On the Trade-Offs of Combining Multiple Secure Processing Primitives for Data Analytics

Hugo Carvalho(✉), Daniel Cruz, Rogério Pontes, João Paulo, and Rui Oliveira

INESC TEC and Universidade do Minho, Braga, Portugal
{hugo.a.carvalho,daniel.c.cruz,rogerio.a.pontes,
joao.t.paulo,rui.oliveira}@inesctec.pt

Abstract. Cloud Computing services for data analytics are increasingly being sought by companies to extract value from large quantities of information. However, processing data from individuals and companies in third-party infrastructures raises several privacy concerns. To this end, different secure analytics techniques and systems have recently emerged. These initial proposals leverage specific cryptographic primitives lacking generality and thus having their application restricted to particular application scenarios. In this work, we contribute to this thriving body of knowledge by combining two complementary approaches to process sensitive data.

We present SafeSpark, a secure data analytics framework that enables the combination of different cryptographic processing techniques with hardware-based protected environments for privacy-preserving data storage and processing. SafeSpark is modular and extensible therefore adapting to data analytics applications with different performance, security and functionality requirements.

We have implemented a SafeSpark's prototype based on Spark SQL and Intel SGX hardware. It has been evaluated with the TPC-DS Benchmark under three scenarios using different cryptographic primitives and secure hardware configurations. These scenarios provide a particular set of security guarantees and yield distinct performance impact, with overheads ranging from as low as 10% to an acceptable 300% when compared to an insecure vanilla deployment of Apache Spark.

Keywords: Data analytics · Privacy · Trusted hardware

1 Introduction

Data analytics plays a key role in generating high-quality information that enables companies to optimize the quality of their business while presenting several advantages such as making faster business decisions, predicting users

A. Remke and V. Schiavoni (Eds.): DAIS 2020, LNCS 12135, pp. 3–20, 2020.
https://doi.org/10.1007/978-3-030-50323-9_1

behaviours, elaborating better marketing plans, and improving relationships with customers. As the amount of data to be analysed grows, companies tend to resort to cloud services due to their high levels of reliability, flexibility, and efficiency, as well as competitive costs. However, using cloud services to store and process data increases the risk of unauthorized access to it, thus presenting serious issues to the users, given that some data may contain private or sensitive information, such as personal e-mails, and medical or financial records. The problem can arise internally, for instance when a system administrator manages confidential data inappropriately [1], or externally through, for instance, the exploitation of bugs in the cloud infrastructure [4,6,7]. Also, the existence of regulations such as the European General Data Protection Regulation (GDPR) [3] stresses the need for a new set of security measures for sensitive data being stored and processed at third-party services.

Current secure data analytics solutions aiming at overcoming the previous challenges can be broadly divided into two groups. Applications in the first one operate over encrypted data or protected data to be more generic. These are based on cryptographic techniques such as deterministic [8,30] or homomorphic [24] encryption that allow doing different types of computations (e.g., equality, order, and arithmetic operations) over encrypted data. The second group of solutions uses hardware-based protected environments or trusted hardware as it is commonly known, such as Intel SGX [15] or Arm TrustZone [9], to process data analysis with privacy and integrity guarantees. As expected, each approach has its advantages and limitations as we will elaborate on below in Sect. 2.

With SafeSpark we combine, in a modular and extensible manner, both approaches in a secure data analytics framework. To the best of our knowledge, it is the first tool to do so. The contribution of this paper is threefold. We present a modular and extensible framework capable of combining a growing set of cryptographic data processing techniques with trusted hardware processing devices. We have implemented a prototype that extends the Apache Spark framework with secure operations using standard encryption, deterministic encryption, order-preserving encryption techniques, and the Intel SGX technology while remaining full Spark SQL compliant. And we thoroughly evaluate the prototype with the TPC-DS Benchmark under three scenarios using different cryptographic primitives and secure hardware configurations.

The remainder of the paper is organized as follows. Section 2 presents relevant background and Sect. 3 reviews the state of the art for secure data analytics. Section 4 describes SafeSpark's architecture and Sect. 5 details its prototype implementation. Section 6 presents the experimental evaluation. Section 7 concludes the paper.

2 Background

This section describes the cryptographic techniques we use and their security guarantees as well as the Intel SGX technology.

2.1 Cryptographic Schemes

Current privacy-preserving solutions use different encryption techniques to ensure data privacy [22,29].

STD is a symmetric encryption scheme that provides *Indistinguishability under Chosen-Plaintext Attack* (IND-CPA) security which ensures that no information is disclosed from ciphertexts [20]. This scheme has a strong security definition but does not support any kind of computation over encrypted data. As such, SafeSpark's prototype uses STD to protect data that does not need to be processed at the untrusted premises.

The DET scheme ensures that multiple encryption operations over the same plaintext, and with the same encryption key, will result in the same ciphertext. Therefore, this scheme leaks encrypted values that correspond to the same value in plaintext, thus providing *Indistinguishability under Distinct Chosen-Plaintext Attacks* (IND-DCPA) [30] security. Also, the DET scheme allows performing equality comparisons over ciphertexts, for instance, it can be used to support SQL queries such as GROUP BY, COUNT, or DISTINCT.

The OPE scheme allows comparing the order of two ciphertexts, which is preserved from the original plaintexts [12]. With this scheme, range queries like MAX, MIN, COUNT, ORDER BY and GROUP BY can be applied directly over encrypted data. Since the OPE scheme preserves more properties from the original plaintext data it also has weaker security guarantees - *Indistinguishability under Ordered Chosen-Plaintext Attack* (IND-OCPA).

