SOTERIA: Preserving Privacy in Distributed Machine Learning

Cláudia Brito
INESC TEC & University of Minho
Braga, Portugal
claudia.v.brito@inesctec.pt

Pedro Ferreira
INESC TEC & Faculty of Sciences, University of Porto
Porto, Portugal
pferreira@ipatimup.pt

Bernardo Portela
INESC TEC & Faculty of Sciences, University of Porto
Porto, Portugal
bernardo.portela@fc.up.pt

Rui Oliveira
INESC TEC & University of Minho
Braga, Portugal
rcmo@inesctec.pt

João Paulo
INESC TEC & University of Minho
Braga, Portugal
joao.t.paulo@inesctec.pt

ABSTRACT

We propose SOTERIA, a system for distributed privacy-preserving Machine Learning (ML) that leverages Trusted Execution Environments (*e.g.* Intel SGX) to run code in isolated containers (enclaves). Unlike previous work, where all ML-related computation is performed at trusted enclaves, we introduce a hybrid scheme, combining computation done inside and outside these enclaves. The conducted experimental evaluation validates that our approach reduces the runtime of ML algorithms by up to 41%, when compared to previous related work. Our protocol is accompanied by a security proof, as well as a discussion regarding resilience against a wide spectrum of ML attacks.

CCS CONCEPTS

• **Security and privacy** → **Distributed systems security**;

KEYWORDS

Apache Spark, Machine Learning, Intel SGX, Privacy-Preserving

ACM Reference Format:
Cláudia Brito, Pedro Ferreira, Bernardo Portela, Rui Oliveira, and João Paulo. 2023. SOTERIA: Preserving Privacy in Distributed Machine Learning. In *Proceedings of ACM SAC Conference (SAC'23).* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3555776.3578591

1 INTRODUCTION

Outsourcing Machine Learning (ML) data storage and computation to third-party services (*e.g.*, cloud computing) leaves users vulnerable to attacks that may compromise the integrity and confidentiality of their data. Indeed, the ML pipeline encompasses several stages, both for model training and inference, in which users' data is known to be susceptible to different attacks such as *adversarial attacks*, *model extraction*, and *inversion*, and *reconstruction attacks* [13, 28].

SAC'23, March 27 – March 31, 2023, Tallinn, Estonia

ACM ISBN 978-1-4503-9517-5/23/03…$15.00
https://doi.org/10.1145/3555776.3578591

Recent works have addressed these attacks with solutions based on homomorphic encryption or secure multi-party computation schemes. However, these cryptographic schemes impose a significant performance toll that restricts their applicability to practical scenarios [3]. To circumvent this performance penalty, another line of research is that of exploring hardware technologies enabling Trusted Execution Environments (TEEs), such as Intel SGX [21]. These technologies allow the execution of code within isolated processing environments (*i.e.,* enclaves) where data can be securely handled in its original form (*i.e.*, plaintext) at untrusted servers.

The latter approach typically deploys full ML workloads inside TEEs [15, 16]. However, as the amount of computational and I/O operations performed at the enclaves increases, the performance of ML training and inference is noticeably affected by hardware limitations, limiting the design's applicability in practice [11].

This paper builds upon the idea that ML runtime performance could be improved by reducing the number of operations done at enclaves. In fact, this insight is backed up by previous work [19, 32] exploring the partitioning of computation across trusted and untrusted environments, but in contexts (*e.g.*, SQL processing, MapReduce, distributed coordination) with different security requirements and processing logic than the ones found for ML workloads.

Therefore, the key challenge addressed by this paper is to understand and define the set of ML operations to run inside/outside TEEs. Ideally, these operations should significantly reduce the enclaves' overall computational and I/O load for different ML workloads; and doing so should not leak critical sensitive information during the execution of ML workloads.

Our reasoning is twofold: *i.)* the majority of current attacks on the ML pipeline is only successful if the attacker has some knowledge about the datasets and/or models being used [6, 13]; and *ii.)* studies show that such knowledge cannot be inferred from the information leaked by statistical operations, such as the calculation of confidence results, table summaries, ROC/AUC curves, and probability distributions for classes [8]. As a result, these operations are ideal candidates to be offloaded from enclaves. We support these claims by analyzing the security and performance implications of different ML workloads and attacks.

Thus, we propose SOTERIA, an open-source system for distributed privacy-preserving ML (https://github.com/claudiavmbrito/soteria) that leverages the scalability and reliability provided by Apache Spark and its ML library (MLlib). Unlike previous solutions [14, 25],

Soteria supports a wide variety of ML algorithms without changing how users build and run these within Spark. It ensures that critical operations, which enable existing attacks to reveal sensitive information from ML datasets and models, are exclusively performed in secure enclaves. This means that the sensitive information being processed only exists in plaintext when inside the enclave, being encrypted in the remainder data flow (*e.g.*, network, storage). This solution enables robust security guarantees, ensuring data privacy during ML training and inference.

Soteria introduces a new computation partitioning scheme for Apache Spark's MLlib, Soteria-p, that offloads non-critical statistical operations from the trusted enclaves to untrusted environments. Soteria-p is accompanied by a formal security proof for how data remains private during ML workloads and an analysis of how this guarantee ensures resilience against various ML attacks. Furthermore, Soteria offers a baseline scheme, Soteria-b, where all ML operations are done inside trusted enclaves without a fine-grained differentiation between critical and non-critical operations. Soteria-b provides a performance and security baseline for comparison against our new partitioned scheme.

