.2.: Ciphertext size for summation of 1 million rows as sampling size (x-axes) changes from 5% to 100%

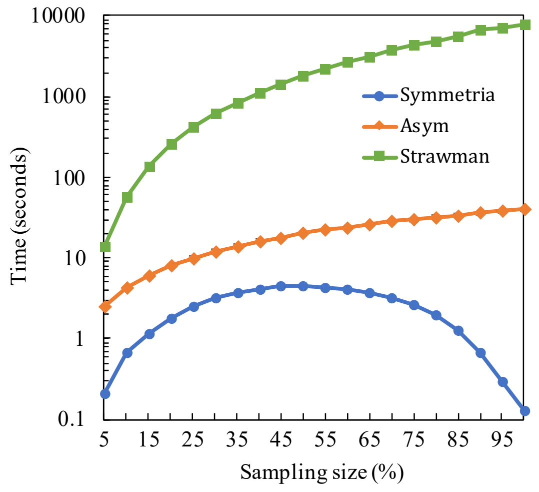
We compare `Symmetria` with all techniques enabled against `Asym` which uses asymmetric schemes that are compact. To examine the other extreme we also compare `Symmetria` against a `Strawman` setup which uses a naïve construction for arithmetic operations that encrypts individual values using the AES (CBC mode) block cipher and performs homomorphic operations by concatenating operators and encrypted operands. Unlike `Symmetria`, `Strawman` has no compaction techniques to limit ciphertext expansion.

Figure 5.2 and Figure 5.3 show the results of this evaluation. We observe that the ciphertext size of the `Strawman` system setup increases linearly as the number of rows (sampling size) increases due to lack of any form of compaction. Even worse, its execution time increases exponentially since every subsequent addition operation occurs on a strictly larger aggregated result.

As expected, the ciphertext size of the `Asym` system setup remains constant (512 bytes) since it uses Paillier (with a 2048-bit modulo resulting in 4096-bit ciphertexts) to perform summation which is a compact scheme. Naturally, its execution time



(a) Linear scale



(b) Log scale

Fig. 5.3.: Time for summation of 1 million rows as sampling size (x-axes) changes from 5% to 100%

increases linearly with respect to sampling size, but due to the complex modulo operations involved in homomorphic addition the overall execution time is high.

Unlike `Strawman`, the compaction techniques of `Symmetria` keep the ciphertext size more bounded in practice, resulting in a much lower execution time. We also note that due to non-compactness, the ciphertext size of `Symmetria` is higher than that of `Asym`. At 50% sampling size, half of the rows are randomly filtered out making the range folding technique (5.2.4) less effective. Once the sampling size increases further, more sequences of consecutive ids occur, leading to more compact ciphertexts and therefore better execution times. At 100% selectivity, `Symmetria` achieves optimal compactness with the ciphertext becoming smaller than that of `Asym` (50 bytes compared to 512 bytes for `Asym`). Despite the overall higher ciphertext size of `Symmetria`, the execution time of `Symmetria` remains on average 30$\times$ faster than the `Asym` system. We further investigate the effect of increased ciphertext size in `Symmetria` and the associated network and decryption time overheads in Section 5.5.7.

Table 5.7.: Encryption overheads. `Plaintext` (text) indicates uncompressed data. All other setups use Parquet to store compressed data. Time column refers to compression time for `Plaintext`, and adds encryption time for other setups.

Benchmark	**System Setup**	**Size**	**Time**
TPC-H	`Plaintext` (text)	106.8 GB	–
	`Plaintext`	34.0 GB	2.4 min
	`Asym`	363.7 GB	84 min
	`Symmetria`	67.8 GB	14 min
TPC-DS	`Plaintext` (text)	38.6 GB	–
	`Plaintext`	15.1 GB	1.5 min
	`Asym`	482.4 GB	228 min
	`Symmetria`	39.7 GB	4 min

5.5.5 Encryption Overhead

Next, we assess the time needed to encrypt the input data for the TPC-H and TPC-DS benchmarks and examine the size of the resulting encrypted data. We show the results of this evaluation in Table 5.7. TPC-H at scale 100 generates 106.8 GB of uncompressed data. We load this data into HDFS using the Parquet format which generates a total of 34 GB of compressed data and takes 2.4 min to complete the compression. We then encrypt the data using the two setups `Asym` and `Symmetria`. Under `Asym`, the size of the encrypted data is 363.7 GB which is over 5$\times$ larger than the encrypted data generated by `Symmetria`. In addition, `Asym` takes 84 min to complete encryption compared to 14 min with `Symmetria`. TPC-DS uses 38.6 GB of uncompressed data which, when compressed using Parquet, occupies 15.1 GB. Encrypting with `Asym` results in 482.4 GB of encrypted data which is a size overhead of 32$\times$ compared to plaintext data. In comparison, `Symmetria` generates only 39.7 GB

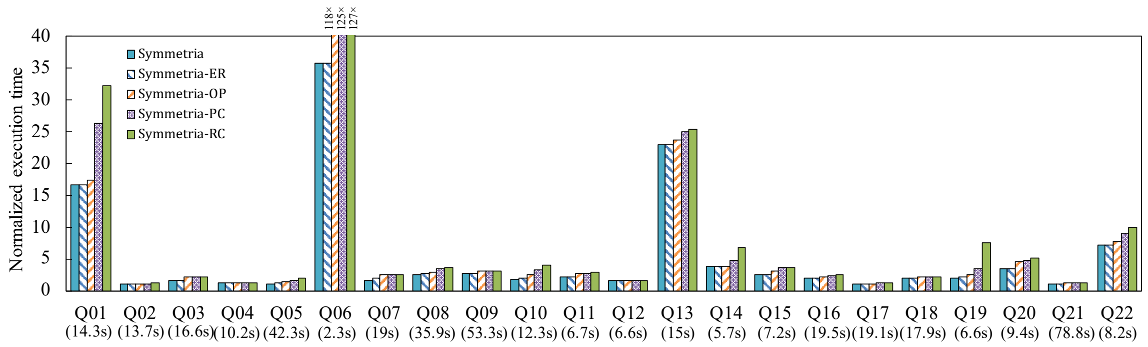


Fig. 5.4.: TPC-H end-to-end execution times normalized to `Plaintext` execution (slowdown factor). `Plaintext` execution times in absolute are shown in parentheses under the query names (x-axis). We cut the y-axis off at 40× slowdown (reporting numbers above) to focus on details of our `Symmetria` solution.

of encrypted data with a size overhead of 2.6×. Encryption time with `Asym` takes 228 min compared to only 4 min with `Symmetria`.

5.5.6 Effect of Compaction Techniques

We use the TPC-H and TPC-DS benchmarks to evaluate the benefits of our proposed compaction techniques and query optimizations. Figure 5.4 shows the results of this experiment for TPC-H. To be able to evaluate secondary homomorphic operations that operate on a ciphertext value and a plaintext value such as adp and mlp we choose 4 columns out of a total of 61 columns that do not seem to hold sensitive data, namely the columns *tax*, *discount*, *available-quantity*, and *supplier-quantity* and leave them in plaintext.

The `Symmetria` bars show the execution times of `Symmetria` with all compaction techniques and query optimizations enabled, normalized to the `Plaintext` execution time (`Plaintext` execution times in absolute are shown in parentheses under the query names (x-axis)). Each subsequent `Symmetria` bar shows the normalized execution time with one less feature enabled. Specifically, `Symmetria-ER` shows the

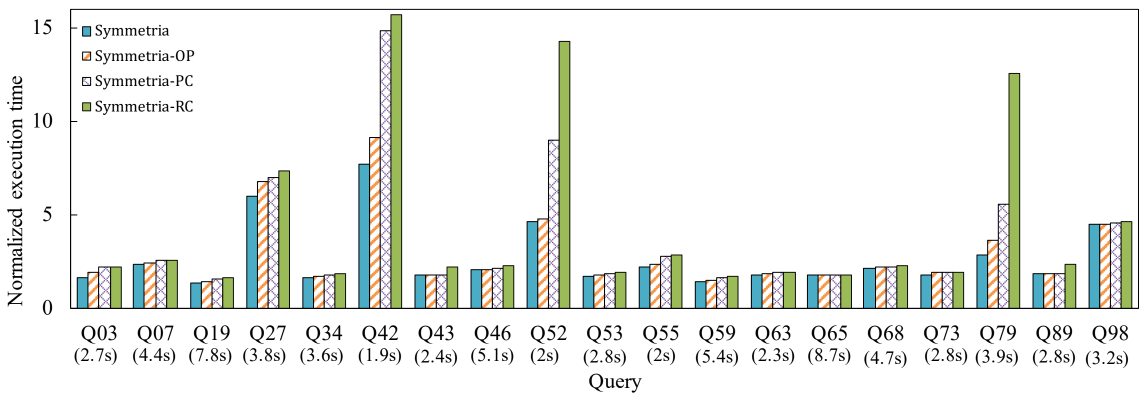


Fig. 5.5.: TPC-DS (subset) end-to-end execution times normalized to `Plaintext` execution (slowdown factor). `Plaintext` execution times in absolute are shown in parentheses under the query names (x-axis). We cut the y-axis off at 15× slowdown to focus on details of our `Symmetria` solution.

normalized execution time when expression re-writing (Section 5.3.3) is disabled, `Symmetria-OP` shows the normalized execution time when *in addition* to expression re-writing, operation pipelining (Section 5.3.3) is disabled, in `Symmetria-PC` PFOR compression (Section 5.2.4) is disabled and finally, in `Symmetria-RC` range compression (Section 5.2.4) is disabled. We keep some compaction techniques permanently enabled — list aggregation (Section 5.2.4), id grouping (Section 5.2.4), and telescoping (Section 5.2.4) — as without these techniques the id lists of ciphertexts become too large and many of the queries time out. The average execution time overheads compared to plaintext execution time of `Symmetria`, `Symmetria-ER`, `Symmetria-OP`, `Symmetria-PC` and `Symmetria-RC` are 5.35×, 5.40×, 9.50×, 10.57×, and 11.39× respectively.

Figure 5.5 shows the results of the same experiment on 19 queries of TPC-DS. TPC-DS uses 9 input tables and a total of 166 columns. This time, we leave a single column, *sales-price*, remains in plaintext allow the use of secondary homomorphic operations. We show the breakdown of `Symmetria` with some features disabled. Expression re-writing did not apply to any of the queries. The average execution

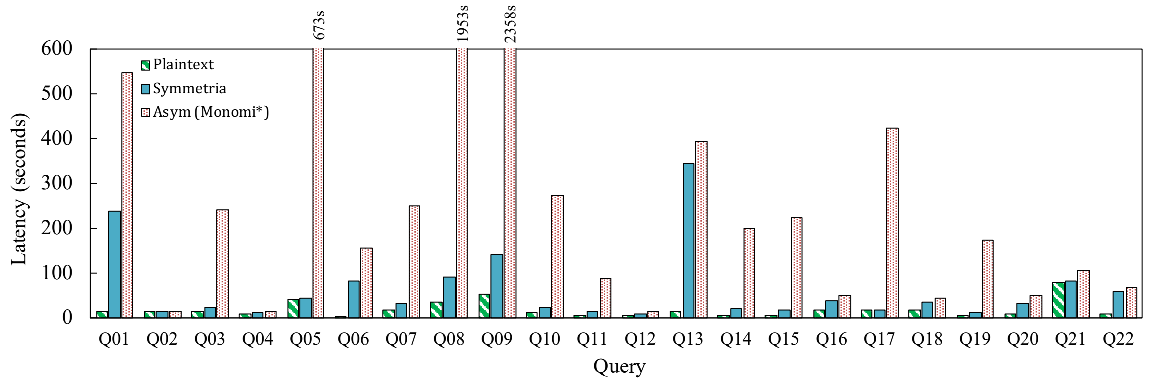


Fig. 5.6.: TPC-H end-to-end latency. We cut the y-axis off at 600 seconds (reporting numbers above) to focus on details of our `Symmetria` solution.

time overheads compared to `Plaintext` execution time of `Symmetria`, `Symmetria-OP`, `Symmetria-PC` and `Symmetria-RC` are 2.73×, 2.95×, 3.66×, and 4.45× respectively. The gains we observe in both TPC-H and TPC-DS demonstrate the effectiveness of our proposed techniques and optimizations.

5.5.7 Comparison to Existing Systems

Finally, we use the TPC-H and TPC-DS benchmarks to compare end-to-end execution times of `Plaintext`, `Asym`, and `Symmetria` system setups, by measuring the time from the point each query is submitted until the results are returned to the user after having been returned to the driver and decrypted. For this evaluation, `Asym` closely follows Monomi's choice of cryptosystems [26] which we further augment by an MHE scheme (ElGamal [10]) and a searchable encryption (SRCH) scheme (SWP [18]), implemented in Spark instead of the proposed PostgreSQL though. The resulting system called `Monomi*` uses split execution the same way as `Symmetria`. We leave the same columns in plaintext as in the previous experiment for both TPC-H and TPC-DS. For a fair comparison we leave the same columns in plaintext for the `Monomi*` system setup.

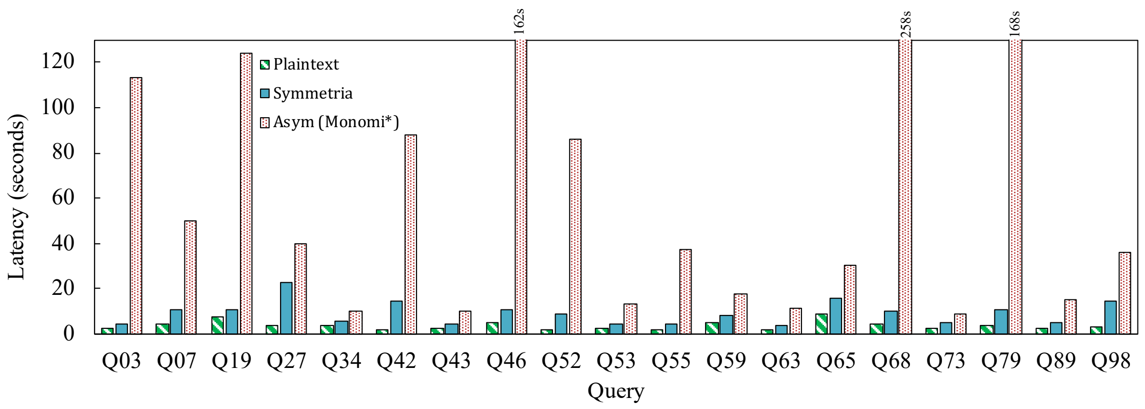


Fig. 5.7.: TPC-DS end-to-end latency. We cut the y-axis off at 120 seconds (reporting numbers above) to focus on details of our `Symmetria` solution.

Figure 5.6 shows the end-to-end latency in seconds of the aforementioned systems for TPC-H. The `Plaintext` system setup has an average end-to-end latency of 19.11 s across all queries, `Symmetria` has a 63.29 s average end-to-end latency and `Asym` has a 378.10 s average end-to-end latency. The average overhead of `Symmetria` compared to `Plaintext` is 5.35× and the average overhead of `Monomi*` compared to `Plaintext` is 20.39×. We further observe that for `Symmetria`, the bulk of the work is done in the untrusted cloud. On average, only 1% of the overall query execution is spent on the trusted client side. This time includes communication overhead and decryption of final results. A notable exception is Q22 where 12.96% of the time is spent on the client side.

Figure 5.7 shows the end-to-end latency in seconds for `Plaintext`, `Symmetria`, and `Asym` systems for TPC-DS. The `Plaintext` system setup has an average end-to-end latency of 3.79 s across all queries, `Symmetria` has a 9.30 s average end-to-end latency and `Asym` has a 67.49 s average end-to-end latency. The average overhead of `Symmetria` compared to `Plaintext` is 2.73× and the average overhead of `Monomi*` compared to `Plaintext` is 19.16×. The average time spent on client side is only 2.12% for `Symmetria`.

Overall, `Symmetria` with all compaction techniques and query optimizations enabled is on average 3.8× faster on TPC-H queries than the state-of-the-art PHE-based systems using asymmetric schemes, and 7× faster on TPC-DS queries. This demonstrates the practicality of our schemes SAHE and SMHE despite the lack of strict ciphertext compactness and shows the effectiveness of our proposed optimizations.

5.6 Related Work

In this section we discuss PHE-based systems and systems based on trusted hardware related to `Symmetria`.