Other schemes, such as Paillier Encryption [24] or Secure Multi-Party Computation [18] can also be used for building secure data processing systems. However, their performance impact is high thus affecting the practicality of the resulting solution [34]. Nevertheless, SafeSpark has a modular and extensible design capable of supporting additional schemes such as these in the future.

2.2 Intel SGX

Intel SGX [15] is a trusted hardware solution contemplating protected execution environments - called Enclaves - whose security relies on the processors' instructions and a set of keys only accessible to the hardware. Enclaves have isolated memory addresses with the assurance that no malicious external environment, such as the operating system or hypervisor can compromise their security.

SGX splits an application into a trusted and an untrusted environment. When a user wants to compute data using SGX, she starts by creating an Enclave, which is placed in a trusted memory region. Then, when the user's application calls a trusted function (*i.e.*, a function that runs within SGX Enclaves), the execution of the application and the input data needed for that function, are transferred to the enclave. Therefore, by exchanging encrypted data with the enclave, and securely transmitting the corresponding encryption keys, applications can safely execute operations over the plaintext of sensitive data without leaking information to the server where the operation is deployed [15].

Enclaves also provide sealing capabilities that allow encrypting and authenticating the data inside an enclave so that it can be written to persistent memory without any other process having access to its contents. Also, SGX relies on software attestation, which ensures that critical code is running within a trusted enclave. One of the main advantages of SGX against its predecessors is its lower Trusted Computing Base (TCB). This factor defines the set of components, such as hardware, firmware, or software components that are considered critical to system security. With SGX, TCB only includes the code that users decide to run inside their enclave. Thus, SGX provides security guarantees for attacks from malicious software running on the same computer.

2.3 Threat Model

SafeSpark considers a trusted and untrusted site. The Spark client resides on the trusted site (e.g.: private infrastructure) and the Spark cluster is deployed on the untrusted one (e.g.: public cloud). We assume a semi-honest, adaptive adversary (internal attacker) with control over the untrusted site, with the exception of the trusted hardware. The adversary observers every query, its access patterns and can also replay queries. However, our model assumes that the adversary is honest-but-curious and thus does not have the capability of modifying queries nor their execution. The parameters and results of queries are encrypted with a secret key only available to the client and enclaves.

3 Related Work

Current secure data analytics platforms fall into two broad approaches. One, like the Monomi [33] system, resort to cryptographic schemes such as DET and OPE to query sensitive data on untrusted domains. The other, relies on hardware-based protected environments.

Monomi, in particular, splits the execution of complex queries between the database server and the client. The untrusted server executes part of the query, and when the remaining parts cannot be computed on the server or can be more efficiently computed on the client-side, the encrypted data is sent to the client, which decrypts it and performs the remaining parts of the query. Seabed [25] has a similar approach with an architecture based on splitting the query execution between the client and the server. This platform proposes two new cryptographic schemes, ASHE and SPLASHE which allow executing arithmetic and aggregation operations directly over the cryptograms.

Contrarily, VC3 [31] and Opaque [35] follow a trusted hardware approach. Namely, they use Intel SGX [16] to create secure enclaves where sensitive data can be queried in plaintext without revealing private information. VC3 uses SGX to perform secure MapReduce operations in the cloud, protecting code and sensitive data from malicious attackers. Opaque is based on Apache Spark and adds new operators that, in addition to ensuring the confidentiality and integrity of the data, ensure that analytical processing is protected against inference attacks.

These additional security guarantees lead however to a high impact on performance, with Opaque being up to 62 times slower than the original Spark.

Segarra et al. in [32] propose a secure processing system build on top of Spark Streaming that uses Intel SGX to compute stream analytics over public untrusted clouds. This solution offers security guarantees similar to those proposed in Opaque without requiring changes to applications code.

Unlike previous work, this paper aims at exploring the combination of both cryptographic and trusted hardware primitives for the Spark SQL engine. To the best of our knowledge, this approach is still unexplored in the literature and, as shown in the paper, provides novel trade-offs in terms of performance, security, and functionality that better suit a wider range of data analytics applications.

4 Architecture

SafeSpark's architecture is based on the Apache Spark platform [10], which currently does not support big data analytics with confidentiality guarantees. In this section, we describe a novel modular and extensible architecture that supports the simultaneous integration of cryptographic and trusted hardware primitives.

4.1 Apache Spark

Apache Spark is an open-source data analytics engine for large-scale distributed and parallel data processing. Spark uses in-memory processing, which makes it way faster than its predecessors, such as Apache Hadoop [19]. Our work is based on Spark SQL, which is an upper-level library for structured data processing. Spark SQL provides a programming abstraction, called DataFrames, which presents a data table with rows and named columns, similar to a database table, and on which one can perform traditional SQL queries [10].

Spark's architecture, depicted by the white boxes in Fig. 1, consists of three main components. The *Driver Program* is responsible for managing and scheduling the queries submitted to the Spark cluster, while the *Cluster Manager* allocates resources (*e.g.*, CPU, RAM) to each query, dividing it into smaller tasks to be processed by the *Spark Workers*. Spark proposes a distributed architecture that scales horizontally. Namely, by launching *Spark Workers* on new servers, the queries being processed by the Spark cluster can leverage the additional computational resources of such servers.