We compare experimentally both approaches with a non-secure deployment of Apache Spark and a state-of-the-art solution, namely SGX-Spark [14]. Our experiments, resorting to the HiBench benchmark [17] and including four different ML algorithms, show that Soteria-p, while considering a larger subset of ML attacks, reduces training time by up to 41% for Gradient Boosted Trees workloads and up to 4.3 hours for Linear Regression workloads, when compared to SGX-Spark. Also, when compared to Soteria-b, Soteria-p reduces execution time by up to 37% for the Gradient Boosted Trees workloads and up to 3.3 hours for the Linear Regression workloads.

2 BACKGROUND

2.1 Apache Spark and MLlib

Apache Spark is a distributed cluster computing framework that supports ETL, analytical, ML, and graph processing over large volumes of data. Spark follows a Master/Workers distributed architecture and can be deployed on a cluster of servers in the cloud that may access several data sources (*e.g.*, HBase, HDFS) for reading the data to be processed and storing the corresponding output and logs [31]. Spark is able to perform most of the computation in-memory, thus promoting better performance for data-intensive applications when compared to Hadoop's MapReduce.

The MLlib library [22] enables Spark users to build end-to-end ML workflows. These workflows are divided into 5 stages (Figure 1). The first stage goes from the collection of data to its treatment. In the second stage, data is split into train and test datasets, and a given ML algorithm is chosen. The third stage is the training stage, where data is iterated to deliver an optimized trained model at the fourth stage. In the fifth stage, the trained model can then be saved (persisted) and loaded (accessed) for inference purposes.

2.2 Intel Software Guard Extensions

Intel SGX provides a set of new instructions, available on Intel processors, that applications can use to create trusted memory regions. These regions (enclaves) are isolated from any other code on the host system, preventing other processes, including those

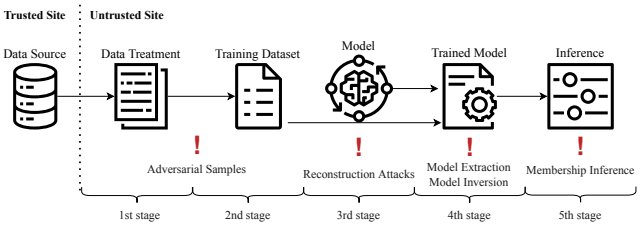


Figure 1: ML pipeline and known attack vectors.

with higher privilege levels (such as the host OS, hypervisor, and BIOS), from accessing their content [21, 23].

Since SGX protects code and data from privileged access, sensitive plaintext data can be processed at the enclave without compromising its privacy. Thus, TEEs outperform typical traditional cryptographic computational techniques (*e.g.*, searchable encryption, homomorphic encryption) [23]. Even though the second generation of SGX has improved the size of the protected memory region, it still defines the Enclave Page Cache (EPC) to 128MB per CPU [12]. When such limitation is met, memory swapping occurs, which is a performance-costing mechanism [11]. Thus, SGX-based solutions must balance the number of I/O operations and the amount of data handled by enclaves as well as the Trusted Computing Base (TCB) to optimize performance.

We chose SGX over other TEEs in this paper because of its broad availability and use in academia [19, 32] and industry [4].

3 THREAT MODEL AND ATTACKS

3.1 Soteria Threat Model

Soteria enables the secure outsourcing of ML training and inference workloads. These are scenarios where the data owner holds sensitive information (a private dataset and/or model) and wants to perform some ML workload on it using an external cloud provider.

Our deployment model is depicted in Figure 2 and is as follows. The client (data owner) will be trusted and will provide input for ML tasks Then, a Spark Master node and N Worker nodes will be deployed in an untrusted environment (cloud provider), equipped with Intel SGX technology. Externally, we also consider a distributed data storage backend. The protocol assumes an implicit setup where the client stores its input data securely within this backend, which is also considered untrusted throughout the protocol execution.

We consider semi-honest adversaries, which means that security is defined according to a threat that attempts to break the confidentiality of data and model, but that will not actively deviate from the protocol specification. This is a good fit for cloud-based systems, where data breaches are common and malicious entities can read internal processing data temporarily [18]. In brief, our security goal is to allow clients to provide input data for training and inference in a way that is not vulnerable to breaches in confidentiality.

3.2 ML Workflow Attacks

Throughout the paper, we will follow the black-box setting of [9]. Essentially, when we state that an adversary has black-box access to a model, it means it can query any input x and receive the predicted class probabilities $P(y|x)$ for all classes y. This allows the adversary

Table 1: Comparison between state-of-the-art solutions and Soteria regarding the safety against ML attacks.

Attacks		Systems [15]	[16]	[25]	[14]*	Soteria
Adversarial	Gradient-based	✗	✗	✓	✗	✓
	Score-based	✗	✗	✓	✗	✓
	Transfer-based	✗	✗	✓	✗	✓
	Decision-based	✗	✗	✓	✗	✓
Model Extraction	Equation-solving	✓	✓	✗	✓	✓
	Path-finding	✓	✓	✗	✗	✓
	Class-only	✓	✓	✗	✗	✓
	DFKD	✓	✓	✗	✓	✓
Model Inversion		✓	✓	?	✓	✓
Reconstruction Attacks		✓	✓	✓	✓	✓
Membership Inference		✗	✗	✗	✗	✓

*Data encryption is not provided on the open-source version.
✓ - Protected; ✗ - Non-protected; ? - Not disclosed.

to interact with the trained model without retrieving additional information, e.g. computing the gradients. Ensuring security against attacks on this pipeline entails including countermeasures against a wide array of attack vectors, as depicted in Figure 1.