5.6.1 PHE-based Systems

As mentioned in Section 4.8, several PHE-based systems have been proposed that show how PHE can be made practical for confidentiality-preserving computations. CryptDB [25] extends a MySQL database to enable the execution of SQL queries in a confidentiality-preserving manner by using PHE. CryptDB uses Paillier to perform homomorphic additions but does not support any MHE scheme so multiplications cannot be performed.

Arx [23] is a PHE-based system that extends MongoDB to support confidentiality-preserving database applications. Arx introduces the ArxEq and ArxRange primitives that enable equality and range computations over data encrypted using semantically secure encryption. We plan to extend `Symmetria` to support these primitives and replace the PPE schemes currently used which offer lower security guarantees. Since the focus of `Symmetria` is to demonstrate the benefits of using our symmetric PHE schemes over asymmetric PHE schemes, in the current version of `Symmetria` we chose to use PPE schemes whose implementations are publicly available.

Seabed [38] is a system built on Apache Spark that introduced ASHE. ASHE and Seabed were designed with the goal of performing large-scale summations efficiently and have limited homomorphic expressiveness, as opposed to `Symmetria` that uses SAHE and SMHE that support a wider range of homomorphic operations.

Several techniques have been proposed to compact ciphertexts of PHE schemes [29, 99, 100]. Most prominently, Ge and Zdonik [29] describe a technique that allows packing multiple plaintext values into a single ciphertext, thereby amortizing the ciphertext size overhead. The technique applies to asymmetric AHE schemes but not to symmetric AHE schemes or to MHE schemes. In addition, it introduces the issue of overflows for aggregation functions and limits expressiveness, as homomorphic operations cannot be performed to packed values individually, nor can some homomorphic operations such as multiplication with plaintext, negation, or subtraction be performed on packed ciphertexts.

Monomi [26] is a PHE-based database system using the above-mentioned packing technique to reduce ciphertext size overhead and accelerate summations. Similarly, Liu et al. [101] employ packing to reduce the overheads of computing the trajectory similarity function over encrypted data. These works, in addition to not supporting an MHE scheme, inherit the drawbacks of the packing technique described above.

In contrast, the compaction techniques we introduce herein (Section 5.2.4) apply to both of our proposed symmetric schemes, do not limit the expressiveness of homomorphic operations, and do not suffer other drawbacks of packing multiple values into one ciphertext.

5.6.2 Trusted Hardware-based Systems

Recently, Trusted Execution Environments (TEEs) like Intel Software Guard eXtensions (SGX) [24] have also gained traction for performing secure computations in the cloud. As described in Section 4.8 works like VC3 [73], Opaque [43], and TensorScone [102], leverage SGX for ensuring data confidentiality and integrity during computation in untrusted environments. In this work, we focus on a software-only solution based on PHE and PPE that does not require specialized hardware and place emphasis on improving the performance and expressiveness of existing PHE and PPE based systems.

5.7 Conclusions

In this chapter, we tackle the important problem of preserving the confidentiality of sensitive information while executing queries on it in an untrusted cloud. We introduce two novel symmetric PHE schemes that allow us to efficiently perform operations over encrypted data. Our schemes are faster compared to existing asymmetric PHE schemes and do not compromise homomorphic expressiveness. Our system `Symmetria` utilizes these schemes to enable the execution of queries over encrypted data. `Symmetria` achieves average speedups of $3.8\times$ and $7\times$ over state-of-the-art asymmetric PHE schemes on the standard TPC-H and TPC-DS benchmarks respectively.

6. CONFIDENTIAL STREAM PROCESSING FOR THE INTERNET OF THINGS

With the advent of the Internet of Things (IoT), billions of devices are expected to continuously collect and process sensitive data (e.g., location, personal health factors). Due to the limited computational capacity available on IoT devices, the current de facto model for building IoT applications is to send the gathered data to the cloud for computation. While building private cloud infrastructures for handling large amounts of data streams can be expensive, using low-cost public (untrusted) cloud infrastructures for processing continuous queries including on sensitive data leads to strong concerns over data confidentiality.

In this chapter we present `C3PO`, a confidentiality-preserving, continuous query processing engine, that leverages the public cloud. The key idea is to intelligently utilize Partially Homomorphic and Property-Preserving encryption to perform as many computationally intensive operations as possible — without revealing plaintext — in the untrusted cloud. `C3PO` provides a simple abstraction to the developer to hide the complexities of 1) applying complex cryptographic primitives, 2) reasoning about the performance of such primitives, 3) deciding which computations can be executed in an untrusted tier, and 4) optimizing cloud resource usage. An empirical evaluation with several benchmarks and case studies shows the feasibility of our approach. We consider different *classes of IoT* end devices that differ in their computational and memory resources (from Amazon AWS to a tiny device with Cortex-M3 microprocessor) and through the use of novel optimizations we demonstrate the feasibility of using Partially Homomorphic and Property-Preserving Encryption on IoT devices.

6.1 Overview

The ubiquity of computing devices is driving a massive increase in the amount of data generated by humans and machines. With the advent of the IoT, many more billions of devices are expected to continuously collect sensitive data and compute on it, promising improvements in various sectors. For instance, improvements in sensors and increasingly practical wearable devices allow complex, automatic and real-time health monitoring [103]. Such monitoring is beneficial by providing patients direct information on their current health status, facilitating diagnosis and treatment, and reducing costs of interventions and risks.

6.1.1 Cloud-backed IoT and Confidentiality

Due to limited storage and computation capacity available on IoT devices, the current de facto model for building IoT applications is to send the data gathered from physical devices to the cloud for both computation and storage. Many IoT applications therefore leverage the cloud to compute on data streams from a large number of devices, leading to the paradigm of Cloud of Things (CoT). For example, in the case of smart health, health care providers can closely monitor a larger number of patients and correlate data on a bigger scale. If privacy concerns are addressed, individuals may be more open to sharing their data [104], which in turn is critical for contact tracing applications and to identify pandemics or epidemics early-on.

Due to the sheer amount of streaming data, building a private cloud infrastructure is very expensive compared to using a low cost public (untrusted) cloud infrastructure such as Amazon EC2 or Microsoft Azure. Therefore, public clouds are typically used for processing continuous queries including on sensitive data. Public clouds are also preferred because of the variety of software services they provide that make development and deployment of corresponding applications very fast. However, this trend is fueling concerns over data confidentiality, and is becoming one of the major factors preventing further widespread adoption of IoT solutions. These concerns

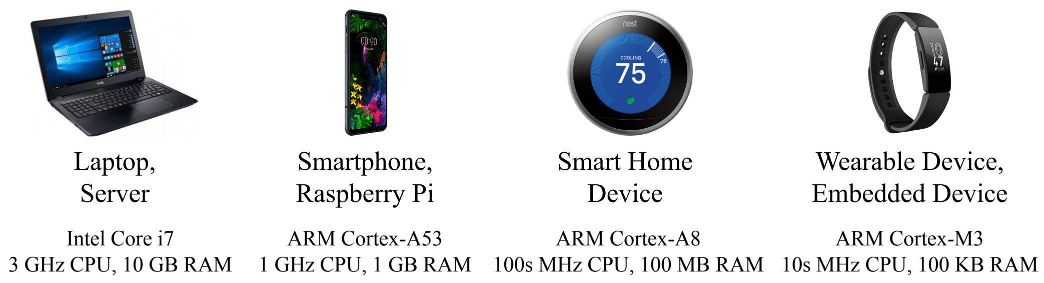


Fig. 6.1.: IoT devices of various compute and memory capabilities supported by `C3PO`

represent a significant deterrent for industry domains like healthcare to adopt public clouds.

One way to mitigate these concerns is to encrypt data at the source (i.e., IoT devices), and solely use cloud infrastructure for storage purposes (e.g., Bolt [105]). Thus, as long as encryption keys are maintained securely by consumers, the confidentiality of their data is enforced. While this approach addresses the above confidentiality concerns, all computations need to be performed in trusted environments, thus limiting the computational capabilities of public clouds for IoT solutions.

A promising approach to overcome this bottleneck is to use *homomorphic encryption* and execute all operations over encrypted data. However, Fully Homomorphic Encryption (FHE) is prohibitive in practice [3], causing significant slowdowns for complex computations despite continuous advances [4, 33]. An alternative, practical approach is to use less expensive Partially Homomorphic Encryption (PHE) [106] in combination with Property-Preserving Encryption (PPE) [107] to execute specific operations over encrypted data.

Existing solutions based on PHE and PPE have mostly focused on database and batch processing systems. For instance, the seminal CryptDB [25] was implemented on top of MySQL, Monomi [26] was implemented on top of the Postgres database while Crypsis [27] was implemented in Hadoop (Apache Pig) and Cuttlefish [84], and Symmetria [108] were implemented in Apache Spark. These database-centric and batch processing solutions are not a good fit for many IoT applications since IoT applications are typically implemented as continuous queries in a stream processing system.

In addition IoT devices can vary significantly in terms of their compute power and memory capacity as shown in Figure 6.1. To enable devices which can be very resource-constrained (e.g., embedded devices using the ARM Cortex-M3 CPU can have as little as 72 MHz of processing power and 64 KB of memory) to encrypt sensitive data under various PHE and PPE schemes, these schemes must be carefully implemented and optimized.

6.1.2 Challenges

A straightforward application of PHE and PPE to existing stream processing solutions to support computations over encrypted data is unlikely to be practical:

R1. ***Cryptosystems on IoT devices.*** IoT devices are oftentimes resource-constrained with limited amount of processing power and memory capacities. The ability to support PHE and PPE cryptosystems on IoT devices in an efficient manner is a major concern.

R2. ***Complexity of cryptosystems.*** Dozens of PHE and PPE *schemes* exist, varying by operations supported, efficiency, ciphertext size overhead, etc.; IoT application developers such as health care app engineers do not necessarily possess sufficient knowledge of cryptosystems and their properties to judiciously select among these.

R3. ***Complexity of queries and data.*** Analytical continuous queries can become quite complex, leading to intricate intertwining and combining of data items throughout lengthy sequences of processing stages. Tracking of data lineages becomes complex, yet it is necessary to determine which PHE or PPE schemes need to be applied to initial input data.

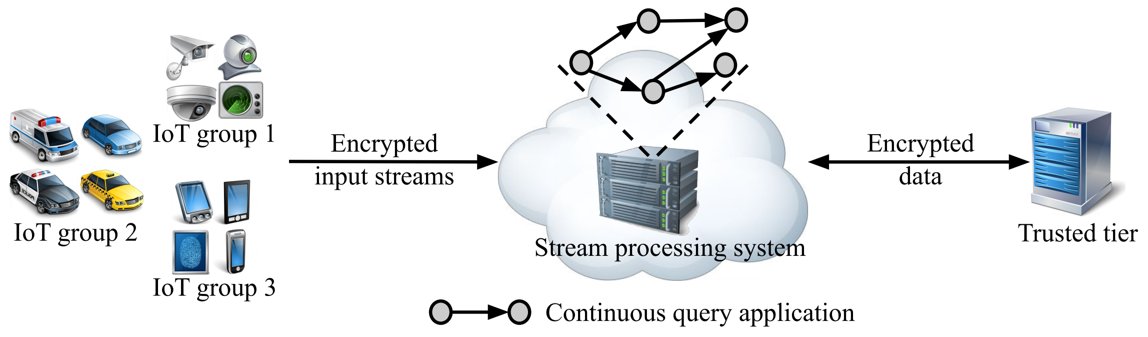


Fig. 6.2.: C3PO design overview.

R4. ***Application variables & constants.*** Variable initialization and constant values in queries must be carefully handled to preserve the confidentiality requirements.

R5. ***Inherent limitations.*** As hinted to by their names, PHE schemes do not support arbitrary operations. To overcome this limitation, either the query processing can continue on the trusted client side after the intermediate results have been decrypted or alternatively, the intermediate results can be re-encrypted to the schemes required by subsequent operations, after which computation can proceed in the cloud.

R6. ***Key compromises.*** With applications running continuously and potentially indefinitely, secret keys used for encryption need to be updated periodically or on demand (e.g., when a device such as a health monitor is compromised). Such updates on IoT devices should be made transparently to the IoT application and should not cause disruptions to the execution of continuous queries or lead to missing results.

6.1.3 `C3PO`

This chapter presents `C3PO` (<u>C</u>loud-based <u>C</u>onfidentiality-preserving <u>C</u>ontinuous Query <u>P</u>r<u>o</u>cessing)[1], a novel managed runtime system, that leverages PHE and PPE to provide confidentiality for IoT applications delegating online streaming jobs to the public cloud. `C3PO` operates on streaming data without revealing any plaintext information to the untrusted cloud. Figure 6.2 gives a high-level overview of the design of `C3PO`. A user designs, implements, and initiates the continuous query application that runs in the untrusted cloud, using the `C3PO` steam processing system. IoT devices automatically encrypt generated data before emitting them as part of streams for analysis. These devices can optionally be assigned to different groups for added security as discussed in Section 6.5. Additional streams of encrypted private data required for analysis can be sent independently from a trusted tier maintained by the user.

In short, to perform analytics in the untrusted cloud over encrypted data while at the same time addressing R1-R6, `C3PO`:

- introduces PHE and PPE optimization techniques (e.g., field masking, pre-computation, ciphertext packing, caching and speculative encryption) to reduce encryption time and ciphertext size overhead and provides efficient implementation of these techniques so they can run on IoT devices (R1);
- enables programmers to develop applications using the `C3PO` API for typical *plaintext* streams and automatically transforms the application to work with encrypted streams. Developers can focus on the application logic and not on the details of the underlying cryptosystems, nor which specific cryptosystem to use for each part of the application (R2, R3) or how to handle variable initialization and constant encryption (R4);

[1]In the *Star Wars*™ saga, C-3PO is a risk-averse droid with a strong need for security and stability.

- is capable of continuing computation in the trusted tier or re-encrypting (parts of) a data stream to enable further computation in the public cloud if a given sequence of computations cannot be performed due to PHE limitations (R5);
- supports transparent periodic and on-demand update of secret keys on IoT devices and enables partitioning of data spaces (IoT groups) to reduce key sharing (R6).

6.1.4 Contributions and Outline

In this chapter we thus make the following contributions. We

1. introduce a secure stream abstraction that gives rise to a high-level API through which programmers can express confidentiality-preserving analytics programs that can be executed efficiently in public clouds, without having to know the details of underlying cryptosystems.
2. present the `C3PO` system that implements this API and addresses related challenges, building on the well-known Apache Storm[2] system. `C3PO` analyzes programs written using the `C3PO` API and identifies computations that can be executed purely on encrypted data and computations that, due to the limitations of PHE, cannot. `C3PO` maximizes the amount of computation performed in the cloud by splitting computation between the untrusted cloud and a small number of trusted nodes (trusted tier).
3. present extensions and optimizations to existing PHE and PPE schemes to make them more practical for use in resource-constrained devices.
4. present key management schemes that limit breaches in cases of key compromises by partitioning data in an application-aware manner, and by supporting on-the-fly key changes without disrupting the execution of continuous queries.

[2]`http://storm.apache.org`

5. propose a heuristic that analyzes resource availability and requirements and generates a deployment profile that optimizes cloud resource usage.

6. evaluate the implementation of C3PO on multiple benchmarks and case studies. Our results indicate that C3PO can be used to express many real-world IoT applications while ensuring confidentiality transparently and keeping low overhead.

The remainder of this chapter is organized as follows. We present background information on continuous queries in Section 6.2. We give an overview of our solution in Section 6.3 including the assumed threat model and architecture of C3PO. We present the programming model of C3PO and its runtime in Section 6.4. In Section 6.5, we discuss key management in C3PO. We discuss implementation details of C3PO in Section 6.6. We present our evaluation results in Section 6.7. Section 6.9 presents a conclusion to this chapter.

6.2 Background

In this section, we discuss relevant details of systems that support continuous queries.

6.2.1 Continuous Queries

The core abstractions offered by systems that support continuous queries are *streams*, *tuples*, and *fields*. Streams are unbounded sequences of tuples. Each tuple within a stream contains one or more fields with each field having an associated name. Values of each field can be accessed by dereferencing a tuple by the field name or the field index. Tuples in a same stream have the same set of fields (with distinct values). The tuples in the stream are processed in a distributed fashion. Application logic is arranged as a directed graph where vertices of the graph are computation components and edges are streams that represent the data flow between components.