Spark considers a *Data Storage* phase where information is uploaded to a given data source (*e.g.*, Apache HDFS). Stored data is then loaded into tabular representation (in-memory DataFrames) that can be efficiently queried.

During the *Data Processing* phase, clients start by creating a *SparkContext* object, that connects the program being executed (*Driver Program*) to the Spark environment. Then, each client submits queries to the system through the Spark SQL component, which generates an execution plan that is sent to the *Cluster Manager*. The latter divides the execution plan into multiple tasks and assigns each task to a subset of *Spark Workers* with available resources (*e.g.*, CPU, RAM). When all the tasks are completed, the result is sent from the *Spark Workers* to the *Driver Program*, which returns the output to the clients.

4.2 SafeSpark

SafeSpark extends Spark's architecture [10] by integrating multiple secure processing primitives that can be combined to offer different performance, security and functionality trade-offs to data analytics applications. Figure 1 shows the proposed architecture which contemplates four new components: *SafeSpark Worker*, *Handler*, *CryptoBox* and *SafeMapper*.

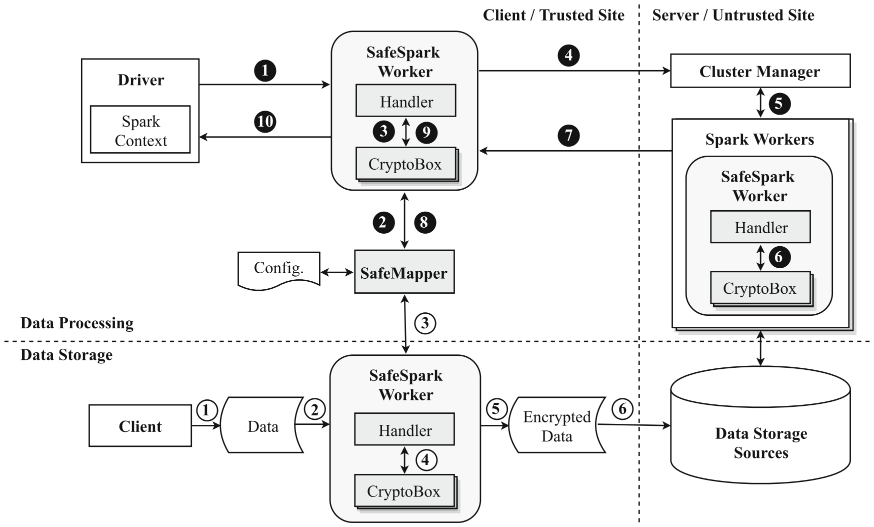

Fig. 1. SafeSpark's architecture

During the *Data Storage* phase, sensitive data is encrypted on the trusted site before being uploaded to the untrusted Spark data source. For this, the user must first specify in a configuration file how the data will be represented in a tabular form. Then, for each data column, the user will specify the type of cryptographic scheme (*e.g.* STD, DET, OPE) or trusted hardware technology (*e.g.* Intel SGX) to be employed.

The *SafeMapper* module is responsible for parsing the information contained in the configuration file and forwarding it to the *SafeSpark Worker*. The latter will intercept the plaintext data being uploaded to the untrusted data source and will encrypt each data column with the specified secure technique. The conversion of plaintext to encrypted data is actually done by the *Handler* component, which provides *encode()* and *decode()* methods for encrypting and decrypting information, respectively. Moreover, the *Handler* uses modular entities, called *CryptoBoxes*, each one corresponding to a different cryptographic technique or trusted hardware technology. Each *CryptoBox* contains an API with methods that allow generating a key, as well as methods to encrypt and decrypt data using the respective *CryptoBox* key.

The *SafeSpark Worker* is present on both sites and has the goal of abstracting the integration of cryptographic techniques and trusted hardware into the system. In addition to the *encode()* and *decode()* methods, it also implements a *process()* method that is used on the untrusted side to execute secure operations, during the *Data Processing* phase. This method is essential to enable the execution of secure operations, such as sums or averages, at the trusted hardware enclaves deployed on the untrusted premises.

The proposed architecture allows exploring different trade-offs between performance, privacy, and functionalities through the combination of different secure processing and storage primitives. Also, SafeSpark's modular design aims at easing the integration of new cryptographic algorithms and trusted hardware technologies, such as ORE [13], into the platform.

4.3 Flow of Operations

To exemplify the flow of operations in our platform let us consider the use-case of a company that wishes to store and query their employees' information in a third-party cloud service. The company's database will have an *Employees* table holding the *Salary*, *Age*, and *Category* of each employee (database columns). These columns contain sensitive information so the company's database administrators define a secure schema using SGX for the *Salary*, OPE for the *Age* and DET for the *Category*.

Firstly, the database's information must be uploaded to the corresponding cloud service (①). Given the sensitive nature of this data, the upload request is intercepted by the *SafeSpark Worker* (②) that initializes the SGX, OPE, and DET *CryptoBoxes* specified in the configuration schema (③), while using them to encrypt the corresponding data columns (④). The resulting encrypted data columns (⑤) are then uploaded into the untrusted data storage source (⑥).

Note that for encrypting data with the SGX technology, we consider a symmetric cipher similar to the STD scheme. During SafeSpark's bootstrap phase, the client application, running on the trusted premises must generate this key and exchange it with the enclave, through a secure channel, so that encrypted data can be decrypted inside the secure enclave and the desired operations can be done privately over plaintext data. This paper tackles the architectural challenges of integrating Intel SGX and other cryptographic primitives in Spark. Thus, we do not focus on the protocols of secure channel establishment or key exchange between clients and remote enclaves. Such challenges have been addressed in [11,27], which SafeSpark can rely upon in a real-world instantiation and that would not require any code changes at Spark's core codebase.