Adversarial attacks. These attacks are characterized by the injection of malicious data samples, to manipulate the model and to disclose information about the original data being used for training or inference purposes. Successful attacks in the literature require the attacker to have direct access to the training dataset (data poisoning, transfer-based, and gradient-based attacks), the model and gradients (gradient-based attacks), or the full results and class probabilities (score-based attacks) [6, 20].

Model Extraction. These attacks aim at learning a close approximation to an objective function of the trained model. This approximation is based on the exact confidence values and response labels obtained by inference. To attain the desired output, the attacker must know the dimension of the original training dataset (equation-solving attacks), the dimension of the decision trees, data features and the final confidence values (path-finding attacks), or hold real samples from the training dataset (class-only attacks and data-free knowledge distillation (DFKD)) [28, 29].

Model Inversion and Membership Inference. These attacks target the recovery of values from the training dataset. Both consider an adversary that queries the ML system in a black-box fashion and both are currently based on ML services, which define publicly their trained models and the confidence values. In model inversion, the adversary must have partial knowledge of the training dataset's features to infer and query the model with specific queries [13]. Membership inference aims to test if a specific data point d was used as training data and requires the adversary to know a subset of samples used for training the model (that does not contain d) [26].

Reconstruction attacks. The goal of this attack is similar to that of membership inference, but instead of testing for the existence of a specific data point, the adversary intends to reconstruct raw data used for training the model. To be successful, some attacks require the adversary to have model-specific information, namely feature vectors (*e.g.,* Support Vector Machines or K-Nearest Neighbor) [2], others only require black-box access to the model [24].

Summary. Unlike previous works [14–16, 25], which typically consider a small subset of ML attacks, our proposal aims at providing mechanisms that cover the full range of the above-mentioned exploits. Table 1 presents relevant state-of-the-art solutions, the security attacks covered by these, and the attacks addressed by Soteria. Intuitively, the resilience of our system is the result of combining several mechanisms, which are only partially ensured by other systems: i.) authenticity verification of inputs excludes injections necessary for *adversarial attacks*; ii.) isolation guarantees of our protocol ensure that malicious workers gather no additional information other than statistical data, an essential aspect for preventing most attacks, and iii.) query input via secure channel prevents the adversary from performing arbitrary queries to our system, which is also a central requirement for *model inversion* or *reconstruction attacks*. This is analysed in detail in Section 4.5.

TEE-related security issues such as *side-channel* and *memory access pattern* attacks are considered orthogonal and complementary to our design goals. Indeed, mechanisms such as ObliviousRAM [27] can be layered over Soteria to address these, at the cost of additional performance overhead.

4 SOTERIA

Soteria is a privacy-preserving ML solution that avoids changing Apache Spark's main architecture and processing flow while retaining its usability, scalability, and fault tolerance properties.

4.1 Apache Spark: Architecture and Flow

As depicted in Figure 2, Apache Spark's operational flow is as follows. Before submitting ML tasks (*e.g.*, model training, and/or inference operations) to the Spark cluster, users must load their local datasets and models to a distributed storage backend. Users can then submit ML processing tasks, specified as ML task scripts, to the Spark client, which is responsible for forwarding these scripts to the Master node. At the Master node, tasks are forwarded to the Spark Driver, which generates a Spark Context that then distributes the tasks to a set of Worker nodes.

As Workers may be executing different steps of a given task, they need to be able to transfer information (*e.g.*, model parameters) among each other through the network. After finishing the desired computational steps, Workers send back their outputs to the Master node, which merges the outputs and replies back to the client.

Similar to the regular flow of Apache Spark, Soteria can be divided into two main environments or sides: the Soteria Client, trusted side, and the Soteria Cluster, untrusted side, (*e.g.*, cloud environment). Next, we describe the main modifications required by Soteria to the original Apache Spark's design, depicted in Figure 2 by the white dashed and solid line boxes.

4.2 Soteria Client

Soteria's client module is used by users for three main operations: i) loading data into the distributed storage backend, ii) sending ML training tasks to the Spark cluster, and iii) sending ML inference tasks to the Spark cluster. Soteria does not change the way users typically specify and perform the previous operations. The only exception is that users need to provide additional information in a *Manifest* configuration file, as described next.

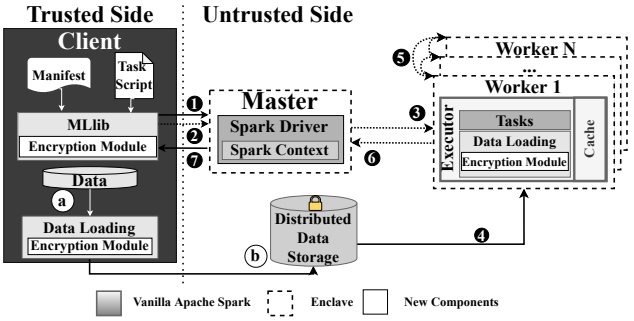


Figure 2: Soteria architecture and operations flow.