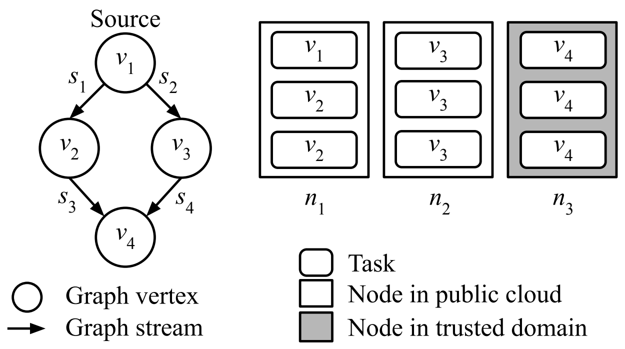


Fig. 6.3.: `C3PO` graph and tasks.

Application programmers write application logic for vertices of the graph. A subset of these vertices are also designated as *source vertices*. Source vertices act as entry points for data into the graph. Source vertices typically read data from a queue, log file, or external subscriptions. As data is generated in real time and added to a queue, it is picked up by source vertices and forwarded down the graph for processing according to a grouping clause (described below).

Figure 6.3 shows an example graph with four vertices with v_1 designated as the source vertex. The figure also shows streams $s_1 - s_4$. Each vertex of the graph may have multiple runtime instantiations called *tasks*. Vertex v_1 has one task running on public node n_1 (a *node* represents a virtual or physical machine), v_2 has two tasks running on public node n_1, and v_3 has three tasks running on public node n_2. Finally, v_4 has three tasks running on a trusted node n_3. We refer to this assignment, i.e., the specific number of tasks per vertex, as the *deployment profile* of the graph.

The stream emitted by each vertex is declared explicitly in the vertex itself. Once all vertices of the graph are designated, the graph is assembled by defining the input stream of each vertex and specifying the *grouping clause* of each stream. A *grouping clause* defines how tuples in a stream are partitioned among the tasks of a vertex that receives the stream. This grouping clause is also part of the definition of the graph and is provided by the programmer. Common grouping clauses are: a) *shuffle grouping* – tuples are distributed randomly across tasks in such a way that each task gets an equal number of tuples, b) *field grouping* – tuples are partitioned according to a designated field and distributed among tasks, and c) *all grouping* – the stream is replicated across all tasks.

6.3 C3PO Design

In this section, we give an overview of C3PO's main components. We first introduce C3PO's threat model and discuss some assumptions about the IoT devices used in C3PO. We then present C3PO's program abstractions and runtime execution flow.

6.3.1 Threat Model

The goal of C3PO is to preserve data confidentiality in the presence of a semi-honest adversary. We assume the adversary has access to the cloud nodes and can observe data residing in the nodes as well as the execution of applications and any generated intermediate results. We assume that the adversary cannot make changes in the queries, results or data stored in the cloud and consider integrity and availability attacks to be out of scope for our system.

We also consider physical IoT device compromises out of the scope but assume that IoT devices have a memory-protected area where secret keys can be stored and the application logic as well as the encryption scheme implementation reside in a tamper-proof ROM. With these security measures we first assume that IoT devices are not compromised and focus on preserving the confidentiality of data in the untrusted cloud. We later relax this requirement and show how encryption keys of IoT devices can be updated in case of a compromise (Section 6.5). We assume that C3PO has access to a limited set of trusted resources outside the cloud (e.g., where the query results are used). As we will see shortly, this environment is leveraged to perform a few specific computations.

C3PO preserves data confidentiality by utilizing a set of encryption schemes. As previously mentioned, some of these encryption schemes such as OPE and DET schemes are deterministic and are known to provide lower security guarantees or even reveal partial plaintext [20–22]. `C3PO` minimizes the security implications of using these schemes by issuing a warning to the programmer when the application requires to use OPE or DET, giving the option to the programmer to either deploy the parts of the query that require OPE or DET operations on the trusted resources at the expense of performance, or deploy the application as is, an option which could be viable if the data requiring OPE or DET holds semi-sensitive or high entropy information such as timestamps. As part of our future work, we plan to replace the OPE and DET schemes `C3PO` currently uses with recently introduced schemes that offer semantic security [23] as soon as their implementation becomes available.

6.3.2 Assumptions

We present the assumptions about IoT devices and their connectivity considered in `C3PO`.

IoT device classes We assume that IoT devices used in `C3PO` are of the *C2* class or higher (see RFC7228 [109], "Classes of Constrained Devices") with at least 50KB of RAM and CPU operating at frequency of at least a few 10s of MHz. A large range of IoT devices are included in this category, such as the ones shown in Figure 6.1. In addition, we assume that IoT devices are in the *E9* class of energy limitation (RFC7228 [109], "Classes of Energy Limitation") with no direct quantitative limitations to available energy. We plan to incorporate battery-powered IoT devices with limited energy capacity and examine the effect encryption has on battery life as part of our future work.

Key sharing Managing encryption keys for IoT devices in a distributed setting is a challenging problem [110, 111]. With no trusted party, keys can be pre-distributed by being hardwired into the device. Such hardwired keys can then be used to establish new secret keys [112]. In `C3PO` we consider a case where all IoT devices are owned by a single party (for example, a healthcare provider that issues health monitoring IoT devices). IoT devices are connected to the Internet and are capable of establishing a secure, authenticated channel to the device owner (key manager), allowing keys to be updated using standard protocols (TLS [113]) following the approach of previous work [114].

6.3.3 PHE and PPE for IoT Devices

To ensure the confidentiality of data, sensitive information needs to be encrypted at source, before sent to the public cloud for processing. Therefore, data is generated and encrypted on IoT devices. Oftentimes, encryption of PHE and PPE schemes is a computationally expensive operation and a straightforward use of these schemes on IoT devices with limited resources is unlikely to be practical. In the following paragraphs, we discuss how the set of extensions and optimizations presented in Chapter 3 can be used to reduce the time and space overhead associated with PHE and PPE encryption, making these schemes more practical for use in resource-constrained devices, thereby addressing challenge R1. `C3PO` uses the same encryption schemes used in `Cuttlefish` (see Table 4.2) with the exception of the ASHE cryptosystem. We provide details about their implementation in Section 6.6. In Section 6.7.2, we evaluate PHE and PPE on a range of IoT devices in the C2 class and higher (Section 6.3.2) with the most constrained device having a 72 MHz CPU and 64 KB RAM, which corresponds to wearable devices or embedded appliances.

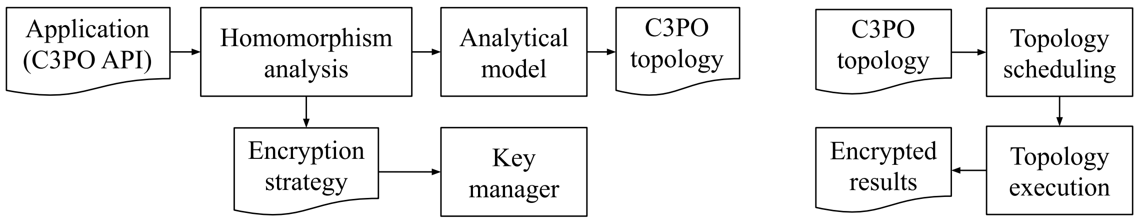
Post-encryption packing `C3PO` uses post-encryption packing (Section 3.4) which is particularly useful for continuous queries applications, since these applications often retain aggregated values over long periods of time, e.g., to keep track of daily, weekly, monthly or yearly statistics. These aggregated values can be packed together as a single ciphertext value. We demonstrate the effect of post-encryption packing in Section 6.7.4.

PRN pre-computation Oftentimes IoT devices take measurements sparsely and have plenty of idle time in between measurements. In `C3PO`, IoT devices use this otherwise idle time to perform PRN pre-computation (Section 3.5) and store a small number of pre-computed components for Paillier (Equation 3.13) and ElGamal (Equation 3.15). As shown in Section 6.7.2 pre-computation has a drastic improvement on encryption performance.

Ciphertext caching IoT devices in `C3PO` use ciphertext caching (Section 3.6) to store a map of plaintext-ciphertext value pairs for deterministic encryption schemes. The number of items that the map can hold is configurable and adjusted depending on the memory capacity of each IoT device.

Speculative encryption Similarly, IoT devices in `C3PO` use speculative encryption (Section 3.7) to encrypt and store a small number of plaintext-ciphertext value pairs proactively, during times that IoT devices are idle. Speculative encryption is more effective when the range of possible values is small, such as when measuring temperature in a closed environment.

Format-preserving encryption `C3PO` uses format-preserving encryption (Section 3.8) to keep the ciphertext size overhead small. Keeping the ciphertext size overhead small leads to smaller end-to-end latency because a smaller volume of data needs to be transmitted from the IoT devices to the cloud nodes for processing. But beyond this, using format-preserving encryption to keep ciphertext size smaller is particularly useful when this optimization is used in combination with the caching and speculative encryption optimization described above. Since `C3PO` supports devices with as little as 64KB of memory, having ciphertexts of smaller size allows IoT devices to retain a larger number of cached ciphertexts in memory.



(a) Compilation phase (b) Deployment phase

Fig. 6.4.: C3PO execution flow

6.3.4 Program Abstractions

One of the main challenges of computing over encrypted data is that the application developer needs to have a detailed understanding of each cryptosystem used to encrypt fields of a stream. Adoption of PHE and even FHE for generic application development will depend on the ease with which a programmer can incorporate the properties offered by the cryptosystem into their regular programming tasks. C3PO tackles this problem by offering simple programming abstractions to express and operate on encrypted data streams. Fields representing sensitive data are defined using the `SecField` abstraction, irrespective of the kind of operation that needs to be performed on them or the underlying cryptosystem used to represent the field, thereby relieving the programmer from the task of identifying the cryptosystem required for the operation that needs to performed. To this end, a programmer simply annotates the stream with the desired operation and C3PO deduces the cryptosystem that needs to be used at the source IoT devices. We give more details about the C3PO API in Section 6.4.

6.3.5 Execution Flow

Figure 6.4 outlines the steps followed by C3PO to set up and deploy an application securely in an untrusted cloud. Application programmers use the C3PO API and associated annotations to describe a graph which contains the application logic. C3PO then performs *homomorphism analysis* on the graph to generate an *encryption strategy*, containing the cryptosystems required to execute the graph in a confidential manner. The encryption strategy is then passed to the *key manager* that generates keys for each cryptosystem as described in Section 6.5 and sends them to the IoT devices using a secure channel. Next, C3PO analytically identifies the number of tasks required for different vertices and schedules the graph for execution. C3PO leverages the idea that oftentimes users have some limited (but trusted) computing resources available. We refer to these resources as the *trusted tier*. The compute resources in the cloud, though potentially unlimited for practical purposes, are untrusted. C3PO utilizes the trusted tier for application development and compilation and uses the cloud for the deployment phase. In deployments that require resources from the trusted tier, C3PO tries to minimize their usage. The deployment steps are detailed in Section 6.4.3.

6.4 C3PO Stream Processing

In this section, we describe the programming abstractions used in C3PO and explain how these abstractions are used in addressing challenges R2-R5 and leveraged for improving performance.

6.4.1 Programming Model

Application programmers use the abstractions provided by our C3PO Java API to specify the C3PO graph. Each vertex in the graph is designed as a separate class by extending the `C3POVertex` class. The stream emitted by each vertex is declared explicitly in the vertex itself. Once all vertices are designed, the graph is put together by defining the input stream of each vertex and specifying the grouping clauses.

Abstractions Next, we explain the abstractions that are new to C3PO over typical stream processing systems such as Storm:

`C3POVertex`: This base class is extended by programmers to express the computation in a vertex of the graph. The class provides the `execute()` method which is invoked by `C3PO` when a tuple containing `SecField`s arrives at a vertex for processing.

`SecField`: Programmers use this class to realize the abstraction of secure field that refers to a confidential input field. Programmers can get a reference to a `SecField` value in a tuple using the `SecField.getField()` method or by reading an encrypted value directly from a stream.

`SecOper`: The secure operator class provided by `C3PO` allows programmers to express standard operations such as `add`, `subtract`, `multiply`, `divide`, `compare`, `equals`, and `match`. Both primary and secondary operations listed in Table 4.2 are exposed through functions of the `SecOper` class. These functions take `SecField`s (or a `SecField` and an `int` value for secondary operations) as input and return a `SecField` for arithmetic operations, a Boolean value for equality comparisons and string matching operations or an integer (-1, 0, or 1) for order comparisons.

`@encOperations`: Programmers can also annotate each stream with the operations they want to perform on that stream through the `@encOperations` annotation. These annotations enable our compile-time graph analysis to identify the suitable cryptosystems for performing these operations and to apply additional performance improvement techniques, introduced shortly in Section 6.4.2, without requiring a modified compiler.

Example Listing 6.1 shows a code snippet used in a `C3PO` vertex class extending the `C3POVertex` class (Line 3). The code is part of a graph that keeps track of the sum of values in different groups within a sliding window — last 60 seconds (`numSlots = 60`) in this example, as indicated by Line 4. The input tuple contains two fields: the group name and the value for that group. The code shown retrieves the group and value fields from the input tuple (Lines 9, 10) and updates the sum for that group's current time slot (Line 11) with the value. Note that the annotation

```
1  // Code executed when a vertex receives a tuple
2  @encOperations(operations = {"{eq}", "{sum}"})
3  public class C3POSumVertex extends C3POVertex {
4    SlotBasedSum<SecField> slidingWinGroupSums = new
   SlotBasedSum<SecField>(60);
5    public void execute(Tuple tuple) {
6      //if timing tuple ...
7      emitCurrentWindowCounts(slidingWinGroupSums);
8      //else ...
9      SecField group = SecField.getField(tuple, 0);
10     SecField value = SecField.getField(tuple, 1);
11     slidingWinGroupSums.updateSum(group, getCurTimeSec(), value);
12   }
13 }
14
15 // Class that keeps track of sum of values per group in each time
   slot
16 public class SlotBasedSum<T>  {
17   ...
18   public void updateSum(T group, int slot, SecField value) {
19     SecField[] sums = groupSums.get(group);
20     if (sums == null) {
21       sums = new SecField[this.numSlots];
22       groupSums.put(group, sums);
23     }
24     sums[slot] = SecOper.add(sums[slot], value);
25   }
26 }
```

Listing 6.1: `C3PO` code for finding the sum of each group in a sliding window.

`@encOperations(operations = {"{eq}", "{sum}"})` in Line 2 indicates to the compiler that the first field of the steam is used in equality comparisons and the second field in summing. Every time the vertex receives a timing tuple, signifying a minute

```
1  public class SlotBasedSum<T> {
2    ...
3    PaillierPK publicKey = readPaillierPubKey();
4    public void updateSum(T group, int slot, BigInteger value) {
5      BigInteger[] sums = groupSums.get(group);
6      if (sums == null) {
7        sums = new BigInteger[this.numSlots];
8        groupSums.put(group, sums);
9      }
10     sums[slot] = (sums[slot] == null) ? value :
11             sums[slot].multiply(value).mod(publicKey.N.pow(2));
12   }
13 }
```

Listing 6.2: Code for finding the sum of each group in a sliding window without `C3PO` abstractions.

has elapsed (code omitted), it emits the sum of all groups in the current sliding window (Line 7). The object maintaining the sliding window internally contains a map and updates the group's sum every time the `updateSum()` method is called using `C3PO`'s `SecOper.add()` method (Line 24).

Listing 6.2 shows just the function `updateSum()` from Listing 6.1 written without using `C3PO` abstractions. As can be seen, this example includes several implementation complexities and requires the programmer to a) know that Paillier is the correct cryptosystem to use for performing additions, b) explicitly read the Paillier public key (Line 3) which contains the generator, g and the modulus, N, and c) perform the exact computation ψ (see. Equation 2.1) for homomorphic addition with Paillier — multiplication modulo the square of the modulus N of the public key (Line 11) — including handling of `null` values (Line 10). These implementation complexities are not specific to summation and the Paillier cryptosystem. For example, ciphertexts of the ElGamal cryptosystem contain two components and homomorphic multiplication of two ciphertexts is achieved by multiplying the two components of the ciphertexts consecutively to generate the encrypted result. Similarly, equality and order comparisons over encrypted data require non-trivial computations over the ciphertexts. The `C3PO` API hides all these implementation complexities from the application programmer.

6.4.2 Processing Secure Streams

We now give details on how we tackle the challenges R2-R5 introduced in Section 6.1.2 when processing continuous queries over encrypted streams.