After completing the database's loading phase, clients can then query the corresponding information. Let us consider a SQL query that averages employees' salaries who are between 25 and 30 years and then groups the results by category.

```
SELECT Category, avg(Salary)
FROM Employees
WHERE Age BETWEEN 25 AND 30
GROUP BY Category
```

By sending the query through the *Spark Context* (❶), the request is intercepted by the *SafeSpark Worker*, which verifies the user-defined configuration file (❷), checking whether it is necessary to change the query, in order to invoke secure operators from *CryptoBoxes* (❸). Since stored values for the column *Age* are encrypted with OPE, the *SafeSpark Worker* encrypts the values "25"and "30" by resorting to the same OPE *CryptoBox* and key. Moreover, as the *Salary* column is encrypted using SGX, the operation *avg* needs to be performed within secure SGX enclaves. Therefore, SafeSpark provides a new operator that allows computing the average within SGX enclaves, while the *SafeSpark Worker* replaces the common operator *avg* by this new operator (*AVG_SGX*).

Then, after protecting sensitive query arguments at the trusted premises, the request is sent to the untrusted premises, namely to the *Cluster Manager*, which dispatches the tasks to *Spark Workers* (❺). Since the GROUP BY and BETWEEN operators internally perform equality and order comparison operations, and considering that *Category* and *Age* columns are encrypted with DET and OPE schemes, Spark is able to execute the operation directly over ciphertexts. However, the operation *avg* needs to be executed by the *SafeSpark Workers* using the *process()* method of the *CryptoBox* SGX (❻). At the SGX enclave, this method receives the input data to calculate *avg* and decrypts it with the previously agreed key. Then it does the *avg* calculation in plaintext and encrypts the result before sending it back to the untrusted Spark engine.

The query's encrypted result is sent to the Spark Client (❼) and intercepted by *SparkWorker* that, based on the *SafeMapper* component (❽), decrypts it using the appropriate *CryptoBox* (❾). Lastly, the plaintext result is sent back to the client (❿).

5 Implementation

SafeSpark's prototype leverages the *SafeMapper* and *CryptoBox* components used by SafeNoSQL [22]. Thereby, the STD and DET schemes were implemented with an AES 128-bit key in CBC mode with and without a random initialization vector, respectively, and by using the OpenSSL cryptographic library [5]. For the OPE scheme, we follow the implementation proposed by *Boldyreva et al.*, using the OpenSSL and MPFR (Multiple-Precision Floating-Point) libraries [5,17]. On the other hand, since the SafeNoSQL platform does not consider the use of SGX technology, we extended the library of *CryptoBox* components, in order to support arithmetic and relational operations using SGX. Next, we describe the implementation of the other SafeSpark components.

5.1 Data Storage

The conversion of plaintext data to encrypted one, during the data storage phase (Fig. 1), is done by using Parquet files [26] as these provide the standard format for Spark environments [14]. Parquet is a column-oriented data storage format that provides optimizations that make it very efficient, particularly for analytical processing scenarios. Our converter was implemented using the JAVA programming language, and it provides encode and decode (*i.e.*, encrypt and decrypt) methods that allow protecting sensitive data based on a secure database configuration schema. Thus, each column at the Parquet file is encrypted with the chosen encryption methods before being uploaded to the untrusted premises.

5.2 Data Processing

For the data processing phase the *SafeSpark Worker*, deployed at the untrusted site, is able to natively perform equality and order operations over columns protected with DET and OPE. However, when the SGX technology is being used, operations must be redesigned to execute within secure enclaves. For this reason, we resorted to Spark *Used-Defined Functions* (UDF) and *User-Defined Aggregate Functions* (UDAF's) since these allow us to change Spark's behaviour without directly changing its source code. The Scala programming language was used to implement these UDF/UDAFs. However, since SGX technology does not support the Scala programming language, we used the *Java Native Interface* (JNI) to call functions, developed in the C language, that are able to perform arithmetic and comparison operations using the SGX technology.

Considering this new set of functionalities, the SQL query presented at Sect. 4.3 is translated by the *SafeSpark Worker*, by invoking the corresponding SafeSpark operators, in the following way:

```
SELECT Category, AVG_SGX(Salary)
FROM Employees
WHERE Age BETWEEN 0FC6AC2E AND 0FC6D497
GROUP BY Category
```

Note that the *avg* operator is replaced by *AVG_SGX*, which is a new operator provided by SafeSpark that computes the salary average within secure SGX Enclaves. Moreover, the values "25" and "30" are replaced by "0FC6AC2E" and "0FC6D497", respectively, which is the hexadecimal representation for the output produced by the OPE encryption operation.

As a drawback of the current implementation, the Spark's framework does not yet provide a stable API for enabling a developer to define their own *User-Defined Types* (UDT). Therefore, if a specific data column was protected with SGX and that column is included in a GROUP BY or ORDER BY clause, its execution is not attainable since it is not possible to specify a UDF or UDAF for these two clauses. To solve this problem, we adopt a column duplication strategy. Thereby, when a data column is encrypted using SGX and one needs to perform GROUP BY or ORDER BY operations over it, that column is duplicated and protected with a DET or OPE primitive, respectively. However, this approach is not suitable for nested arithmetic and order operations, for instance, a SUM operation followed by an ORDER BY operation applied to the same column. Furthermore, as proposed by SafeNoSQL [22], this column duplication strategy is also used to improve the performance impact of decrypting data protected with the OPE scheme. Since this is a time-consuming operation, a duplicate column with the STD scheme is introduced so that, whenever a value encrypted with OPE needs to be retrieved in plaintext to the client (*e.g.*, the age of an employee) a faster decryption method is applied. The performance and storage space overhead trade-offs of these optimizations are further analysed in Sect. 6.