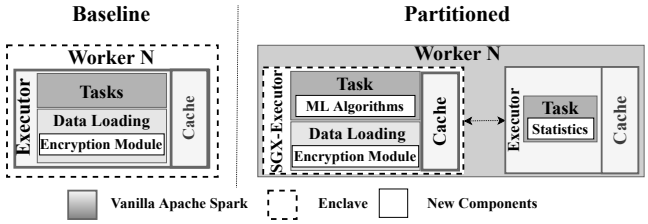


Figure 3: Comparison between Soteria-b and Soteria-p schemes.

Data Loading. For the first operation, the user must specify the data to be loaded to the storage backend. However, such data has to be encrypted before leaving the trusted user premises. This step is done by extending Spark's data loading component with a transparent encryption module (Figure 2-ⓐ), This module encrypts the data being loaded into the distributed storage backend with a symmetric-key encryption scheme (Figure 2-ⓑ).

Tasks submission. ML training and inference operations include two main files: the ML task script and the *Manifest* file. The transparent encryption module, also integrated within MLlib, is used to encrypt the ML task script (Figure 2-❶), which contains sensitive arguments (*i.e,* model parameters) and the ML's workload processing logic, and to decrypt the outputs (*e.g.*, trained model or inference result) returned by Spark's Master node to the client.

The *Manifest* file contains the libraries to be used by the ML task script, as well as the path at the storage backend where the training or inference data, for that specific task, is kept (Figure 2-❷). Briefly, and as explained in the next sections, this file ensures that different Spark components can attest the integrity of libraries and data being used/read by them and, moreover, cannot access other libraries or data that these are not supposed to.

The encryption module is in charge of securely exchanging the *Manifest file*, and the user's symmetric encryption key with the SGX enclave on the Master node (Figure 2-❶❷). This is done once, at the ML task's bootstrapping phase, and requires establishing a secure channel between the client and Master's enclave. This channel guarantees the security and integrity of the user's encryption key and the *Manifest file*, while the encrypted ML task scripts can be safely sent via an unprotected channel.

With the previous design, sensitive data is only accessed in its plaintext format at trusted user premises or inside trusted enclaves. This includes users' encryption keys, the information in the *Manifest file* and ML task scripts, as well as the final output.

4.3 Soteria Cluster

Training and inference ML task scripts are sent encrypted to Spark's Master node to avoid revealing sensitive information. However, the node requires access to the plaintext information contained in these cryptograms to distribute the required computational load across Workers. So, the Spark Driver and Context modules must be deployed in a secure SGX enclave where the cryptograms can be decrypted and the plaintext information can be securely accessed. The cryptograms, however, can only be decrypted if the secure enclave has access to the user's encryption key, thus explaining why the key must be sent through a secure channel established between the client module and the enclave.

For inference operations, the Master node also needs to access the distributed storage backend to retrieve the stored ML model. The user's encryption key is necessary so that the encrypted model is only decrypted and processed at the secure enclave. The *Manifest* file ensures that only the storage locations specified in the file are accessible to the Master Node (Figure 2-❷).

After processing the ML task scripts, the Master's enclave establishes secure channels with the enclaves of a set of Workers to send the necessary computational instructions[1] along with the user's encryption key and *Manifest file* (Figure 2-❸). The user's encryption key is needed at the Worker nodes so that these can read encrypted data (*e.g.,* train dataset or data to be inferred) from the storage backend while decrypting and processing it in a secure enclave environment (Figure 2-❹). The *Manifest file* is used, once again, to prevent unwanted access to stored data. Furthermore, the enclaves at the worker nodes establish secure channels between themselves to transfer sensitive metadata information such as model training parameters (Figure 2-❺).

Finally, after completing the desired computational tasks, the Workers send the corresponding inference or training outputs to the Master node, through the established secure channel (Figure 2-❻). The Master node then merges the partial outputs into the final result and sends it encrypted, with the user's encryption key, to the trusted client module (Figure 2-❼). At the latter, the result (*i.e.,* trained model or inference output) is decrypted by the transparent encryption module and returned to the user in plaintext.

4.4 Soteria Design

Soteria proposes a novel partitioning scheme, Soteria-p, that does fine-grained partitioning of which operations execute inside and outside secure enclaves. Note that this partitioning is only done for ML operations executed at Spark Worker nodes. The remaining operations done at other Spark components (*i.e.*, Master) are always executed inside trusted enclaves.

To better understand the novelty of our partitioning scheme, we first introduce a common state-of-the-art approach, Soteria-b, which is also supported by our system and is used in this paper as a security and performance baseline.

Soteria Baseline (Soteria-b). In Soteria-b, all computation done by Spark Workers is included in a trusted environment. The

[1]The same metadata sent by a vanilla Spark deployment so that Workers know the computational operations to perform.

executor processes launched by each Worker node are deployed inside an enclave, as depicted in Figure 3. Outside the enclave, data is always encrypted in an authenticated fashion, which allows the Worker to decrypt and validate data integrity within the enclave.

Soteria Partitioning Scheme (Soteria-p). Our novel scheme is based on the observation that ML workloads are composed of different computational steps. Some must operate directly over sensitive plaintext information (*e.g.,* train and inference data and model), while others do not require access to this type of data and are just calculating and collecting general statistics about the operations being made. For instance, in a multiclass ML task, where the user may want to predict multiple classes, the evaluation of such an algorithm would need to measure the precision and the probability of each individual class. These measurements can be performed independently of other operations over sensitive information.