Identifying encryption schemes (R2, R3**)** This step identifies the cryptosystems that are required for the various fields based on the operations that the application wishes to perform on those fields. To apply these inferences, `C3PO` first has to identify different streams and their grouping clauses in the application logic. These can be derived from the graph declaration provided by the application programmer as explained in Section 6.4.1. Secondly, `C3PO` derives the operations performed on each stream from program annotations (`@encOperations`) in each vertex class in the graph.

Once `C3PO` derives the distinct streams and operations to be performed on those streams, we can proceed similarly as in our prior work [36] to infer the cryptosystems required to execute the graph. In brief, we start by constructing an expression tree where fields in tuples form the leaf nodes. Operations performed on those fields form the non-leaf nodes. For `C3PO` graphs, we use field annotations (as specified in Section 6.4.1) to determine the non-leaf, operator nodes. For each operator, a lookup table identifies the cryptosystem of the operands and result of the operator. Our goal now is to identify the cryptosystem in which all the leaf nodes (fields) should be encrypted. This can be done by identifying the parent operator node for each leaf node and using the lookup table to identify the type of cryptosystem required for operands for that operator node.

Handling public streams (R3**)** Sometimes a program may involve plaintext data from public streams, such as stock quotes, as part of an application logic. Even though this data is public it can still be used in combination with private, encrypted streams to carry out useful computations. `C3PO` achieves this by allowing vertices to receive tuples containing both encrypted (`SecField`) and plaintext data. Through the use of the secondary homomorphic operations described in Section 2.2, `SecField`s and plaintext data can be combined without compromising the confidentiality of sensitive data.

Field masking (R2) Computing on encrypted data introduces the additional challenge of dealing with operands with increased sizes. For instance, an addition of two 32-bit `int` operands in plaintext may transform into an operation over 4096-bit encrypted operands in the encrypted data stream. This means a factor of 128 increase in the operand size. Typically, in stream processing systems, the source vertex receives all the fields in the stream, irrespective of a field being used or not. For plaintext program graphs, this is usually not a substantial overhead and the additional computation required for removing unused fields may not always offset the improvement that is observed. When the computation happens over encrypted data, filtering out fields has a much more significant impact because of the size of the fields. For example, consider a stream with two fields similar to the stream used for `group by...sum` in Listing 6.1. The first field is encrypted under a deterministic (DET) scheme and occupies 16 bytes, and the second field is encrypted under Paillier and occupies 512 bytes. If there is a continuous query which finds unique groups, the second field will be unused. Simply removing the unused field reduces the size of a tuple from 528 bytes to 16 bytes. `C3PO` performs this unused field removal automatically using field masking. Since an unused field may be at any index within a tuple, if we simply drop the field, program logic that accesses other fields using their indices may fail. To avoid this problem, during compile time `C3PO` keeps track of the indices of unused fields and appropriately adjusts all other indices that appear in program logic. To identify unused fields, `C3PO` relies on the stream annotations described in Section 6.4.1. For each vertex, we identify fields for which no operations are specified. Our masking process itself is very lightweight. Since we have information about fields to be masked at compile time, we update the `C3PO` runtime with this information. `C3PO` then suppresses the emission of the masked fields.

Initialization and constants (R4) Oftentimes, application logic requires variables to be initialized to a specific value, say α. To preserve confidentiality, value α cannot remain in plaintext and should instead be encrypted under the appropriate cryptosystem during program compilation. To identify the appropriate cryptosystem for encrypting α, `C3PO` first identifies the operation and the `SecField` that α is used in. Then `C3PO` uses the `@encOperations` annotation to identify the cryptosystem used to encrypt the relevant `SecField` and uses the same cryptosystem to encrypt α. Similarly, `C3PO` encrypts any constants in the application.

Automatic re-encryption (R5) Once the analyzer determines the cryptosystems required for each stream, it may detect situations where some operations cannot be performed over the available cryptosystems in the cloud. This can occur if there is a mismatch between parent and child operator nodes because they express operations not supported by the same cryptosystem. To get around this issue, `C3PO` can decide to either perform those operations in the trusted tier, or to re-encrypt the stream in the trusted tier. For re-encryption, `C3PO` inserts special *re-encryption vertices* into the graph and marks them so they get scheduled on the trusted tier only.

6.4.3 `C3PO` Scheduler

The primary responsibility of the `C3PO` scheduler is to decide on which host machine(s) each vertex of the graph will be executed. The `C3PO` scheduler is provided with two lists of hostnames, one that lists hosts in the untrusted cloud, and another that lists hosts in the trusted tier. The scheduler reads the graph annotation to identify where each vertex must be executed.

For components that need execution in a trusted tier, the scheduler sends the appropriate class files to the worker instances running in the trusted tier. The trusted tier workers have access to private keys that are required for encryption/decryption. The workers in the untrusted cloud only see the encrypted data and have access only to the public keys required to perform the homomorphic operations.

We note that the scheduler service can also run in the untrusted cloud. An attacker can try to manipulate the scheduler in the following two ways:

(i) by trying to execute trusted vertices in the untrusted cloud, and

(ii) by trying to execute untrusted code in the trusted tier.

The former way does not compromise confidentiality since the untrusted cloud does not possess the private keys required to reveal the plaintext data. However, the latter can compromise confidentiality if the attacker is successful in executing malicious code that retrieves private keys or read data when they are in plaintext while being re-encrypted. To avoid this, a hash of the vertices to be executed in the trusted tier is generated before deployment. When tasks are delivered to the trusted tier for execution, the trusted tier first computes a hash of the task class, and compares it with the hash generated before deployment. Execution proceeds only if the hash is verified.

6.5 IoT Key Management

To reduce the risk of secret keys being compromised in continuous query applications, `C3PO` updates keys periodically or on demand without causing disruptions to query executions. In this section we, thereby, address challenge R6, and describe key management in `C3PO`, with particular focus on replacement and sharing of keys.

6.5.1 Key Sharing

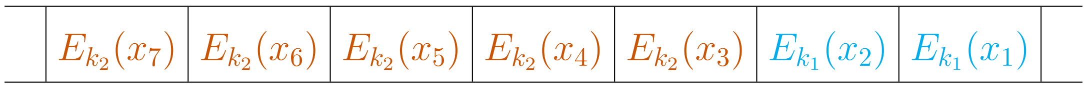
As mentioned in Section 6.3.2, we consider a setting where all IoT devices are owned by a single party and managed by a trusted key manager that distributes keys to IoT devices. In order to limit the sharing of keys across devices and over time we employ the following techniques in `C3PO`:

- Key updates: allow updating encryption keys periodically (or in the event of a key compromise) without breaking stream outputs.
- Multi-group mode: limit the number of devices that share an encryption key by splitting devices into multiple groups, each with a different key.
- Field level key identification: ensure fields that are not part of a common operation do not share encryption keys.

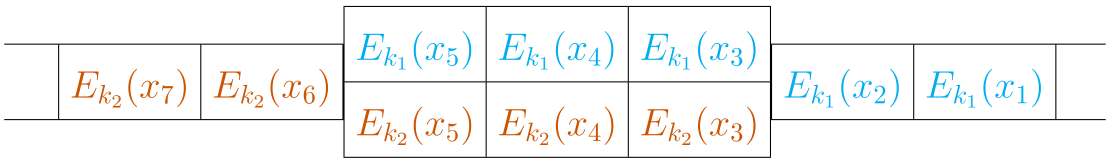
6.5.2 Key Update

One of the challenges of using homomorphic encryption for IoT-based streams is that applications are often long-lived, which increases the chance of key compromises. To mitigate this, `C3PO` allows the encryption keys to be updated, periodically or on demand. Periodic key updates could be part of a security policy. Further, if a key has been compromised, an on-demand key update is initiated. Due to the nature of continuous queries and because of the fact that homomorphic operations can only be carried out on operands encrypted with the same key, key updates are not straightforward. Therefore, we first start with an explanation of how keys are updated in the general case and then present how to handle queries that involve computations such as aggregated values or sliding windows. Then, we present a method that helps reduce the number of keys that need to be updated in case of a key compromise.

General case `C3PO` supports key changes without disrupting the output. Usually, all IoT devices are considered to form a single logical group, and share the same



(a) Naïve key update. Values in cyan (x_1, x_2) are encrypted using key k_1 and values in orange (x_3, x_4, x_5, x_6, x_7) are encrypted using key k_2. Aggregate queries will be broken because the aggregation window contains values encrypted with different keys.



(b) Key change in `C3PO`. Values x_3, x_4, and x_5 are encrypted using both old and new keys which enables `C3PO` to produce continuous output.

Fig. 6.5.: Key change in continuous queries. Streams flow from left to right (rightmost element, x_1, is oldest). After x_2 is emitted a key change is initiated.

keys. A direct consequence of this is that every key leak will result in all devices needing to update their encryption keys, a problem which we address shortly with our *multi-group mode*. When a key change is initiated by the key manager, each IoT device first emits a key change marker that includes the new key identifier. When the `C3POVertex` base class detects the key change marker in the stream, it creates a new instance of the application vertex class to process the stream. Any encrypted constants or literals involved in the computations are re-encrypted under the new key, by invoking the trusted tier, at which point computation is moved to the new instance and the old instance is abandoned.

Sliding window The general case described above does not consider queries that contain computations involving older values received via the stream such as computations involving a sliding window. An example of such computation would be a query that computes the sum of the last few received items (or similarly, the sum of a certain time interval based on timestamps). In this case, results of computations that occurred under the previous key must be included in subsequent computations.

C3PO follows the same process as before and the stream encrypted with the new key is channeled into the new instance of the vertex class but this time the stream encrypted with the old key continues processing uninterrupted. Instead, C3PO suppresses emissions from the new vertex instance until the new instance contains tuples spanning the full length of the sliding window. At this point, the old instance of the application vertex class is discarded and the stream from the new instance is emitted. Figure 6.5 illustrates this. 6.5a shows the effect of a naïve key change in an encrypted data stream. Values are first encrypted with key k_1 and then k_2. Aggregations with sliding windows that span values encrypted with both keys will fail. In 6.5b, when a key change is initiated, a specific number of values (equal to the size of the aggregation window) are encrypted with both keys k_1 and k_2. This allows aggregations to preserve semantics, i.e., avoid any disruption in output.

This solution works well as long as the sliding window is small. For large sliding windows, or for queries with aggregation functions that span the entire duration of the query, the above solution becomes inefficient, since for every key change, and as long as the sliding window does not end, another stream is added. Instead, C3PO handles this case by using the trusted tier to re-encrypt the current aggregated result under the new key. After the re-encryption step, the query can correctly handle tuples encrypted with new keys.

6.5.3 Multi-Group Mode

To reduce the surface of affected devices in the event of a key compromise, C3PO introduces the multi-group mode. In this mode, IoT devices are grouped into logical subsets and a different key is assigned to each set (where otherwise the key would be the same). IoT devices can be grouped together based on any user-defined criteria. Devices that are behind the same gateway usually make a good grouping. This allows us to update the encryption keys of devices behind a specific gateway independent of devices outside the gateway.

```java
public class C3POSumCombiner extends C3POCombiner {
  Map<SecField, SecField> mergeMap = new HashMap<SecField,
 SecField>();
  @override
  public void combine(SecField group, SecField value) {
    mergeMap.put(group, SecOper.add(mergeMap.get(group), value))
  }

  @override
  public void emit() {
    for (Map.Entry<SecField, SecField> entry : mergeMap.entrySet())
      emit(new Tuple(entry.getKey(), entry.getValue()));
  }
}
```

Listing 6.3: `C3PO` combiner implementation for aggregation in multi-group mode.

Operations across groups. At the processing end, multiple groups lead to additional tasks for `C3POVertex`. This is because when there are multiple groups using different keys, `C3PO` cannot combine results from all groups entirely in the cloud. In order to complete such queries, `C3PO` keeps track of each device group and saves query results of each group separately. The results from each individual group are then combined together to get the full result at the client site. Note that `C3PO` internally does not truly distinguish between single-group and multi-group modes; the former is simply a special case of the latter with 1 group.

Key updates So far the discussion about key updates assumed a single logical group containing all IoT devices. In this setting, in case of a key compromise, all devices need to replace the compromised key with a new one. The multi-group mode allows us to initiate key updates for any individual group. Listing 6.3 shows a `C3POCombiner` implementation for the `group by... sum` example shown in Listing 6.1. In multi-group mode, each stream with multiple groups is associated with a combiner capable of combining the results of all key groups. In this example, the combiner class overrides the `combine()` and `emit()` functions in the `C3POCombiner` base class (Lines 4 and 9). The `combine()` function sums values corresponding to the same key group, generates intermediate sums per group and stores them in a map with the group as key and the sum as value of the map. Once all values per key are combined the `emit()` function emits each key-value pair as a separate tuple. The receiver of these tuples (1 tuple per group) can finally compute the total sum by adding the intermediate sums together, after decrypting them.

6.5.4 Key Identification

For operations to be performed over encrypted data, fields involved in a same operation need to be encrypted using the same key. Inversely, fields not involved in the same operation need not use the same key, to prevent leaking relations between fields unnecessarily and to minimize the impact of a compromised key. For example, to perform the operation $x_1 + x_2$ both x_1 and x_2 need to be encrypted under the same cryptosystem that allows addition and using the same key, otherwise the operation will generate the wrong result. Separately, if we also need to perform the operation $x_3 + x_4$, then x_1 and x_2 need to be encrypted under the same key but x_3 and x_4 can be encrypted under a different key even though all four fields need to be encrypted under an AHE scheme to carry out the addition operation. We capture these field groupings by assigning fields into "field families" that indicate which fields are involved in the same operations, directly or indirectly. Following this intuition we derive two invariants that need to hold for all keys, to minimize data leaks while preserving program correctness.

I1. Correctness Fields involved in the same *kind* of operations and belonging to the same field family need to be encrypted with the same key.

I2. Security Fields involved in different *kind* of operations or (either or) belong to a different field family need to be encrypted with different keys.

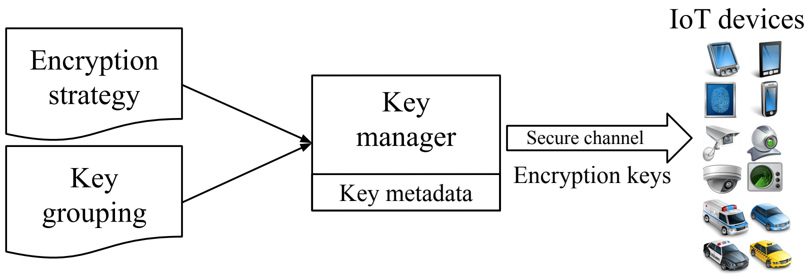


Fig. 6.6.: `C3PO` Key management

6.5.5 Key Generation

The key manager uses key group information and the encryption strategy generated during the homomorphism analysis step of the compilation to decide how to generate keys, as shown in Figure 6.6. It then associates a key to each field in a manner that satisfies invariants I1 and I2 introduced above. More specifically, keys are generated based on the equation given below:

$$K_{c,f,g} = \mathrm{PRP}_{\mathrm{MK}}(c, f, g)$$

where PRP is a pseudo-random permutation (e.g., AES block cipher) and MK is the *master key*, known only to the key manager, from which all other keys are derived. Furthermore:

Cryptosystem c: c indicates the cryptosystem the key is used for. Since the cryptosystem used indicates the operation that can be performed over encrypted data, by invariant I2, different cryptosystems should lead to different keys.

Field family f: f indicates the field family of fields. By invariant I1 fields of the same field family need to be encrypted under the the same key. For different field families, `C3PO` uses a different f which will generate a different key, even if the cryptosystem is the same.

Group g: g captures the group identifier to support multi-group mode. In the general case (single-group mode), there is simply a single all-encompassing group.

Once the keys are generated, the key manager opens a secure channel with each IoT device and sends only the keys each device requires, according to what fields each device generates and what group it belongs to. To keep track of what keys are sent to each device and to be able to identify which devices need to be sent new keys during key updates, the key manager keeps a map of key IDs per device (key metadata).

6.6 Implementation

In this section, we lay out some of the implementation details of `C3PO`.

6.6.1 Storm Integration

`C3PO`'s processes in the cloud are implemented by modifying Apache Storm. Storm is an online, distributed computation system. Application logic in Storm is packaged into directed graphs called *topologies*. Vertices of the topologies are computation components and edges represent data flow between components. There are two types of components in Storm: a) *spouts* that act as event generators, and b) *bolts* that capture the program logic. In other words, spouts produce the data streams upon which the bolts operate. Modifications to Storm are limited to implementing a new scheduler by overriding the `IScheduler` interface and to the way a Storm topology is submitted (`StormSubmitter` and related classes). These changes add an additional 1031 lines of Java code to Storm. The `C3PO` programming interface and cryptographic classes that allow computations over encrypted data (but do not include encryption/decryption functions) are packaged as a separate `jar` library, implemented in 3633 lines of Java code. The key manager is implemented in 900 lines of Java code and uses DTLS [115] to establish an end-to-end secure channel with IoT devices. Key metadata are stored in XML files and the key manager includes an XML parser to retrieve this data.