Finally, Spark SQL's DataFrames API was extended by creating a new operator, called *collectDecrypt*, that is responsible for decrypting the result of a query before presenting it to the user.

6 Experimental Evaluation

SafeSpark's prototype was evaluated to understand the impact of combining different privacy-preserving techniques. Namely, we compared Spark Vanilla against three different secure settings, on which we alternate the cryptographic and trusted hardware primitives being used and the data these are applied to.

6.1 Experimental Setup and Methodology

The experiments consider a distributed cluster composed of five nodes, configured with Cloudera Manager v.6.1.1. We used version 2.4 of Apache Spark and version 3.0.0 of HDFS for data storage. For the Client node, which is responsible for executing the queries and managing the cluster, we used a node equipped with an Intel Core i3-4170 CPU 3.70 GHz, 15.9GiB (DDR3) of RAM, a SATA3 119GiB SSD and with a Gigabit network interface. The nodes with data processing function (Workers) are equipped with an Intel Core i3-7100 CPU 3.9 GHz (with Intel SGX support), 7.8GiB (DDR3) of RAM, a SATA3 119GiB SSD and with a Gigabit network interface. During the data storage phase, we used a separate server to encrypt the data. This is equipped with an Intel (R) Xeon (R) CPU E5-2698 v4 @ 2.20 GHz, 31.3GiB (DDR3), and a Gigabit network interface.

We used the TPC-DS [23] benchmark, which models the decision support functions of a retail product supplier, considering essential components of any decision support system: data loading, multiple types of queries, and data maintenance. To explore different user behaviors for a decision support system, the TPC-DS benchmark provides four classes of SQL queries, each one representing a different database user activity in this type of system: Iterative, Data Mining, Reporting, and Ad-Hoc queries. For the experiments, we selected two queries from each group based on previous work [21,28]. Namely, we chose queries 24 and 31 from the Iterative OLAP class, queries 27 and 73 from the Reporting class, queries 37 and 82 from the Ad-Hoc class, and queries 40 and 46 from the

Data Mining class. TPC-DS was configured with a 10× scale factor, corresponding to a total of 12 GB of data to be loaded into Spark's storage source.

We performed ten runs of each TPC-DS query for Spark Vanilla, which computes over plaintext data, and for the different SafeSpark setups, which run on top of encrypted data. For each query, we analyzed the mean and standard deviation of the execution time. Also, the *dstat* framework [2] was used at each cluster node to measure the CPU and memory consumption, as well as the impact on disk read/write operations and on the network traffic. Moreover, we analyzed the data storage times and the impact of encrypted data on storage space.

6.2 SafeSpark Setups

The evaluation considers three SafeSpark setups with specific combinations of secure primitives for protecting TPC-DS's database schema, namely:

SafeSpark-SGX. This setup aims at maximizing the usage of SGX for doing queries over sensitive information at the TPC-DS database schema. Thus, the data columns which are used within arithmetic operations or filters of equality and order were encrypted using SGX. The OPE scheme was used for all the columns contemplating ORDER BY operations since this type of operation is not supported by the SGX operator, as explained in Sect. 5.2. For the remaining columns contemplating equality operations as GROUP BY or ROLL OUT, we used the DET scheme.

SafeSpark-OPE. This scenario aims at maximizing the use of cryptographic schemes, starting by using OPE and followed by the DET scheme. Therefore, in this case, SGX was only used for operations that are not supported by DET and OPE, namely arithmetic operations, sums or averages. Thus, OPE was used for all the operations containing order and equality comparisons, as ORDER BY, GROUP BY or BETWEEN clauses. For the remaining columns, that only require equality operations, the DET scheme was used.

SafeSpark-DET. As in the previous scenario, this one also maximizes the use of cryptographic schemes. However, it prioritizes the DET primitive instead of the OPE one, thus reducing the number of OPE columns that were being used in GROUP BY and ROLL UP operations in the previous scenario. Thus, SGX was only used for operations not supported by OPE or DET primitives. For columns that need to preserve equality, we used DET. For columns requiring order comparisons, we used the OPE scheme. In some cases, it was necessary to duplicate some columns already protected with the DET scheme. For example, when a column is targeted simultaneously by a GROUP BY (equality) and ORDER BY (order) operation.

Finally, we used the STD scheme to protect all columns on which no server-side data processing is performed. The secure setups used are further detailed at https://hugocarvalho9.github.io/safespark/testing-setups.html where it is shown the different secure primitives used for the TPC-DS schema.

6.3 Results

This section presents the results obtained from the experimental evaluation.

6.3.1 Loading and Storage

Table 1 shows that Spark Vanilla took 4.7 min for the storage phase. For the SafeSpark configurations, we considered not only the loading time but also the time used to encrypt the data. The SafeSpark-SGX setup took 697.1 min to encrypt and load the data, and the stored data size increased by 4.73×. The SafeSpark-OPE loading time was 735.1 min, and the data size increased by 6.23×. Lastly, the loading time for SafeSpark-DET was 776.3 min, and the data size increased by 6.39×.