Therefore, Soteria-p decouples statistical processing, used for assessing the performance of inference and training tasks, from the actual computation of the ML algorithms done over sensitive plaintext information. This decoupling builds directly upon MLlib and refactors its implementation without requiring any changes to the way users submit ML tasks. As depicted in Figure 3, statistical processing is done by executor processes in the untrusted environment, while the remaining processing endeavors are done by another set of executors inside a trusted enclave.

This decoupled scheme leads Soteria-p to reveal the following statistical information during the execution of ML workloads: the calculation of confidence results (accuracy, precision, recall and F1-scores), table summaries and ROC/AUC curves, and probability distributions for classes.

4.5 Security

Formally, our security goal is defined using the real-versus-ideal world paradigm, similarly to the Universal Composability [7] framework. Succinctly, we prove that Soteria is indistinguishable from an idealized service for running ML scripts in an arbitrary external environment that can collude with a malicious insider adversary. We then use that abstraction to demonstrate how Soteria is resilient to real-world ML attacks. This idealized service is specified as a functionality parametrized with the input data, which simply executes the tasks described in the ML task script, and returns the output to the client via a secure channel.

The full proof of Soteria can be found in part A of [1]. The outline is as follows. The role played by Soteria 's Master node can be seen as an extension of the client, establishing secure channels, providing storage encryption keys, and receiving outputs. We follow the reasoning of [5] and replace the Master node with a reactive functionality performing the same tasks. Similarly, each Soteria Worker behaves simultaneously as a processing node and as a client node, providing inputs to the computation of other Workers (e.g., model training parameters). This enables us to do a hybrid argument, where Worker nodes are sequentially replaced by idealized reactive functionalities executing their roles in the task script.

Finally, all processing is done in ideal functionalities, and all access to external storage is fixed by the ML task script and the Manifest file, so we can refactor the functionalities to process over hard-coded client data, and replace the secure data storage with dummy encryptions. We have now reached the ideal world, where all ML computation is done in an isolated service, and all other protocol interactions are simulated given the ML task script and Manifest files. Our analysis refers to Soteria-b, and thus establishes the baseline security result when no computation is done outside the enclave (no leakage). The reasoning for Soteria-p is identical, with the caveat that statistical data is explicitly revealed as leakage in the ideal world.

4.5.1 Security implications of statistical leakage. To show that our system is resilient against ML attacks, we must consider a common prerequisite for such attacks to be successful: the adversary must have black-box access to the model (Section 3.2). Our result implies that adversaries cannot infer internal data from the workers, and the secure channel between client and Master prevents adversaries from injecting queries into the system. This would intuitively suggest that our adversary is unable to perform queries in a black-box fashion to the model, however, Soteria-p has the aforementioned additional leakage of statistical information.

As such, a crucial security question to answer is: *how does statistical information relate to black-box model access, i.e. does the first imply the second in any way?* Extracting model access from statistical data is an ongoing area of research. However, current attacks suggest one is unable to do this in any successful way [8]. This supports our thesis that statistical values *are not sensitive information*, in the sense that their leakage does not expose our system to these types of attacks. It follows that *Soteria-p scheme is resilient to any attack that requires black-box access to the model to succeed.*

4.5.2 Relation to ML Attacks. We now overview the four types of attacks referred to in Section 3.2 on a case-by-case basis. Part B of [1] contains a more in-depth analysis of these attacks.

Resistance against input forgery is achieved by Soteria through authenticated data encryption. This means that the input dataset is authenticated by the data owner and explicitly defined in the *Manifest file*, allowing enclave Worker nodes to check the authenticity of all input data. Thus, no forged data is accepted for processing, which is necessary for performing any type of *adversarial attack*.

The secure channels between the TEE at the Master node and the client ensure that an external adversary cannot observe legitimate query input/outputs, and cannot submit arbitrary queries to Soteria. This query privacy feature is crucial to block illegitimate model access, which allows us to protect against *model extraction*, *model inversion*, *membership inference* as well as instances of *reconstruction attacks* that require black-box access to the model.

Finally, *reconstruction attacks* require additional knowledge about internal ML model data. Our security result shows that Soteria is indistinguishable from an idealized ML service, which does not reveal the trained model. This includes the important feature vectors required for this attack to occur, which also cannot be inferred from confidence values and class probabilities alone. Alternatively, *reconstruction attacks* requiring black-box access to the model are strictly stronger, but this, as we have argued, is not possible only with knowledge of confidence values, class probabilities, ROC/AUC curves, and table summaries (the explicit leakage of Soteria-p) [1].

Table 2: Representation of the tasks of each ML algorithm and the data sizes for different workloads.

Algorithms	Tasks	Workloads			
		Tiny	Large	Huge	Gigantic
ALS	RS	193KB	345MB	2GB	4GB
PCA	DR	256KB	92MB	550MB	688MB
GBT	P	36KB	46MB	92MB	183MB
LR	C + P	11GB	134GB	335GB	894GB

RS: Recommendation Systems; DR: Dimensionality Reduction; P: Prediction; C: Classification.

4.6 Implementation

Soteria's prototype is built on top of Apache Spark 2.3.0 and implemented using both Java and Scala. Spark's data loading library was extended to include Soteria's transparent encryption module. The latter uses the AES-GCM-128 authenticated encryption cypher mode, which provides both data privacy and integrity guarantees.