6.6.2 Cryptosystems

Our cryptosystems including the extensions and optimizations described in Chapter 3 are implemented in C[3] and accessed where necessary through Java Native Interface (JNI). Randomized encryption (RND) is implemented using the AES [116] cryptosystem in CBC mode with a random initialization vector. Deterministic encryption (DET) is implemented using an AES pseudo-random permutation block cipher with a variant of CMC mode [16] with a zero initialization vector. FNR [45] is used as an alternative DET cryptosystem to preserve the format of small values. The Boldyreva et al. [17,46] cryptosystem is used as our OPE scheme implementation. We implement an extended version of Song et al. [18] cryptosystem as our SRCH scheme that supports all 4 constructions described in Section 3.3.

We implement the Paillier [5] cryptosystem as our AHE scheme and follow the approach introduced by Damgård and Jurik [9] to set the generator $g = N + 1$ for a more optimized implementation of encryption. We also use the Chinese Remainder Theorem to optimize the decryption function of Paillier. Finally, we implement ElGamal [10] as the MHE scheme. Paillier and ElGamal require arbitrary precision arithmetic computations as part of their encryption, decryption, and homomorphic operations. We implement 3 different versions of Paillier and ElGamal, each using a different arbitrary precision arithmetic library, since not all these libraries are supported on all IoT devices. We use the GMP library [117] (version 6.1.2) and its `mpz` arithmetic primitive when available. Alternatively, we use the OpenSSL library [118] (version 1.1.1) and its `BIGNUM` arithmetic primitive. For highly resource-constrained devices that do not support GMP or OpenSSL we use the BigDigits library [119] (version 2.6) and its `BIGD` arithmetic primitive which is a very small but less optimized library.

[3]`https://github.com/ssavvides/homomorphic-c`

6.7 Evaluation

In this section, we present an extensive evaluation of `C3PO` using standard benchmarks and use cases. Our evaluation shows that `C3PO` can preserve confidentiality by executing on encrypted data with $20-30\%$ higher latency and around 23% reduction in throughput.

6.7.1 Synopsis and Overview

We use several scenarios for evaluation as follows:

Encryption latency: We first assess the feasibility of using `C3PO` with IoT devices that are resource-constrained, by analyzing the *encryption latency* of various cryptosystems and associated optimizations used by `C3PO` on IoT devices of the *C2* or higher classes (see Section 6.3.2). These classes include IoT devices in the entire range shown in Figure 6.1.

Smart meter analytics: We use the Smart* dataset [120] as our input together with queries adapted from IoTBench [121] to compare the *throughput* of `C3PO` to vanilla Storm. In this scenario volume of processed data is of primary concern. This includes an assessment of *field masking*.

Heartbeat analysis: We use a heartbeat analysis application that computes individual and group statistics. We use this application to evaluate the *latency* of `C3PO`; query response times are critical in such healthcare applications for triggering emergency responses in a timely manner. This includes an assessment of *PRN pre-computation* and *post-encryption packing*.

Yahoo streaming: We also use a more generic streaming benchmark, namely Yahoo Streaming Benchmark (YSB) [122] for further evaluating the *latency* of `C3PO`. Latency is critical in this benchmark as the goal is to react quickly to advertisements.

Multiple groups: We use a microbenchmark to analyze the effects of `C3PO`'s *multi-group* feature.

New York taxi statistics: Finally, we evaluate the costs of *re-keying* by computing statistics over a large number of nodes (devices) based on a publicly available data set [123] from New York taxis released under FOIL (Freedom of Information Law).

6.7.2 Encryption Latency

To evaluate the feasibility of our approach from the point of view of the end devices, we consider the encryption latency of various cryptosystems on different IoT devices. We use 5 devices of different computational capabilities and memory capacities:

2xl: Amazon AWS m5.2xlarge instance with 3.1 GHz CPU and 16 GB RAM.

Pi3: Raspberry Pi 3 Model B with Quad Core 1.2 GHz Broadcom BCM2837 CPU and 1 GB RAM.

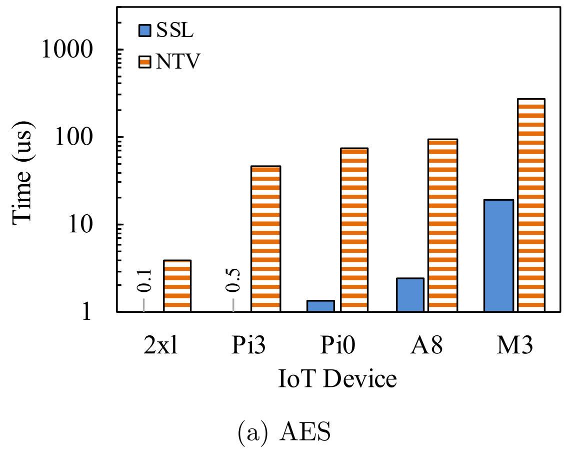
Pi0: Raspberry Pi Zero W with a 1 GHz 32 bit single-core CPU and 512 MB RAM.

A8: ARM Cortex-A8 with 600 MHz 32-bit microprocessor and 256 MB RAM.

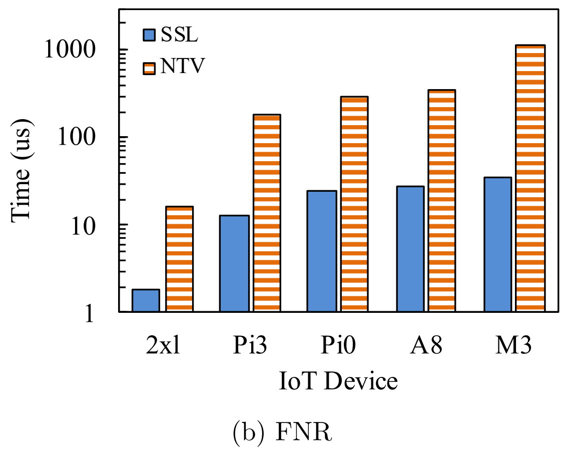
M3: ARM Cortex-M3 with a 72 MHz 32-bit microprocessor and 64 KB RAM.

We evaluate 2 PPE schemes (AES and FNR) and 2 PHE schemes (ElGamal and Paillier), each implemented under different libraries as explained in Section 6.6: 1) the **NTV** native implementation of AES[4] which is also used internally in the FNR cryptosystem; 2) the Open**SSL** library; 3) the **GMP** library; 4) the **BDS** BigDigits library. We use a 128-bit block for AES and a 2048-bit modulus for Paillier and ElGamal.

[4]`https://github.com/kokke/tiny-AES-c`

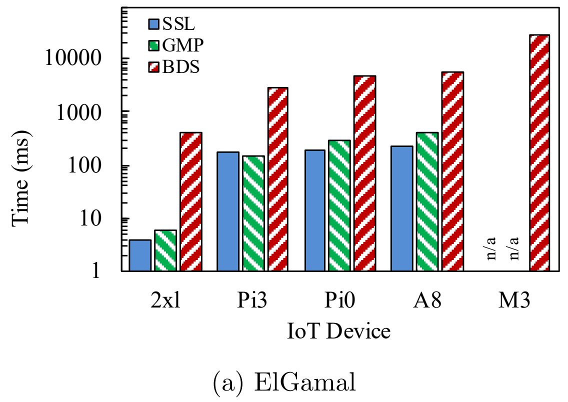


(a) AES

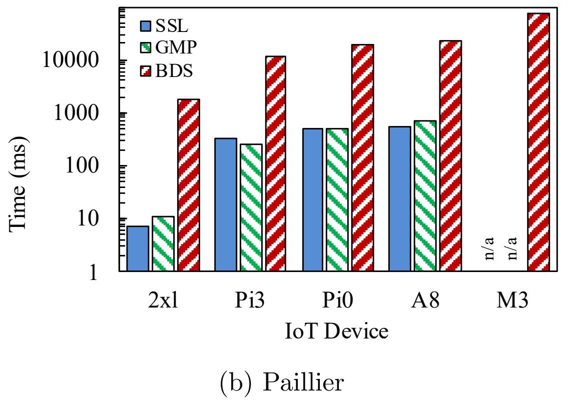


(b) FNR

Fig. 6.7.: Encryption latency of various PPE schemes (time in microseconds) across different IoT devices. y-axis in log scale.



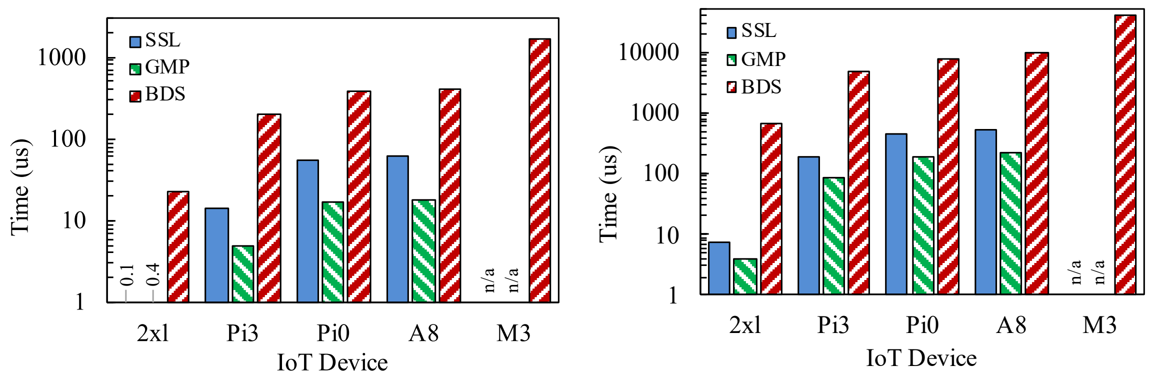
(a) ElGamal



(b) Paillier

Fig. 6.8.: Encryption latency of various PHE schemes (time in milliseconds) across different IoT devices. y-axis in log scale.

Figure 6.7 shows the execution time for encrypting a random 128-bit string for AES and FNR and Figure 6.8 shows the execution time for encrypting a random 32-bit `int` number for ElGamal and Paillier. We observe that if SSL is available, AES and FNR encryption is very fast taking only 19 us for AES and 36 us for FNR, on the most computationally constrained device, M3. If SSL is not available, the much



(a) ElGamal

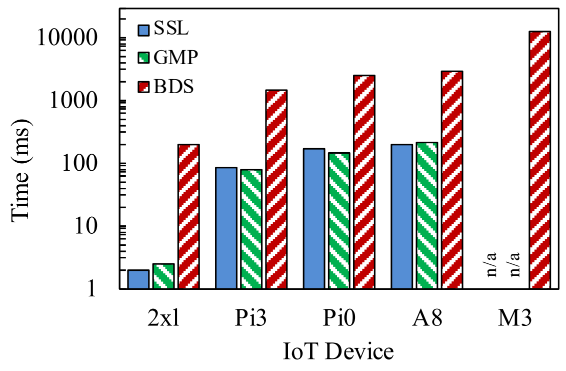
(b) Paillier

Fig. 6.9.: Encryption latency of ElGamal and Paillier with *PRN pre-computation* (time in microseconds) across different IoT devices. y-axis in log scale.

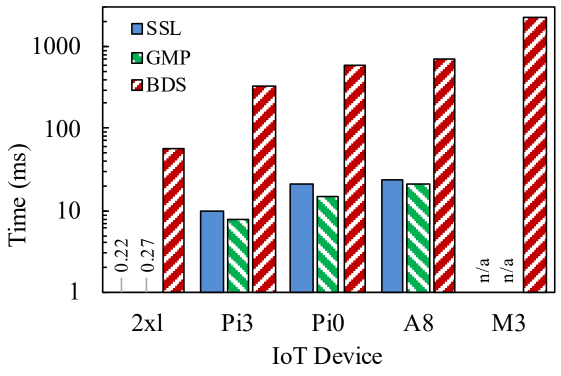
simpler but less performant NTV implementation can be used which requires 270 us for AES and 1137 us for FNR on an M3 device.

As expected, encryption using ElGamal or Paillier is a more expensive operation, since these are asymmetric schemes. Yet in all IoT devices that support the GMP or the SSL libraries ElGamal and Paillier exhibit decent performance. ElGamal encryption implemented using SSL takes 3.9 ms on 2xl, 169.6 ms on Pi3, 193.8 ms on Pi0, and 219.6 ms on A8. Corresponding times for ElGamal implemented using GMP are slightly higher. Similarly, Paillier encrypted using SSL takes 7.3 ms on 2xl, 331.5 ms on Pi3, 492.9 ms on Pi0, and 554.7 ms on A8. Due to luck of support for the GMP library and the `BIGNUM` primitive in the versions of SSL supported in the M3 device, ElGamal and Paillier cannot be implemented using GMP or SSL. Therefore, ElGamal and Paillier encryption on the M3 device implemented using the less optimized BigDigits library is impractical, requiring several seconds to complete. This justifies the need for the optimizations we propose in Chapter 3.

As shown in Figures 6.9 and 6.10 our proposed optimizations improve the performance of ElGamal and Paillier across all IoT devices dramatically. With PRN pre-computation, ElGamal encryption implemented in GMP is slightly faster than



(a) ElGamal



(b) Paillier

Fig. 6.10.: Encryption latency of ElGamal and Paillier with *ciphertext packing* (time in milliseconds) across different IoT devices. y-axis in log scale.

SSL and takes only 0.4 us on a 2xl device, 5 us on Pi3, 18 us on Pi0, and 18 us on A8. Even in the worst case, ElGamal implemented using the less optimized BigDigits library takes only 1.7 ms on the most resource-constrained device, M3. PRN pre-computation has similar benefits for Paillier with the worst case of implementing Paillier using BigDigits taking 41.6 ms on an M3 device. The ciphertext packing optimization allows us to pack 2 `int` values in a single ElGamal ciphertext which roughly cuts the encryption time in half (packing messages to a single plaintext itself takes minimal time). Packing is even more effective for Paillier since we can pack up to 33 `int` items in a single Paillier ciphertext. The results demonstrate that with our proposed optimizations, encryption latency of the cryptosystems used by `C3PO` is acceptable even in very resource-constrained devices.

6.7.3 Smart Meter Analytics

In this evaluation we study the throughput of `C3PO` by running a set of analytical queries to analyze electricity usage of homes. We use the Smart* dataset [120] as our input. This dataset represents electrical meter readings collected over a 24-

Table 6.1.: Description of continuous queries used for smart meter analytics.

#	**Short Description**	**Output**
Q1	Number of readings	Total number of readings for a given time window
Q2	Consumption	Sum of total resource consumption for a given time
Q3	Peak consumption	Sorted list of aggregate consumption per 10 s in a given window
Q4	Top consumers	List of distinct consumers, sorted by their monthly consumption
Q5	Consumption series	Time-series of aggregate consumption per 10 s in a given
Q6	Billing	Monthly bill for each consumer based on time of usage

hour period at the rate of 1 reading per minute from 443 unique homes, totaling 637,526 records. Each reading is a tuple of three fields: `<timestamp, meter-id, meter-reading>`. We define throughput as the number of tuples processed by the application graph in unit time. The runtime is configured to avoid dropping tuples by using a fixed sized queue. When the queue becomes full, the source vertices stop emitting tuples. When slots in the queue free up, source vertices start emitting tuples again. To measure throughput, we adapted the queries used in IoTBench [121] for streaming systems. Details of queries are given in Table 6.1.