Table 1. Loading time and data size.

Setup	Vanilla	SafeSpark-SGX	SafeSpark-OPE	SafeSpark-DET
Loading time	4.7 min	637.1 min	735.1 min	776.4 min
Data size	4.1 GB	19.4 GB	25.54 GB	26.2 GB

The impact shown throughout the storage phase can be explained by the use of the OPE scheme to encrypt data since it has a longer encoding time comparing with the other schemes, especially when the plaintext size is larger [22]. Also, the cryptograms produced by this scheme are significantly larger than the original plaintext, which can sustain the observed increase for the stored data size. In some situations, the cryptogram's size increases up to 4× when compared to the size of the original plaintext. It is important to note that all setups resort to the OPE primitive. However, SafeSpark-SGX is the setup that uses least this primitive and so has the fastest loading time. On the other hand, SafeSpark-DET has a higher loading time because it duplicates some columns to incorporate both DET and OPE primitives, as explained in Sect. 6.2.

6.3.2 Latency

Figure 2 presents the query latency results for the three SafeSpark configurations and Vanilla Spark. The values reflect each query execution time, as well as the time used to encrypt the query's parameters and to decrypt the query results when these are sent back to the client.

As expected, SafeSpark has worse performance than Spark Vanilla due to the secure primitives performance overhead. The SafeSpark-SGX scenario exhibits the highest overhead, while its best result occurs for query 24 with a 1.54× penalty and the worst for query 82 with a 4.1× penalty. These values can be justified by two factors. First, this scenario maximizes the use of SGX to protect data, leading to a wide number of data transfer and processing operations being executed within the SGX enclaves. We noted that, for example, query 31 has

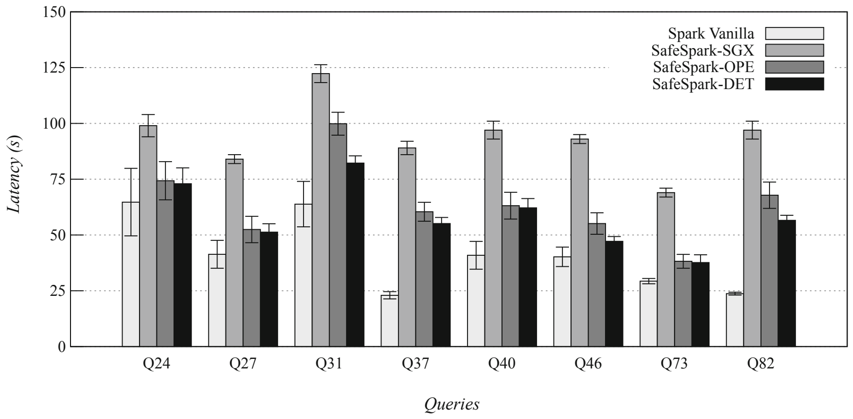

Fig. 2. Query execution times.

done approximately 4.5 million operations to SGX enclaves with an average time of 2.2 µs for each operation. Second, we use Spark SQL UDFs to perform operations on data protected with SGX. However, a limitation of Spark is that it currently does not support query plan optimizations for UDFs. Thus, the same query running on Spark Vanilla and SafeSpark may generate different execution plans, which can compromise the performance values obtained.

The SafeSpark-OPE maximizes the use of cryptographic schemes, thereby reducing the number of operations that are performed within SGX Enclaves. As we can observe in the Fig. 2, this testing scenario is more efficient than the previous one. This improvement is justified not only by the lower number of operations within Enclaves but also by reducing the use of UDFs, which leads Spark to generate optimized query execution plans. The best (1.15× penalty) and worst (2.86× penalty) execution times are still visible at queries 24 and 82, respectively. Although SafeSpark-OPE improves the results presented by SafeSpark-SGX, there are some queries where the execution time is significantly penalized by the time to encrypt the query parameters and decrypt the query results. For example, we noticed that query 31 took on average 14,226 s for decrypting the results, while 13,112 s were spent on OPE's decryption operation. In fact, the use of OPE to decrypt results shows a notable impact on the execution time, considering that the process of decrypting data using OPE is significantly slower than the analogous operations for the DET or STD ciphers, especially when the size of the cryptogram is larger.

SafeSpark-DET has its best execution time also for query 24, with a penalty of 1.13×. The worst result is for query 37, which is 2.4× slower than the same query executed on Spark Vanilla. It is also worth highlighting that there are six queries (24, 27, 31, 40, 46 and 73) where the execution time penalty is between 1.13× and 1.52×. Ad-Hoc queries (37 and 82) require a higher execution time due to the usage of UDFs for arithmetic operations done within SGX enclaves.

The results also show that SafeSpark-DET alleviates the penalty of decrypting data, by reducing the usage of the OPE scheme and maximizing the usage of the DET scheme. Consequently, as the number of values decrypted with the OPE scheme decreases, so it does the query execution time.

6.3.3 Resource Usage

Overall, resource usage results were similar for all SafeSpark setups. Due to space constraints, Table 2 highlights the worst-case results obtained for each resource (*i.e.*, CPU, memory, disk and network I/O). The full results can be consulted at https://hugocarvalho9.github.io/safespark/resource-usage.html.