Both Soteria-b and Soteria-p schemes are supported by our prototype. For Soteria-p's implementation, Spark's MLlib implementation was decoupled into two sub-libraries, one with the statistical processing (to be executed outside SGX), and another with the remaining ML computational logic (to be executed inside SGX).

Graphene-SGX 1.0 was used for the overall management of Intel SGX enclaves' life cycle, for specifying the computation (*i.e.*, internal Spark and MLlib libraries) to run at each enclave, and for establishing secure channels (*i.e.*, with the TLS-PSK protocol) between the enclaves at the Master and Worker nodes [30]. Soteria's *Manifest* file was also provided by Graphene.

5 EVALUATION

Our evaluation answers three main questions: *i) How does Soteria impacts the execution time of ML workloads? ii) How does the Soteria-p scheme compares, in terms of performance, with state-of-the-art approaches (*i.e.*, Soteria-b and SGX-Spark)? iii) Can Soteria efficiently handle different algorithms and dataset sizes?*

5.1 Methodology

Environment. The experiments use a cluster with eight servers, with a 6-core 3.00 GHz Intel Core i5-9500 CPU, 16 GB RAM, and a 256GB NVMe. The host OS is Ubuntu 18.04.4 LTS, with Linux kernel 4.15.0. Each machine uses a 10Gbps Ethernet card connected to a dedicated local network. We use Apache Spark 2.3.0 and version 2.6 of the Intel SGX Linux SDK (driver 1.8). The client and Spark Master run in one server while Spark Workers are deployed in the remaining seven servers. SGX memory is configured to use 4GB.

Workloads. We resort to the HiBench benchmark [17] for evaluating four ML algorithms (Table 2), that are broadly used and natively implemented on top of MLlib, namely: Alternating Least Squares (ALS), Principal Component Analysis (PCA), Gradient Boosted Trees (GBT) and Linear Regression (LR). For each algorithm, the benchmark suite offers different workload sizes ranging from *Tiny* to *Gigantic* configurations.

Setups and metrics. To validate Soteria's performance, and the benefits of fine-grained differentiation of secure ML operations, we compare the implementations of our system with the Soteria-b and Soteria-p schemes. These setups are compared with a deployment of Apache Spark that does not offer privacy guarantees (Vanilla).

Moreover, we test SGX-Spark [14], a state-of-the-art SGX-based solution that protects both analytical and ML computation done with Apache Spark. It is designed to process sensitive information inside SGX enclaves, so it can be considered the most similar to Soteria. However, SGX-Spark can only guarantee that *User Defined Functions (UDFs)* are processed in secure enclaves. This decision leaves a large codebase of Spark outside the protected memory region and, consequently, limits the users to only being able to execute privacy-preserving ML algorithms based on UDFs.

For each experiment discussed in the next section, we include the average algorithm execution time and standard deviation for 3 independent runs. The *dstat* monitoring tool was used to collect the CPU, RAM, and network consumption at each cluster node.

5.2 Performance Overview

Figures 4a, 4b, 4c and 4d present the performance evaluation for *PCA*, *GBT*, *ALS* and *LR* algorithms for different workload sizes. Next, we list our main observations to aid in the characterization of these results. Unless stated otherwise, the performance overhead values discussed in this section correspond to the number of times that the algorithm's execution time increases for a given setup when compared to the Vanilla Spark deployment results. *Obs. 1* to *5* correspond to the *Huge* workload for the defined algorithms, whilst *Obs. 6* to *9* refer to the overall results in Figure 4.

Observation 1. Vanilla Spark's execution times for ALS, PCA, LR, and GBT algorithms are, respectively, 55, 655, 657, and 189 seconds.

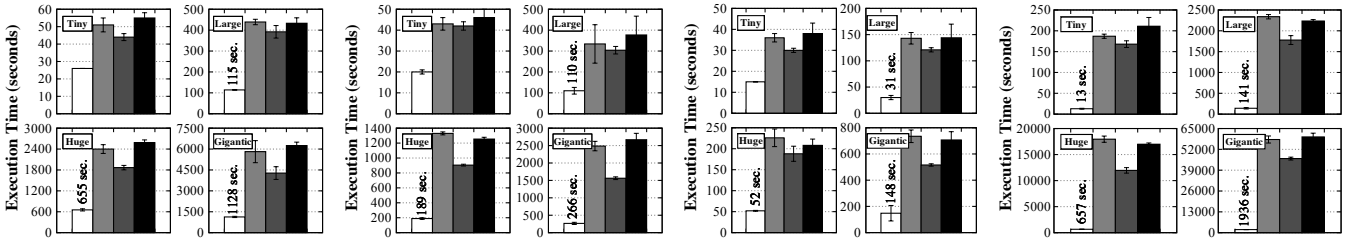
Observation 2. The execution time for ALS increases by 3.62x and 4.35x for Soteria-p and Soteria-b, respectively. SGX-Spark incurs an execution overhead of 4x. Thus, the three setups have similar results while requiring approximately 150 seconds more processing time than the vanilla deployment. Nevertheless, Soteria-p performs slightly better than the other two approaches.

Observation 3. For PCA, Soteria-b and Soteria-p have an execution overhead of 3.67x and 2.85x, while SGX-Spark increases the computational time by 3.95x. When compared to SGX-Spark, Soteria-p decreases the execution time by 12 minutes (27.8%).