We used a time window of 60 s and executed the queries for at least 600 s. We ran these queries on 4 *m3.large* nodes on Amazon EC2. For C3PO, one of the 4 nodes was specified as a trusted tier node. The bandwidth of the trusted node was throttled to 8 Mbit/s to simulate a wide area network link. The results of our evaluation are

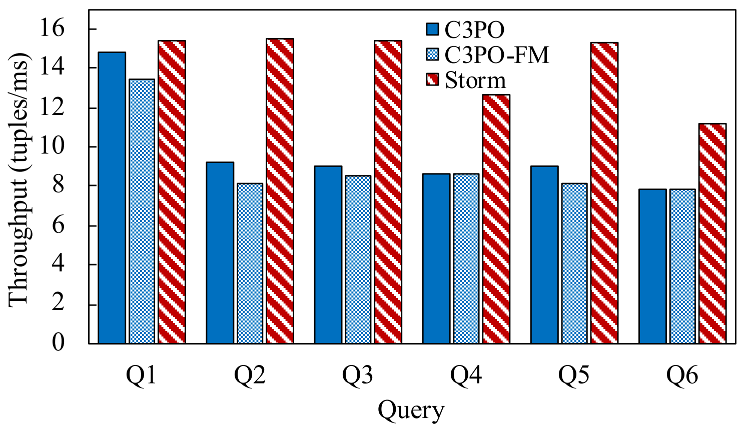


Fig. 6.11.: Smart meter analytics throughput

presented in Figure 6.11. Q1 simply counts the number of readings and performs at 96% throughput of plaintext stream. Queries Q2 to Q6, all perform Paillier additions and result in a throughput of 59% to 70% compared to Storm (plaintext) throughput. The results also show the effect of disabling field masking (C3PO-FM). For Q1, Q2, Q3 and Q5, we are able to mask one field, resulting in an average of 7% increase in throughput. Since queries Q4 and Q6 use all the fields in the stream, no fields could be masked.

6.7.4 Heartbeat Analysis

Next, we study how C3PO can be used for an online health care application like a heartbeat monitor. The end user application runs on specialized hardware (the monitoring IoT device) and counts the number of heartbeats per minute. The monitoring device uses *PRN pre-computation* to efficiently encrypt this value and send it to the cloud for processing and storing. PRN pre-computation allows us to use IoT devices with as little computational capabilities as the M3 node described above.

The graph running in the cloud keeps track of daily, weekly, monthly and yearly statistics. We use *post-encryption packing* to pack these 4 values into a single ciphertext, thereby reducing the ciphertext size by 4. The end user may request to see

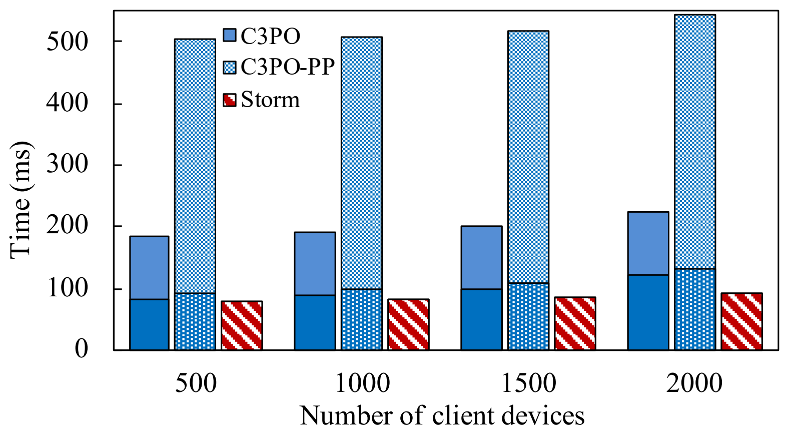


Fig. 6.12.: Heartbeat analysis response time

these statistics on their device, in which case the data is retrieved from the cloud, decrypted on the monitoring device and shown to the end user. The statistics are maintained by two vertices, a "per user" vertex (v_1) and an "all users" vertex (v_2). User statistics are distributed across the multiple instances of v_1. v_1 also emits a summary of its per user statistics every minute which is grouped by week, month, or year by v_2 to find the average value across all users. The client device emits a message every time the client requests to see a specific data point, in response to which, the requested values are retrieved. For this application, the most critical metric is the response time, i.e., the time a user has to wait after requesting to see a metric until the metric is displayed.

We deploy v_1 and v_2 on 3 *m3.medium* nodes in EC2 and use a single end user device deployed on an A8 device. We measure the response time as we increase the volume of incoming tuples to v_1 and v_2 by simulating additional end user devices using an Apache Kafka queue. Each of these end user devices, including the one deployed on the A8 node emits one tuple per minute containing an encrypted timestamp and the encrypted number of heartbeats.

The results of this evaluation are presented in Figure 6.12 where we compare the response time of `C3PO`, `C3PO`-PP which denotes `C3PO` with post-encryption packing

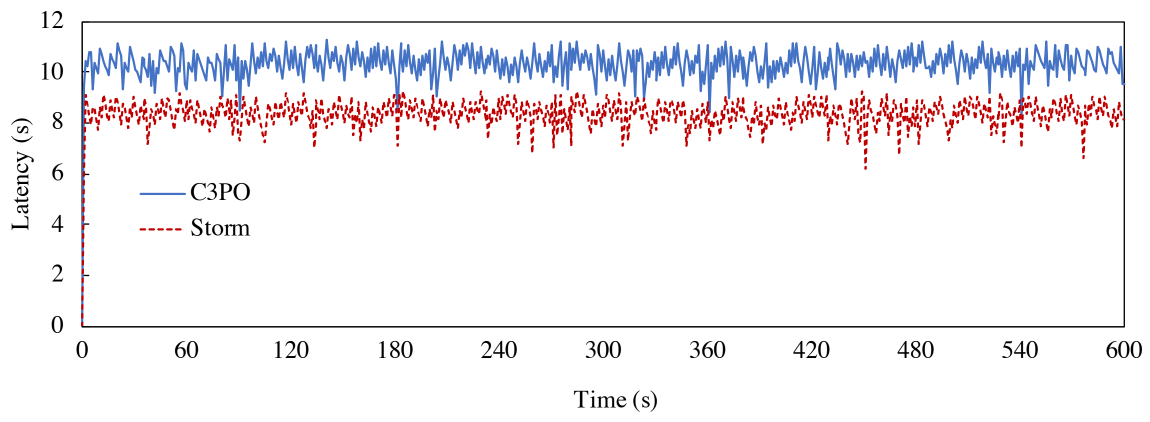


Fig. 6.13.: YSB latency

disabled, and Storm. The top part of each stacked column indicates decryption overhead. We observe that response times for `C3PO` when excluding decryption time are very close to the plaintext version for up to 1500 client devices after which `C3PO`'s response time degrades due to the increased load of v_1 and v_2 compared to the plaintext stream running on Storm. Decryption is a significant source of overhead. In `C3PO` without packing, the end user device receives and decrypts 4 4096-bit Paillier ciphertexts containing the daily, weekly, monthly and yearly statistics. Packing reduces that to a single ciphertext which leads to lower network overhead and significantly lower decryption time (˜4× lower).

6.7.5 Yahoo Streaming

We use the Yahoo Streaming Benchmark (YSB) to study the latency of `C3PO`. This benchmark simulates an advertising analytics use case with several advertising campaigns, each containing several advertisements. The benchmark reads various events from Apache Kafka, identifies the events relevant to the advertisement campaign, and stores a windowed count of relevant events per campaign. The steps in this analytical processing pipeline are as follows: a) Read an event (in `JSON` format) from a Kafka queue. b) Deserialize the `JSON` formatted event string into individual event fields. c) Filter out irrelevant events, based on the event type. d) Take a projection of the relevant fields keeping the ad id and the event time. e) Join each event with its associated campaign. This ad-to-campaign mapping information is stored in a Redis in-memory data store. f) Take a windowed count of events per campaign and store each window in Redis along with a timestamp of the time the window was last updated in Redis.

The input data for this evaluation is generated using a `clojure` program that generates uniformly random tuples of the following format:

```
<user_id:UUID, page_id:UUID, ad_id:UUID, ad_type:String,
event_type:String, event_time:Timestamp, ip_address:String>
```

These tuples are then sent to the Kafka queue. The results find the latency that a particular processing system produces at a given input load. The test computes the latency (in ms) from when the last event was emitted to Kafka for that particular campaign window and when it was fully processed. Figure 6.13 shows the results of running this benchmark on encrypted input data (`C3PO`) and plaintext data (Storm). The results show that on average `C3PO` operates with only 23% higher latency than running the same computation over plaintext data.

6.7.6 Multiple Groups

In this evaluation we look at the effectiveness of using multiple key groups as outlined in Section 6.5. We evaluate the throughput (tuples/s) of a `group by.. sum` operation with varying number of key groups. The devices emit a two field tuple `<group-name, value>` and the application finds the total sum of values for each group for every one minute time interval. A key change is initiated for one of the groups every two minutes. The devices in the group which is changing the keys will emit tuples encrypted in the old key and new key for a time span on one minute.

As outlined in Section 6.5, when a key change is initiated, a special key change tuple is emitted. The `C3PO` worker node receives the key change notification and

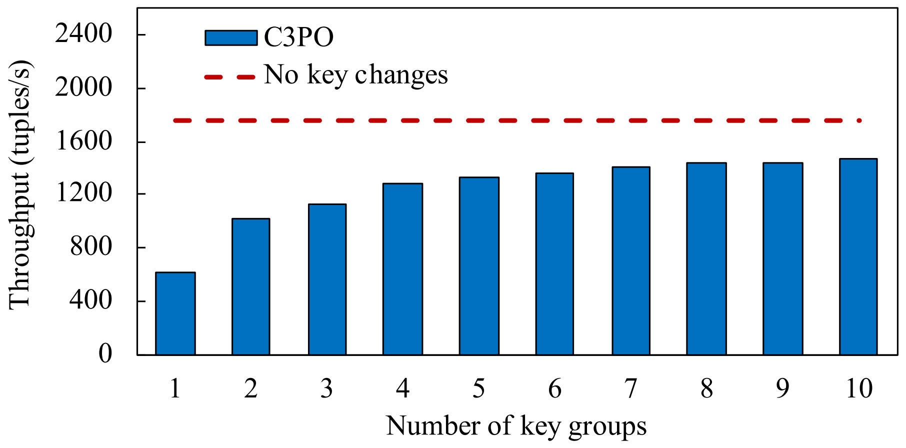


Fig. 6.14.: Effect of varying the number of key groups.

keeps track of partial results for the new key group until the next time window starts. Figure 6.14 shows the results. The system runs at a throughput of 1763 tuples per second when no keys are changed. If an encryption key is compromised and we are to update the key without using *multi-group* mode, we have to emit all input tuples under both the old and new key for the aggregation time window, reducing throughput to 622 tuples per second. Multi-group mode allows us to change the key of a specific group, reducing the impact of key changes.

In Figure 6.14, we can see that using multi-group mode, with the number of key groups increasing to 10, throughput increases from 622 to 1466. The throughput increases because the impact (number of tuples to be emitted under old and new key) of an updating a key is smaller. This shows that multi-group mode is an effective way to update encryption keys.

6.7.7 New York Taxi Statistics

The New York taxi statistics application finds the top 10 most frequent routes during the last 30 minutes of taxi servicing. A route is represented by a starting grid cell and an ending grid cell. The data for this application is based on a publicly

Table 6.2.: Completion and response times for Top-10 taxi routes. In `C3PO`^{a} the entire stream is emitted under same key. In `C3PO`^{b} the stream is emitted with a new key every month.

System	**Completion Time (s)**	**Average Response Time (ms)**
Storm	8106	36.05
`C3PO`^{a}	10039	45.10
`C3PO`^{b}	10140	46.61

available taxi data set released under FOIL (Freedom of Information Law). The input data contains the locations (latitude and longitude) of passenger pick ups and drop offs, MD5 digests of the medallions of the taxis that picked up the passengers, and the trip times. The dataset contains records that span over a year. Whenever the top 10 values change, the output is appended. In other words, for every top-10 tuple emitted, there is a specific input tuple that caused the top-10 values to be updated. Response time is defined as the time difference between these two actions. I.e., given a tuple t that causes the top-10 values to change from tuple $top10_{t-1}$ to $top10_t$, and a function $T(x)$ that gives us the time at which tuple x is emitted, response time is $T(top10_t) - T(t)$ In order to evaluate the effect of key changes on response time, we simulate a key change at the beginning of each month. This means that all data is emitted with a timestamp within the first 30 minutes of every month will be encrypted under both the old and new keys.

We deployed this application on 10 *m3.large* nodes in Amazon EC2. Table 6.2 summarizes the results of these runs. We can see that `C3PO` with no key changes completes processing the data with only a 23.8% and 25.1% increase of completion time and average response time respectively compared to the Storm running on a plaintext stream. Furthermore, the increase in completion times and average response times caused by a monthly key change are minimal (about 1%). Figure 6.15 shows the

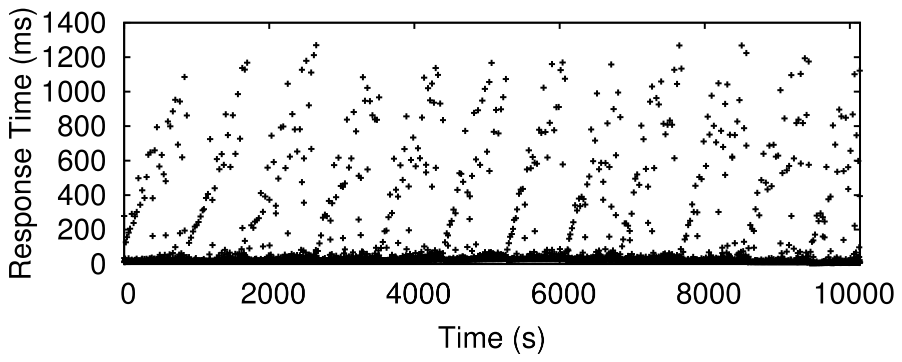


Fig. 6.15.: Response time of top-10 taxi route query with monthly key change

response times for the full 10,000 s run to process a year-long data with key changes in input data every month. In this plot, we can see intermittent spikes (total of 12) in response time for some tuples around the time a key change is in progress, but the majority of tuples (90th percentile within 31ms and 99th percentile within 818ms) respond with the same response time as when no change was in effect.

6.8 Related Work

In this section we overview PHE and trusted hardware based systems that provide confidentiality in stream processing and IoT settings.

6.8.1 PHE-based Systems

Gentry introduced an implementable FHE scheme [3] which has been becoming more practical ever since [33, 124], but is still not suited for encryption-enabled continuous query processing due to its prohibitive cost. Instead, a lot of research work is focused on using PHE schemes to perform computations over encrypted data. Section 4.8 has discussed several such systems.

JEDI [125] is another system, that introduces an end-to-end encryption scheme that leverages the hierarchical resource structure of IoT systems to delegate keys in a decentralized manner across multi-hops. WKD-IBE and AES schemes, and assembly level optimizations are incorporated to support embedded IoT devices.

Talos [114] is a system that encrypts and stores data specific to the IoT in a cloud database while still allowing query processing. Talos uses TLS to encrypt communication between end devices and cloud nodes which offers better security for data in-transit. Talos also utilizes optimizations proposed in previous work to make Order-Preserving Encryption (OPE) and Additive Homomorphic Encryption (AHE) schemes more suitable for IoT devices.

None of these systems consider streaming workloads as supported by `C3PO` and therefore do not address issues that arise in this setting, such as key management to limit the effect of key compromises or efficient cloud deployments for streaming applications.

The STYX [28, 88] system is an early version of `C3PO`. STYX promoted similar programming abstractions as `C3PO`, but did not allow for limited key sharing in time (updates) or space (multiple groups). `C3PO` makes PHE and PPE schemes applicable to resource-constrained IoT devices through the use of novel optimizations and presents additional empirical evaluation results, in particular with respect to encryption on IoT devices (and associated optimizations) and key management.

6.8.2 Trusted Hardware based Systems

Another way to enforce data confidentiality is through the use of specialized hardware that provide a trusted execution environment. An approach that is gaining popularity now is to use an Intel SGX enabled processor which offers a trusted execution environment (so-called *enclaves*) in which data computations can be carried out confidentially. SGX offers hardware encrypted and integrity protected physical memory, which allows data and code to reside in the untrusted cloud.

SecureCloud [126] shows how SGX can be used to enable secure and private execution of big data applications in the cloud.

Havet et al. [127] describe the design of *secure streams*, a streaming system that uses SGX to preserve confidentiality. Secure streams use a Lua VM running inside SGX enclaves to capture worker and router components. Workers handle the application logic and routers use a dispatching policy to handle message passing from one worker to another. Each component is then wrapped in Docker containers for isolation and ease of deployment. Secure streams demonstrate that a streaming system using SGX is feasible. Instead, `C3PO` focuses on PHE and does not require specialized hardware. In addition, `C3PO` addresses challenges such as managing keys and allows for different deployments that improve performance.