Table 2. Resource consumption results

Resource	Query	Setup	Master	Worker #1	Worker #2	Worker #3	Worker #4
CPU (%)	40	Spark Vanilla	16	15.2	11	15.4	15.9
		SafeSpark-DET	10.5	19.1	24.9	17.7	24.2
Memory (GB)	37	Spark Vanilla	14.8	6.6	5.6	5.8	5.9
		SafeSpark-SGX	15.8	6.5	6.7	6.4	7
Disk read (KB/s)	46	Spark Vanilla	0.7	1.4	193.9	686.2	594.9
		SafeSpark-SGX	0.9	516.3	909.5	656.2	975.9
Disk write (KB/s)	82	Spark Vanilla	84.1	2726.6	2783.4	2791	2149.2
		SafeSpark-DET	83.7	11354.4	165.3	160.6	8961.2
Network recv (MB/s)	46	Spark Vanilla	304	1570.5	1939.4	3013	3083.1
		SafeSpark-DET	301.5	4309	4501.5	4467.6	4853.2
Network send (MB/s)	46	Spark Vanilla	7.4	0.3	0.5	0.6	1
		SafeSpark-DET	15.8	0.5	0.6	0.9	0.6

The CPU and memory consumption does not show notable changes, even considering the process of decrypting the query results and the computational power used by Intel SGX. The worst CPU consumption result occurred on query 40 with SafeSpark-DET, presenting an overhead 31%, when compared to Vanilla Spark. Regarding memory consumption, the worst overhead was 10% for SafeSpark-SGX, also on query 37.

SafeSpark has an impact on disk and network I/O. Query 46 with SafeSpark-SGX shows an overhead of 107% on disk reads, and query 82 with SafeSpark-DET has a 97% overhead on disk writes, when compared with Spark Vanilla. Finally, network traffic has the highest impact on query 46 with SafeSpark-DET (approximately 87%). These overheads are justified by the fact that cryptograms generated by SafeSpark, which will be sent through network channels and stored persistently, are larger than plaintext data. This is even more relevant when using the OPE scheme as it generates larger cryptograms.

6.4 Discussion

Based on the experimental results presented, we distilled a set of considerations that are described next.

Applications that collect vast amounts of real-time data and focus on decreasing the loading time and transferred/stored data size should avoid the usage of OPE. As we have seen, this scheme generates larger cryptograms and its encryption/decryption time introduces a significant impact on the loading time. Thereby, reducing the usage of OPE leads to better results in the storage phase, as well as on network and disk I/O traffic.

Concerning the queries execution time, we observed that performance can be influenced by two main factors: i) The number of columns that need to be decrypted with the OPE scheme when the result is sent back to the client; ii) The number of operations performed within SGX enclaves.

The first could be improved by leveraging SafeSpark's modular design to integrate more efficient secure order-preserving primitives such as ORE [13].

Regarding the second challenge, a significant source of overhead comes from our current implementation relying on Spark SQL's UDF/UDAF mechanisms for supporting SGX operations. These are not integrated with Spark's query planner component and thus, do not provide optimized query execution plans. A potential approach to face this problem could be to develop our own Spark operators and optimized execution plans, as done in Opaque [35]. Also, as future work, we could devise batching strategies to enable multiple operations to be executed in a single enclave call, which would reduce the number of calls to the enclave and their inherent performance overhead.

Finally, the SafeSpark-DET setup, which only uses OPE for ORDER BY operations and SGX for operations not supported by deterministic schemes, is able to achieve the best performance results. In fact, this setup supports six queries with overheads between 13% and 52%, when compared to Spark Vanilla. Nevertheless, it is important to have in mind that, with this performance increase, one is reducing the provided security guarantees. For instance, SafeSpark-DET presents lower security guarantees than SafeSpark-SGX.

Comparing our platform with the existing state-of-the-art systems, SafeSpark differs from the hardware-based approaches [31,32,35] since it enables the use of deterministic schemes to compute equality and order operations. This functionality makes it possible to achieve better performance results while relaxing the security guarantees. On the other hand, SafeSpark distinguishes itself from Monomi and Seabed platforms by using the SGX technology to perform arithmetic operations instead of using Homomorphic Encryption schemes.

7 Conclusion

This paper presents SafeSpark, a modular and extensible secure data analytics platform that combines multiple secure processing primitives to better handle the performance, security, and functionality requirements of distinct data analytics applications. Distinctively, SafeSpark supports both cryptographic schemes and the Intel SGX technology according to users' demand.

SafeSpark's experimental evaluation shows that it is possible to develop a practical solution for protecting sensitive information being stored and processed at third-party untrusted infrastructures with an acceptable impact on application performance. Moreover, while supporting the entire Spark SQL API. When comparing SafeSpark's performance with Spark Vanilla, the prototype's overhead ranges from roughly 10% to 300%. Particularly, with the SafeSpark - DET configuration, we show that for a majority of queries it is possible to maintain the performance overhead below 50%.

Currently, we are working to extend SafeSpark with other secure processing primitives with different security and performance trade-offs (e.g., ORE [13]). Evaluation with even larger data sets and new types of queries is underway too.

Acknowledgements. We thank the anonymous reviewers and our shepherd Pierre-Louis Aublin for their helpful suggestions. The research leading to these results has received funding from the European Union's Horizon 2020 - The EU Framework Programme for Research and Innovation 2014–2020, under grant agreement No. 857237 and FCT - Fundação para a Ciência e a Tecnologia grant SFRH/BD/142704/2018.