Observation 4. For LR, Soteria-b and SGX-Spark exhibit an overhead of 27.31x, while Soteria-p reduces this value to 18.2x. This reduction of 29.6% allows Soteria-p to complete this workload 1.4 hours earlier.

Observation 5. With the GBT algorithm, Soteria-b shows similar execution times when compared to SGX-Spark, with a 7.04x and 6.64x increase, respectively. Soteria-p outperforms both approaches, with an overhead of 4.79x, 27.8% less than SGX-Spark.

Observation 6. For *Tiny* and *Large* workloads with the PCA algorithm, Soteria performs similarly for our two schemes, while outperforming SGX-Spark. With larger workload sizes, the overhead imposed by our solutions increases, however, it continues to show better performance than SGX-Spark. Soteria-b has an overhead of 1.96x to 5.15x for *Tiny* and *Gigantic* workloads, whilst Soteria-p incurs an overhead of 1.72x to 3.79x. When compared with SGX-Spark, the results show an absolute difference of 4 seconds and 7 minutes (7%), for Soteria-b, and 7 seconds and 33 minutes (19% and 31%) respectively, for Soteria-p.



(a) PCA (b) GBT (c) ALS (d) LR

Figure 4: Runtime execution for PCA, GBT, ALS and Linear Regression for *Tiny, Large, Huge* and *Gigantic* workloads. The legend is as follows: □ Vanilla Spark; ■ SOTERIA-B; ■ SOTERIA-P; ■ SGX-Spark.

Observation 7. Regarding the GBT algorithm, and the *Tiny* workload, the overhead of SOTERIA-B, SOTERIA-P, and SGX-Spark are similar. However, the difference between the three approaches is more visible when increasing the workload size. SOTERIA-P (*Tiny-2.13x and Gigantic-5.88x*) outperforms both approaches, while SOTERIA-B (*Tiny-2.18x, Gigantic-9.35x*) and SGX-Spark (*Tiny-2.3x, Gigantic-10.34x*) have similar results. SOTERIA-P is able to surpass SGX-Spark's execution time in the *Gigantic* workload by up to 41%.

Observation 8. With ALS, SOTERIA-P shows an execution time overhead of 2.04x and 3.28x, for the *Tiny* and *Gigantic* workloads, respectively. SOTERIA-P achieves lower overhead than SOTERIA-B and SGX-Spark for all dataset sizes, with the execution time decreasing by 8 seconds (9%) for the *Tiny* and 191 seconds (27%) for the *Gigantic* workloads.

Observation 9. For LR, with the *Tiny* workload, SOTERIA-B and SOTERIA-P increase execution time by 14.39x and 12.95x, respectively. As for the *Gigantic* workload, SOTERIA-B incurs an overhead of 30.04x and SOTERIA-P of 23.89x. Compared to SGX-Spark, our SOTERIA-P decreases the execution time by 43 seconds for the *Tiny* workload and by 4.31 hours for the *Gigantic* workload (22.6%).

Observation 10. Overall, the CPU, RAM, and network usage for both SOTERIA schemes is similar to the vanilla Spark baseline. In more detail, SOTERIA-B with LR presents the upper-bound limit for both memory and CPU, showing an increase of 9% in both when compared with vanilla Spark (20%). Whilst the network shows an upper-bound increase of 10% (vanilla Spark shows an upper-bound network of 135MB) in SOTERIA-B with PCA due to extra encrypted data paddings being sent between Spark Workers.

Observation 11. SOTERIA does not impact the accuracy of ML workloads. For all experiments, we measured the corresponding accuracy metrics (*e.g.*, accuracy, root mean square error, or ROC). The results corroborate that both SOTERIA-B and SOTERIA-P show accuracy values similar to the vanilla Spark version.

5.3 Analysis

We analyze the results based on *i)* dataset size; and *ii)* size of trusted computing base (TCB).

Dataset size. For PCA, GBT, and ALS with smaller datasets, SOTERIA-B and SOTERIA-P perform similarly (Figure 4). However, as the size of the datasets increases, more operations and data must be transferred to the SGX enclave, thus taking a more noticeable toll on the overall performance. The page swapping mechanism of SGX, which occurs due to its memory limitations, incurs a significant performance penalty [11, 12]. For example, when compared to the vanilla setup, the PCA algorithm overhead for SOTERIA-B varies between 1.96x for *Tiny* workload and 5.15x for *Gigantic* workload. While for SOTERIA-P, the execution time increases 1.78x in the *Tiny* workload and 3.79x in the *Gigantic* workload.

SOTERIA-P is the setup that scales better as the amount of data to be processed grows. Indeed, as seen in *Obs. 6-9*, it is able to reduce execution time from 9% up to 31% when compared to SGX-Spark.

Size of TCB. SGX-Spark outperforms SOTERIA-B for some of the evaluated algorithms (*Obs. 2, 4, and 5*). As SGX-Spark only protects UDFs, the performance overhead imposed by the larger TCB of SOTERIA-B is higher. Nevertheless, when compared to SGX-Spark, SOTERIA-B covers a wider range of ML attacks, while keeping performance overhead below 1.59x. Indeed, for algorithms such as PCA, SOTERIA-B has similar or slightly inferior execution times (*Obs. 3*) which is due to both setups performing similar computations at the enclaves while the UDF mechanism is not fully optimized.