Trusted hardware approaches are orthogonal to the PHE-approach used by `C3PO` and could also be used to extend the design of `C3PO` by allowing secure computations to be performed on trusted hardware in an untrusted cloud. Cuttlefish shows the benefits of using SGX selectively, when hitting the limits of PHE (for re-encryption). Though promising, practical overall performance of the processor for big data workloads is yet to be ascertained and immediate widespread adoption of SGX by cloud service providers seems unlikely in the immediate present.

6.9 Conclusions

We presented `C3PO`, a practical distributed stream processing system for evaluating continuous queries over encrypted data streams in public clouds. `C3PO` makes computations over encrypted data practical for stream processing by using a novel API, encryption inference, automatic re-encryption, and a set of other original optimizations. The `C3PO` API allows programmers to develop secure applications with little or no knowledge of the underlying cryptosystems. We evaluated our approach using standard benchmarks and applications, demonstrating its applicability and performance. Our evaluations show that we can meet latency requirements even with high volumes of encrypted traffic.

7. CONCLUSION AND FUTURE WORK

In this chapter we summarize our work and describe future directions.

7.1 Summary

In the past decade, cloud computing has emerged as an economical and practical alternative to in-house datacenters. But due to security concerns, many enterprises are still averse to adopting third party clouds. To mitigate these concerns, several authors have proposed to use Partially Homomorphic Encryption (PHE), Property-Preserving Encryption (PPE), and Trusted Execution Environment (TEE) to achieve practical levels of confidentiality while enabling computations in the cloud. However, these approaches are either not performant or not versatile enough.

In this thesis, we presented extensions and optimizations to existing PHE and PPE schemes. The extensions we introduced make computations over encrypted data more expressive and more secure, while the optimizations we introduced make computations over encrypted data more performant.

We then presented three systems, `Cuttlefish`, `Symmetria`, and `C3PO` that demonstrate the practicality of batch and stream based data analytics in untrusted clouds by performing computations over encrypted data. Our `Cuttlefish` system was built on top of Apache Spark and introduced the abstraction of Secure Data Types that allows an application programmer to specify intrinsic properties about the structure and constraints of data. In turn this abstraction enables a set of compilation techniques, allowing `Cuttlefish` to generate more optimized queries. `Cuttlefish` also described the design of a planner engine that allows `Cuttlefish` to smartly utilize remote and trusted hardware based re-encryption to overcome limitations of PHE.

Our second system, `Symmetria`, is a PHE-based system which is also built on top of Apache Spark. In `Symmetria` we introduced two novel symmetric PHE schemes, namely, Symmetric Additive Homomorphic Encryption (SAHE) and Symmetric Multiplicative Homomorphic Encryption (SMHE). The symmetric nature of SAHE and SMHE make them more performant than state-of-the-art asymmetric PHE schemes. Crucially, SAHE and SMHE improve performance of arithmetic computations over encrypted data while supporting the full range of homomorphic operations that asymmetric PHE schemes support.

Lastly, we presented our system `C3PO` which demonstrated the practicality of stream-based data analytics over encrypted data. In `C3PO` we introduced techniques to make encryption on resource-constrained Internet of Things (IoT) devices under PHE and PPE schemes more efficient. We then showed that continuous query applications can be expressed through an intuitive API without requiring application programmers to have knowledge about specific PHE and PPE schemes.

7.2 Future Work

A hybrid model for secure data analytics This thesis has discussed a number of security mechanisms by which cloud security can be achieved, such as Fully Homomorphic Encryption (FHE), PHE, PPE, TEE and client side computation. A possible future direction is one that combines these and other security mechanisms in ways that preserve confidentiality and offer improved performance. To that end, a programming model can be defined that enables programmers to express data analytics but otherwise hides the complexities of the aforementioned mechanisms. Instead, utilizing ideas from multi-level security, the model can allow programmers to specify the security requirements of the input data in the form of a multi-dimensional security lattice. These requirements can be matched to security mechanisms that satisfy them. Whenever a program is submitted, it can be transparently transformed in such a way that it uses different mechanisms to execute different parts, each time utilizing an appropriate security mechanism that satisfies the security requirements of the involved data. In situations where multiple mechanisms can be used to securely execute a part of the given program, a cost-model can be used to decide which mechanism would lead to better performance.

A combined hardware and software approach to improve cloud security We have mentioned throughout this thesis that the security levels of PHE and PPE schemes varies. In particular, PHE schemes used in this work are malleable and don't preserve the integrity of computation. On the other hand, some PPE schemes are deterministic and have been shown to be vulnerable to frequency analysis and inference attacks. We plan to propose a combined model that uses a nested solution, to improve the security of computations in the cloud. More specifically, under this model, a TEE is used to ensure the integrity and authenticity of data and computation. On top of this, computation within a TEE enclave occurs over encrypted data using PHE and PPE to preserve data confidentiality. This removes the need to trust the TEE vendor to preserve the confidentiality of the data. An adversary that tries to exploit the malleability of PHE or attempts to find access patterns in the use of PPE will be prevented from doing so because of the security guarantees of the TEE. On the other hand, the PHE and PPE schemes provide data confidentiality independently of the trustworthiness of the TEE, since plaintext data is never revealed to the TEE.

REFERENCES

REFERENCES

[1] Cloud Security Alliance, “Top Threats to Cloud Computing (https://cloudsecurityalliance.org/topthreats/csathreats.v1.0.pdf),” 2014.

[2] InfoWorld, “Seven Cloud Computing Security Risks (https://www.infoworld.com/article/2652198/security/gartner--seven-cloud-computing-security-risks.html),” 2010.

[3] C. Gentry, “Fully homomorphic encryption using ideal lattices,” in *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, ser. STOC ’09. New York, NY, USA: ACM, 2009, pp. 169–178. [Online]. Available: http://doi.acm.org/10.1145/1536414.1536440

[4] P. Martins, L. Sousa, and A. Mariano, “A survey on fully homomorphic encryption: An engineering perspective,” *ACM Comput. Surv.*, vol. 50, no. 6, pp. 83:1–83:33, Dec. 2017. [Online]. Available: http://doi.acm.org/10.1145/3124441

[5] P. Paillier, “Public-key cryptosystems based on composite degree residuosity classes,” in *Int. Conf. on The Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 1999, pp. 223–238. [Online]. Available: http://dl.acm.org/citation.cfm?id=1756123.1756146

[6] L. Fousse, P. Lafourcade, and M. Alnuaimi, “Benaloh’s dense probabilistic encryption revisited,” in *AFRICACRYPT*, ser. Lecture Notes in Computer Science, vol. 6737. Springer, 2011, pp. 348–362.

[7] T. Okamoto, S. Uchiyama, and E. Fujisaki, “Epoc: Efficient probabilistic public-key encryption,” *IEEE P1363a*, p. 18, 1998.

[8] D. Naccache and J. Stern, “A new public key cryptosystem based on higher residues,” in *Proceedings of the 5th ACM conference on Computer and communications security*, 1998, pp. 59–66.

[9] I. Damgård and M. Jurik, “A generalisation, a simplification and some applications of paillier’s probabilistic public-key system,” in *International Workshop on Public Key Cryptography*. Springer, 2001, pp. 119–136.

[10] T. ElGamal, “A public-key cryptosystem and a signature scheme based on discrete logarithms,” *Trans. on Information Theory*, vol. 31, no. 4, pp. 469–472, 1985.

[11] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978. [Online]. Available: http://doi.acm.org/10.1145/359340.359342

[12] S. Goldwasser and S. Micali, "Probabilistic encryption," *Journal of computer and system sciences*, vol. 28, no. 2, pp. 270–299, 1984.

[13] T. Sander, A. Young, and M. Yung, "Non-interactive cryptocomputing for nc/-sup 1," in *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*. IEEE, 1999, pp. 554–566.

[14] D. Boneh, E.-J. Goh, and K. Nissim, "Evaluating 2-dnf formulas on ciphertexts," in *Int. Conf. on Theory of Cryptography (TCC)*, ser. TCC'05, 2005, pp. 325–341. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30576-7_18

[15] E. Barker, "Nist special publication 800–57, part 1, revision 5, recommendation for key management," May 2020.

[16] S. Halevi and P. Rogaway, "A tweakable enciphering mode," in *Annual Int. Cryptology Conf. (CRYPTO)*, 2003, pp. 482–499.

[17] A. Boldyreva, N. Chenette, and A. O'Neill, "Order-preserving encryption revisited: Improved security analysis and alternative solutions," in *Annual Int. Cryptology Conf. (CRYPTO)*. Springer-Verlag, 2011, pp. 578–595. [Online]. Available: http://dl.acm.org/citation.cfm?id=2033036.2033080

[18] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Symp. on Security and Privacy (S&P)*, 2000, pp. 44–55. [Online]. Available: http://dl.acm.org/citation.cfm?id=882494.884426

[19] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *ACM Conference on Computer and Communications Security*. ACM, 2015, pp. 668–679.

[20] M. Giraud, A. Anzala-Yamajako, O. Bernard, and P. Lafourcade, "Practical passive leakage-abuse attacks against symmetric searchable encryption," in *14th International Conference on Security and Cryptography SECRYPT 2017*. SCITEPRESS-Science and Technology Publications, 2017.

[21] P. Grubbs, K. Sekniqi, V. Bindschaedler, M. Naveed, and T. Ristenpart, "Leakage-abuse attacks against order-revealing encryption," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2017, pp. 655–672.

[22] M. Naveed, S. Kamara, and C. V. Wright, "Inference attacks on property-preserving encrypted databases," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 644–655. [Online]. Available: http://doi.acm.org.ezproxy.lib.purdue.edu/10.1145/2810103.2813651

[23] R. Poddar, T. Boelter, and R. A. Popa, "Arx: an encrypted database using semantically secure encryption," *Proceedings of the VLDB Endowment*, vol. 12, no. 11, pp. 1664–1678, 2019.

[24] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '13. New York, NY, USA: ACM, 2013, pp. 10:1–10:1. [Online]. Available: http://doi.acm.org.ezproxy.lib.purdue.edu/10.1145/2487726.2488368

[25] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "Cryptdb: Processing queries on an encrypted database," *Commun. ACM*, vol. 55, no. 9, pp. 103–111, Sep. 2012. [Online]. Available: http://doi.acm.org/10.1145/2330667.2330691

[26] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich, "Processing analytical queries over encrypted data," *Proc. VLDB Endow.*, vol. 6, no. 5, pp. 289–300, 2013. [Online]. Available: http://www.vldb.org/pvldb/vol6/p289-tu.pdf

[27] J. J. Stephen, S. Savvides, R. Seidel, and P. Eugster, "Practical confidentiality preserving big data analysis," in *W. on Hot Topics in Cloud Computing (HotCloud)*, 2014. [Online]. Available: https://www.usenix.org/conference/hotcloud14/workshop-program/presentation/stephen

[28] J. J. Stephen, S. Savvides, V. Sundaram, M. S. Ardekani, and P. Eugster, "Styx: Stream processing with trustworthy cloud-based execution," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, ser. SoCC '16. New York, NY, USA: ACM, 2016, pp. 348–360.

[29] 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*, ser. VLDB '07. VLDB Endowment, 2007, pp. 519–530. [Online]. Available: http://dl.acm.org/citation.cfm?id=1325851.1325912

[30] A. Armando, R. Carbone, L. Compagna, J. Cuellar, and L. Tobarra, "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)*, 2008, pp. 1–10.

[31] T. Ristenpart and E. Tromer, "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)*, 2009, pp. 199–212.

[32] G. J. Annas, "Hipaa regulations-a new era of medical-record privacy?" *New England Journal of Medicine*, vol. 348, no. 15, pp. 1486–1490, 2003.

[33] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the AES circuit." *IACR Cryptology ePrint Archive*, 2012, informal publication.

[34] H. Hacigümüs, B. R. Iyer, C. Li, and S. Mehrotra, "Executing SQL over encrypted data in the database-service-provider model," in *Int. Conf. on the Mgt. of Data (SIGMOD)*, 2002, pp. 216–227. [Online]. Available: http://doi.acm.org/10.1145/564691.564717

[35] S. D. Tetali, M. Lesani, R. Majumdar, and T. D. Millstein, "MrCrypt: Static analysis for secure cloud computations," in *Conf. on Object-Oriented Prog. Sys., Lang. and Applications (OOPSLA)*, 2013, pp. 271–286. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509554

[36] J. J. Stephen, S. Savvides, R. Seidel, and P. T. Eugster, "Program analysis for secure big data processing," in *Int. Conf. on Automated Software Engineering (ASE)*, 2014, pp. 277–288. [Online]. Available: http://doi.acm.org/10.1145/2642937.2643006

[37] S. Tople, S. Shinde, Z. Chen, and P. Saxena, "AUTOCRYPT: enabling homomorphic computation on servers to protect sensitive web content," in *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, 2013, pp. 1297–1310. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516666

[38] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, A. Modi, and S. Badrinarayanan, "Big data analytics over encrypted datasets with seabed," in *Symp. on Op. Sys. Design and Implementation (OSDI)*, 2016. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/papadimitriou

[39] S. Brenner, C. Wulf, M. Lorenz, N. Weichbrodt, D. Goltzsche, C. Fetzer, P. Pietzuch, and R. Kapitza, "Securekeeper: Confidential zookeeper using intel sgx," in *Int. Conf. on Middleware (MIDDLEWARE)*, 2016, pp. 14:1–14:13.

[40] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O\textquoterightKeeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "Scone: Secure linux containers with intel sgx," in *Symp. on Op. Sys. Design and Implementation (OSDI)*, 2016, pp. 689–703.

[41] "Intel® software guard extensions programming reference," 2014, https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[42] "Synopsys, inc., open source report 2014," 2014, http://go.coverity.com/rs/157-LQW-289/images/2014-Coverity-Scan-Report.pdf.

[43] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An oblivious and encrypted distributed analytics platform," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 283–298. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zheng

[44] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, 2012, pp. 15–28. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[45] S. Dara and S. R. Fluhrer, "FNR: arbitrary length small domain block cipher proposal," in *Intl. Conf. on Security, Privacy, and Applied Cryptography Engineering (SPACE)*, ser. Lecture Notes in Computer Science, vol. 8804. Pune, India: Springer, 2014, pp. 146–154.

[46] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill, "Order-preserving symmetric encryption," in *Int. Conf. on The Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2009, pp. 224–241. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-01001-9_13

[47] R. A. Popa, F. H. Li, and N. Zeldovich, "An ideal-security protocol for order-preserving encoding," in *Symp. on Security and Privacy (S&P)*, ser. SP '13. IEEE Computer Society, 2013, pp. 463–477. [Online]. Available: http://dx.doi.org/10.1109/SP.2013.38

[48] D. Boneh, K. Lewi, M. Raykova, A. Sahai, M. Zhandry, and J. Zimmerman, "Semantically secure order-revealing encryption: Multi-input functional encryption without obfuscation," in *Int. Conf. on The Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2015, pp. 563–594. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-46803-6_19

[49] K. Lewi and D. J. Wu, "Order-revealing encryption: New constructions, applications, and lower bounds," in *CCS '16 Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1167–1178. [Online]. Available: https://dl.acm.org/citation.cfm?id=2978376

[50] F. Kerschbaum, "Frequency-hiding order-preserving encryption," in *Int. Conf. on on Computer and Communications Security (CCS)*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 656–667. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813629

[51] N. Chenette, K. Lewi, S. A. Weis, and D. J. Wu, "Practical order-revealing encryption with limited leakage," in *Int. Conf. on Fast Software Encryption (FSE)*, ser. FSE 2016. New York, NY, USA: Springer-Verlag New York, Inc., 2016, pp. 474–493. [Online]. Available: https://doi.org/10.1007/978-3-662-52993-5_24

[52] P. Grubbs, K. Sekniqi, V. Bindschaedler, M. Naveed, and T. Ristenpart, "Leakage-abuse attacks against order-revealing encryption," in *Symp. on Security and Privacy (S&P)*, 2017, pp. 655–672. [Online]. Available: https://doi.org/10.1109/SP.2017.44

[53] F. Cajori, "Horner's method of approximation anticipated by ruffini," *Bull. Amer. Math. Soc.*, vol. 17, no. 8, pp. 409–414, 05 1911. [Online]. Available: http://projecteuclid.org/euclid.bams/1183421253

[54] A. Hosangadi, F. Fallah, and R. Kastner, "Optimizing polynomial expressions by algebraic factorization and common subexpression elimination," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 2012–2022, 2006.