References

1. The cambridge analytical files. https://www.theguardian.com/news/series/cambridge-analytica-files. Accessed 2019
2. Dstat: versatile resource statistics tool. http://dag.wiee.rs/home-made/dstat/. Accessed 2020
3. Eu general data protection regulation. https://eugdpr.org/. Accessed 2020
4. Isaac, M., Frenkel, S.: Facebook security breach exposes accounts of 50 million users. https://www.nytimes.com/2018/09/28/technology/facebook-hack-data-breach.html. Accessed 2020
5. Openssl - cryptography and SSL/TLS toolkit. https://www.openssl.org/. Accessed 2020
6. Perlroth, N.: All 3 billion yahoo accounts were affected by 2013 attack. https://www.nytimes.com/2017/10/03/technology/yahoo-hack-3-billion-users.html. Accessed 2020
7. Roman, J.: Ebay breach: 145 million users notified. https://www.bankinfosecurity.com/ebay-a-6858. Accessed 2020
8. Agrawal, R., Kiernan, J., Srikant, R., Xu, Y.: Order preserving encryption for numeric data. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, pp. 563–574. ACM (2004)
9. ARM, A.: Security technology building a secure system using trustzone technology (white paper). ARM Limited (2009)
10. Armbrust, M., et al.: Spark SQL: relational data processing in spark. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 1383–1394. ACM (2015)
11. Bahmani, R., et al.: Secure multiparty computation from SGX. In: Kiayias, A. (ed.) FC 2017. LNCS, vol. 10322, pp. 477–497. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70972-7_27

12. Boldyreva, A., Chenette, N., Lee, Y., O'Neill, A.: Order-preserving symmetric encryption. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 224–241. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01001-9_13
13. Boneh, D., Lewi, K., Raykova, M., Sahai, A., Zhandry, M., Zimmerman, J.: Semantically secure order-revealing encryption: multi-input functional encryption without obfuscation. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9057, pp. 563–594. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46803-6_19
14. Chambers, B., Zaharia, M.: Spark: The Definitive Guide: Big Data Processing Made Simple. O'Reilly Media Inc, Sebastopol (2018)
15. Costan, V., Devadas, S.: Intel SGX explained. IACR Cryptology ePrint Archive 2016(086), 1–118 (2016)
16. Durak, F.B., DuBuisson, T.M., Cash, D.: What else is revealed by order-revealing encryption? In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 1155–1166. ACM (2016)
17. Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., Zimmermann, P.: MPFR: a multiple-precision binary floating-point library with correct rounding. ACM Trans. Math. Softw. (TOMS) **33**(2), 13 (2007)
18. Goldreich, O.: Secure multi-party computation. Manuscript. Preliminary version **78** (1998)
19. Hadoop, A.: Apache hadoop. http://hadoop.apache.org. Accessed 2020
20. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. CRC Press, Boca Raton (2014)
21. Kocberber, O., Grot, B., Picorel, J., Falsafi, B., Lim, K., Ranganathan, P.: Meet the walkers: accelerating index traversals for in-memory databases. In: Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 468–479. ACM (2013)
22. Macedo, R., et al.: A practical framework for privacy-preserving NoSQL databases. In: 2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS), pp. 11–20. IEEE (2017)
23. Nambiar, R.O., Poess, M.: The making of TPC-DS. In: Proceedings of the 32nd International Conference on Very Large Data Bases, pp. 1049–1058. VLDB Endowment (2006)
24. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, pp. 223–238. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48910-X_16
25. Papadimitriou, A., et al.: Big data analytics over encrypted datasets with seabed. In: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), pp. 587–602 (2016)
26. Parquet, A.: Apache parquet. Accessed 2020
27. Pass, R., Shi, E., Tramèr, F.: Formal abstractions for attested execution secure processors. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017. LNCS, vol. 10210, pp. 260–289. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56620-7_10
28. Poess, M., Nambiar, R.O., Walrath, D.: Why you should run TPC-DS: a workload analysis. In: Proceedings of the 33rd International Conference on Very Large Data Bases, pp. 1138–1149. VLDB Endowment (2007)
29. Popa, R.A., Redfield, C., Zeldovich, N., Balakrishnan, H.: CryptDB: protecting confidentiality with encrypted query processing. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, pp. 85–100. ACM (2011)

30. Rogaway, P., Shrimpton, T.: Deterministic authenticated-encryption. Citeseer (2007)
31. Schuster, F., et al.: VC3: Trustworthy data analytics in the cloud using SGX. In: 2015 IEEE Symposium on Security and Privacy (SP), pp. 38–54. IEEE (2015)
32. Segarra, C., Delgado-Gonzalo, R., Lemay, M., Aublin, P.-L., Pietzuch, P., Schiavoni, V.: Using trusted execution environments for secure stream processing of medical data. In: Pereira, J., Ricci, L. (eds.) DAIS 2019. LNCS, vol. 11534, pp. 91–107. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-22496-7_6
33. Tu, S., Kaashoek, M.F., Madden, S., Zeldovich, N.: Processing analytical queries over encrypted data. In: Proceedings of the VLDB Endowment, vol. 6, pp. 289–300. VLDB Endowment (2013)
34. Van Dijk, M., Juels, A.: On the impossibility of cryptography alone for privacy-preserving cloud computing. HotSec **10**, 1–8 (2010)
35. Zheng, W., Dave, A., Beekman, J.G., Popa, R.A., Gonzalez, J.E., Stoica, I.: Opaque: an oblivious and encrypted distributed analytics platform. In: NSDI, pp. 283–298 (2017)