[55] C. Lan, J. Sherry, R. A. Popa, S. Ratnasamy, and Z. Liu, "Embark: securely outsourcing middleboxes to the cloud," in *NSDI'16 Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, 2016, pp. 255–273. [Online]. Available: https://dl.acm.org/citation.cfm?id=2930629

[56] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner, "Rich queries on encrypted data: Beyond exact matches," in *Symposium on Research in Computer Security(ESORICS)*, 2015, p. 123–145. [Online]. Available: http://sprout.ics.uci.edu/pubs/rich-queries-ESORICS15.pdf

[57] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy, "BlindBox: deep packet inspection over encrypted traffic," in *SIGCOMM '15 Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015, pp. 213–226. [Online]. Available: https://dl.acm.org/citation.cfm?id=2787502

[58] A. Verma, L. Cherkasova, and R. H. Campbell, "Aria: Automatic resource inference and allocation for mapreduce environments," in *Int. Conf. on Autonomic Computing (ICAC)*, 2011, pp. 235–244. [Online]. Available: http://doi.acm.org/10.1145/1998582.1998637

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

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

[61] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B. Chun, "Making sense of performance in data analytics frameworks," in *Networked Sys. Design and Implem. (NSDI)*, 2015, pp. 293–307. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ousterhout

[62] "Apache Pig," http://pig.apache.org.

[63] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: A not-so-foreign language for data processing," in *Int. Conf. on the Mgt. of Data (SIGMOD)*, 2008, pp. 1099–1110. [Online]. Available: http://doi.acm.org/10.1145/1376616.1376726

[64] J. Li, M. Krohn, D. Mazières, and D. Shasha, "Secure untrusted data repository (sundr)," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 9–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251263

[65] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten, "SPORC: group collaboration using untrusted cloud resources," in *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, 2010, pp. 337–350. [Online]. Available: http://www.usenix.org/events/osdi10/tech/full_papers/Feldman.pdf

[66] P. Mahajan, S. T. V. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "Depot: Cloud storage with minimal trust," *ACM Trans. Comput. Syst.*, vol. 29, no. 4, pp. 12:1–12:38, 2011. [Online]. Available: http://doi.acm.org/10.1145/2063509.2063512

[67] Y. Dong, A. Milanova, and J. Dolby, "Jcrypt: Towards computation over encrypted data," in *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, 2016, pp. 1–12.

[68] M. Hauck, S. Savvides, P. Eugster, M. Mezini, and G. Salvaneschi, "Securescala: Scala embedding of secure computations," in *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*, 2016, pp. 75–84.

[69] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *The Second Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013, p. 10. [Online]. Available: http://doi.acm.org/10.1145/2487726.2488368

[70] A. Baumann, M. Peinado, and G. C. Hunt, “Shielding applications from an untrusted cloud with Haven,” in *Symp. on Op. Sys. Design and Implementation (OSDI)*, 2014, pp. 267–283. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann

[71] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan, “Orthogonal security with cipherbase,” in *6th Biennial Conference on Innovative Data Systems Research (CIDR’13)*, January 2013. [Online]. Available: https://www.microsoft.com/en-us/research/publication/orthogonal-security-with-cipherbase/

[72] S. Bajaj and R. Sion, “TrustedDB: A trusted hardware-based database with privacy and data confidentiality,” *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 3, pp. 752–765, 2014. [Online]. Available: http://dx.doi.org/10.1109/TKDE.2013.38

[73] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, “Vc3: Trustworthy data analytics in the cloud using sgx,” in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 38–54.

[74] B. Fisch, D. Vinayagamurthy, D. Boneh, and S. Gorbunov, “Iron: Functional encryption using intel sgx,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’17. New York, NY, USA: ACM, 2017, pp. 765–782. [Online]. Available: http://doi.acm.org.ezproxy.lib.purdue.edu/10.1145/3133956.3134106

[75] A. Gollamudi and S. Chong, “Automatic enforcement of expressive security policies using enclaves,” in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2016. New York, NY, USA: ACM, 2016, pp. 494–513. [Online]. Available: http://doi.acm.org/10.1145/2983990.2984002

[76] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani, “Moat: Verifying confidentiality of enclave programs,” in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’15. New York, NY, USA: ACM, 2015, pp. 1169–1184. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813608

[77] R. Sinha, M. Costa, A. Lal, N. P. Lopes, S. Rajamani, S. A. Seshia, and K. Vaswani, “A design and verification methodology for secure isolated regions,” in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’16. New York, NY, USA: ACM, 2016, pp. 665–681. [Online]. Available: http://doi.acm.org/10.1145/2908080.2908113

[78] “SGX Virtualization,” https://01.org/intel-software-guard-extensions/sgx-virtualization.

[79] D. E. Denning, “A lattice model of secure information flow,” *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, May 1976. [Online]. Available: http://doi.acm.org/10.1145/360051.360056

[80] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, Jul. 1977.

[81] D. M. Volpano, C. E. Irvine, and G. Smith, "A sound type system for secure flow analysis," *Journal of Computer Security*, vol. 4, no. 2/3, pp. 167–188, 1996. [Online]. Available: http://dx.doi.org/10.3233/JCS-1996-42-304

[82] T. Freeman and F. Pfenning, "Refinement types for ML," in *Conf. on Prog. Lang. Design and Implementation (PLDI)*. ACM, 1991, pp. 268–277. [Online]. Available: http://doi.acm.org/10.1145/113445.113468

[83] R. Davies, "Practical refinement-type checking," Ph.D. dissertation, Carnegie Mellon University, 2005, aAI3168521.

[84] S. Savvides, J. J. Stephen, M. S. Ardekani, V. Sundaram, and P. Eugster, "Secure data types: A simple abstraction for confidentiality-preserving data analytics," in *Symp. on Cloud Computing (SoCC)*, ser. SoCC '17. New York, NY, USA: ACM, 2017, pp. 479–492. [Online]. Available: http://doi.acm.org/10.1145/3127479.3129256

[85] D. Boneh, C. Gentry, S. Halevi, F. Wang, and D. J. Wu, "Private database queries using somewhat homomorphic encryption," in *International Conference on Applied Cryptography and Network Security*. Springer, 2013, pp. 102–118.

[86] BSI, "Cryptographic methods: Recommendations and key lengths, bsi technical guideline, bsi tr-02102-1," 2019.

[87] E. CSA, "Algorithms, key size and protocols report, d5.4," 2018.

[88] P. Eugster, S. Kumar, S. Savvides, and J. J. Stephen, "Ensuring confidentiality in the cloud of things," *IEEE Pervasive Computing*, vol. 18, no. 1, pp. 10–18, 2019.

[89] D. Ulybyshev, A. O. Alsalem, B. Bhargava, S. Savvides, G. Mani, and L. B. Othmane, "Secure data communication in autonomous v2x systems," in *2018 IEEE International Congress on Internet of Things (ICIOT)*. IEEE, 2018, pp. 156–163.

[90] D. Lemire and L. Boytsov, "Decoding billions of integers per second through vectorization," *Software: Practice and Experience*, vol. 45, no. 1, pp. 1–29, 2015.

[91] D. Lemire, N. Kurz, and C. Rupp, "Stream vbyte: Faster byte-oriented integer compression," *CoRR*, vol. abs/1709.08990, 2017.

[92] R. O. Oded Goldreich, "Software protection and simulation on oblivious rams," *Journal of the ACM*, vol. 43, no. 3, pp. 431–473, 1996.

[93] E. Stefanov and E. Shi, "Multi-cloud oblivious storage," in *CCS '13 Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, 2013, pp. 247–258.

[94] E. Stefanov, E. Shi, and D. Song, "Towards practical oblivious ram," *arXiv preprint arXiv:1106.3652*, 2011.

[95] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path oram: an extremely simple oblivious ram protocol," in *CCS '13 Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, 2013, pp. 299–310.

[96] Google, "Encrypted bigquery client," https://github.com/google/encrypted-bigquery-client.

[97] Intel, "Intel 64 and ia-32 architectures optimization reference manual," April 2019, https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-optimization-reference-manual.

[98] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark sql: Relational data processing in spark," in *SIGMOD '15 Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 1383–1394. [Online]. Available: https://dl.acm.org/citation.cfm?id=2742797

[99] H. Hacigümüs, B. R. Iyer, and S. Mehrotra, "Efficient execution of aggregation queries over encrypted relational databases," in *Database Systems for Advances Applications, 9th International Conference, DASFAA 2004, Jeju Island, Korea, March 17-19, 2004, Proceedings*, ser. Lecture Notes in Computer Science, Y. Lee, J. Li, K. Whang, and D. Lee, Eds., vol. 2973. Springer, 2004, pp. 125–136. [Online]. Available: https://doi.org/10.1007/978-3-540-24571-1_10

[100] E. Mykletun and G. Tsudik, "Aggregation queries in the database-as-a-service model," in *Data and Applications Security XX, 20th Annual IFIP WG 11.3 Working Conference on Data and Applications Security, Sophia Antipolis, France, July 31-August 2, 2006, Proceedings*, ser. Lecture Notes in Computer Science, E. Damiani and P. Liu, Eds., vol. 4127. Springer, 2006, pp. 89–103. [Online]. Available: https://doi.org/10.1007/11805588_7

[101] A. Liu, K. Zheng, L. Li, G. Liu, L. Zhao, and X. Zhou, "Efficient secure similarity computation on encrypted trajectory data," in *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, J. Gehrke, W. Lehner, K. Shim, S. K. Cha, and G. M. Lohman, Eds. IEEE Computer Society, 2015, pp. 66–77. [Online]. Available: https://doi.org/10.1109/ICDE.2015.7113273

[102] R. Kunkel, D. L. Quoc, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer, "Tensorscone: A secure tensorflow framework using intel sgx," *arXiv preprint arXiv:1902.04413*, 2019.

[103] N. Oliver and F. Flores-Mangas, "Healthgear: A real-time wearable system for monitoring and analyzing physiological signals," in *Int. W. on Wearable and Implantable Body Sensor Networks (BSN)*. Cambridge, Massachusetts, USA: IEEE Computer Society, 2006, pp. 61–64.

[104] H. Cho, D. Ippolito, and Y. W. Yu, "Contact tracing mobile apps for COVID-19: privacy considerations and related trade-offs," *CoRR*, vol. abs/2003.11511, p. , 2020.

[105] T. Gupta, R. P. Singh, A. Phanishayee, J. Jung, and R. Mahajan, "Bolt: Data management for connected homes," in *Networked Sys. Design and Implem. (NSDI)*, 2014, pp. 1–14.

[106] R. L. Rivest, L. Adleman, and M. L. Dertouzos, “On data banks and privacy homomorphisms,” *Foundations of Secure Computation, Academia Press*, pp. 169–179, 1978.

[107] O. Pandey and Y. Rouselakis, “Property preserving symmetric encryption,” in *Int. Conf. on The Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, vol. 7237. Cambridge, UK: Springer, 2012, pp. 375–391.

[108] S. Savvides, D. Khandelwal, and P. Eugster, “Efficient confidentiality-preserving data analytics over symmetrically encrypted datasets,” *Proc. VLDB Endow.*, vol. 13, no. 8, pp. 1290–1303, Apr. 2020.

[109] C. Bormann, M. Ersue, and A. Keranen, “Terminology for constrained-node networks,” Internet Requests for Comments, RFC Editor, RFC 7228, May 2014. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7228.txt

[110] Z. Yu and Y. Guan, “A key management scheme using deployment knowledge for wireless sensor networks,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 10, pp. 1411–1425, 2008.

[111] H. Dahshan and J. Irvine, “A robust self-organized public key management for mobile ad hoc networks,” *Security and Communication Networks*, vol. 3, no. 1, pp. 16–30, 2010.

[112] M. Mohammadi and A. Keshavarz-Haddad, “A new distributed group key management scheme for wireless sensor networks,” in *14th International Iranian Society of Cryptology Conference on Information Security and Cryptology, ISCISC*. Shiraz, Iran: IEEE, 2017, pp. 37–41.

[113] T. Dierks and E. Rescorla, “The transport layer security (tls) protocol version 1.2,” Internet Requests for Comments, RFC Editor, RFC 5246, August 2008. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5246.txt

[114] H. Shafagh, A. Hithnawi, A. Droescher, S. Duquennoy, and W. Hu, “Talos: Encrypted query processing for the internet of things,” in *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems, SenSys 2015, Seoul, South Korea, November 1-4, 2015*, 2015, pp. 197–210. [Online]. Available: http://doi.acm.org/10.1145/2809695.2809723

[115] N. Modadugu and E. Rescorla, “The design and implementation of datagram TLS,” in *Proc. of the Network and Distr. Sys. Sec. Sym. (NDSS)*. San Diego, California: The Internet Society, 2004.

[116] J. Daemen and V. Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer Verlag, 2002.

[117] “The GNU Multiple Precision Arithmetic Library,” https://www.gmplib.org, 2020.

[118] “OpenSSL Multiple Precision Arithmetic Library,” https://www.openssl.org, 2020.

[119] “BigDigits Multiple Precision Arithmetic Library,” http://www.di-mgt.com.au/bigdigits.html, 2020.

[120] S. Barker, A. Mishra, D. Irwin, E. Cecchet, P. Shenoy, and J. Albrecht, "Smart*: An open data set and tools for enabling research in sustainable homes," 2012.

[121] M. F. Arlitt, M. Marwah, G. Bellala, A. Shah, J. Healey, and B. Vandiver, "Iotabench: an internet of things analytics benchmark," in *ACM/SPEC Int. Conf. on Performance Eng.* Austin, TX, USA: ACM, 2015, pp. 133–144.

[122] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. Peng, and P. Poulosky, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *IEEE Int. Parallel and Distributed Processing Symposium Workshops (IPDPS).* Chicago, IL, USA: IEEE Computer Society, 2016, pp. 1789–1792.

[123] "NYC's Taxi Trip Data," http://chriswhong.com/open-data/foil_nyc_taxi/.

[124] C. Gentry and S. Halevi, "Implementing gentry's fully-homomorphic encryption scheme," in *Proceedings of the 30th Annual International Conference on Theory and Applications of Cryptographic Techniques: Advances in Cryptology*, ser. EUROCRYPT'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 129–148. [Online]. Available: http://dl.acm.org/citation.cfm?id=2008684.2008697

[125] S. Kumar, Y. Hu, M. P. Andersen, R. A. Popa, and D. E. Culler, "JEDI: many-to-many end-to-end encryption and key delegation for iot," in *USENIX Security.* Santa Clara, CA: USENIX Assoc., 2019, pp. 1519–1536.

[126] F. Kelbert, F. Gregor, R. Pires, S. Köpsell, M. Pasin, A. Havet, V. Schiavoni, P. Felber, C. Fetzer, and P. R. Pietzuch, "Securecloud: Secure big data processing in untrusted clouds," in *Design, Automation & Test in Europe Conference & Exhibition (DATE).* Lausanne, Switzerland: IEEE, 2017, pp. 282–285.

[127] A. Havet, R. Pires, P. Felber, M. Pasin, R. Rouvoy, and V. Schiavoni, "Securestreams: A reactive middleware framework for secure data stream processing," in *Int. Conf. on Distributed and Event-based Systems (DEBS).* Barcelona, Spain: ACM, 2017, pp. 124–133.

VITA

VITA

Savvas Savvides has received a Ph.D. in Computer Science at Purdue University in 2020 under the supervision of Prof. Patrick Eugster. He has previously received a B.Sc. in Computer Science from the University of Manchester, UK and an M.Sc. in Computer Science from New York University, USA. His research interests span the areas of information security, distributed systems, and cloud computing with emphasis on secure and efficient distributed computations.

During his time as a Ph.D. candidate, Savvas has also completed an internship at IBM T. J. Watson Research Center as a Research Summer Intern where he worked on estimating the execution-time of Spark applications through static program analysis and runtime monitoring. He has also completed an internship at Fortanix as a Security Research Engineer Intern where he worked on achieving secure consensus using Intel SGX. He received the Oasis Labs Fellowship in 2019 and completed an internship at Oasis Labs where he worked on enabling privacy-preserving off-chain compute services in the Oasis Labs blockchain network using Intel SGX.

Savvas's departmental service at Purdue University included serving as the Travel Grant Chair for the Computer Science Graduate Student Board (GSB), for the year 2017-2018. During this time he re-initiated the travel grant program and helped provide funding to many Ph.D. students wanting to attend domestic and international conferences. He also served as the 2017-2018 GSB Webmaster and developed the official GSB website and the official GSB survival guide.