On the other hand, SOTERIA-P always outperforms SGX-Spark and SOTERIA-B (*Obs. 2-5*). This is due to the TCB reduction present in our novel partitioning scheme. The results show that this solution can reduce the training time by up to 30%, namely for the LR algorithm with the *Huge* workload (*Obs. 4*).

Discussion. The results show that SOTERIA-P outperforms other state-of-the-art approaches, namely SGX-Spark, for all the considered ML algorithms. Also, SOTERIA-P achieves better performance than the SOTERIA-B setup, while offering similar security guarantees when considering distinct ML attacks (Section 4.5). This is made possible by filtering key operations to be done outside enclaves.

In detail, when compared to SOTERIA-B, SOTERIA-P reduces ML workloads' execution time by up to 37%. When compared with SGX-Spark, the execution time is reduced by up to 41%. Interestingly, for the LR algorithm using a *Gigantic* workload (*894GB*), SOTERIA-P decreases computation time by 4.3 hours and 3.3 hours, when compared with SGX-Spark and SOTERIA-B, respectively. The performance overhead of SOTERIA-P for the four different algorithms ranges from 1.7x to 23.8x when compared to Vanilla Spark.

6 RELATED WORK

Privacy-preserving ML with TEEs. Chiron [15] enables training ML models on a cloud service without revealing information about the training dataset. *Myelin* [16] offers a similar solution to *Chiron* while adding differential privacy and data oblivious protocols to the algorithms to mitigate the exploits from *side-channels* and the information leaked by the model parameters. SOTERIA differs from these works as it is able to cover both the training and inference

phases while providing additional protection against *adversarial samples*, *reconstruction*, and *membership inference* attacks (Table 1). In [23], five ML algorithms are re-implemented with data oblivious protocols. These protocols combined with TEEs ensure strong privacy guarantees while preventing the exploitation of *side-channel* attacks that observe memory, disk, and network access patterns to infer private information. Unlike this solution, Soteria aims at transparently supporting all ML algorithms built with MLlib.

Privacy-preserving analytics with TEEs. TEEs have also been used to ensure privacy-preserving computation for general-purpose analytical frameworks [25]. In comparison to SGX-Spark [14], detailed in Section 5.1, Soteria supports a broader set of algorithms (*i.e.*, any algorithm that can be built with the MLlib API), while protecting users from a more complete set of ML attacks (Table 1). Opaque [32] and Uranus [19] resort to SGX to provide secure general-purpose analytical operations, while only supporting a restricted set of ML algorithms. Opaque combines SGX with oblivious protocols and requires the re-implementation of default Apache Spark UDF operators. Uranus is also based on porting UDF processing to SGX enclaves but includes a single ML workload. Differently, Soteria is targeted at ML workloads and is not limited by UDF-based algorithms that, when compared with MLlib-based ones, exhibit lower performance for some ML workloads [10]. Therefore, the design, implementation, and security requirements to be considered are distinct when comparing with Soteria.

7 CONCLUSION

We propose Soteria, a system for distributed privacy-preserving ML. Our solution builds upon the combination of Apache Spark and TEEs to protect sensitive information being processed at third-party infrastructures during the ML training and inference phases.

The innovation of Soteria stems from a novel partitioning scheme (Soteria-p) that allows specific ML operations to be deployed outside trusted enclaves. Namely, we show that it is possible to offload non-sensitive operations (*i.e.*, statistical calculations) from enclaves, while still covering a larger spectrum of black-box ML attacks than in previous related work. Also, this decision enables Soteria to perform better than existing solutions, such as SGX-Spark, while reducing ML workloads execution time by up to 41%.

ACKNOWLEDGMENTS

We thank Ricardo Macedo for his valuable input. This work was supported by the Portuguese Foundation for Science and Technology through project LA/P/0063/2020 (João Paulo), and by Ph.D. Fellowship (SFRH/BD/146528/2019) and project AIDA (Cláudia Brito), with reference POCI-01-0247-FEDER-045907, co-financed by the ERDF - European Regional Development Fund, through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme under the Portugal 2020 Partnership Agreement, on the scope of the CMU Portugal Program.

REFERENCES

[1] [n. d.]. SOTERIA Proof. https://dbr-haslab.github.io/repository/sac23.pdf.
[2] Mohammad Al-Rubaie and J Morris Chang. 2019. Privacy-preserving machine learning: Threats and solutions. *IEEE Security & Privacy*.
[3] Yoshinori Aono, Takuya Hayashi, Lihua Wang, Shiho Moriai, et al. 2017. Privacy-preserving deep learning via additively homomorphic encryption. *IEEE Transactions on Information Forensics and Security*.
[4] Microsoft Azure. [n. d.]. Azure Confidential Computing. https://azure.microsoft.com/en-us/solutions/confidential-compute/. (Accessed on 22/10/2022).
[5] Raad Bahmani, Manuel Barbosa, Ferdinand Brasser, Bernardo Portela, et al. 2017. Secure multiparty computation from SGX. In *International Conference on Financial Cryptography and Data Security*. Springer.
[6] Wieland Brendel, Jonas Rauber, and Matthias Bethge. 2018. Decision-Based Adversarial Attacks: Reliable Attacks Against Black-Box Machine Learning Models. In *6th International Conference on Learning Representations,*.
[7] R. Canetti. 2001. Universally composable security: A new paradigm for cryptographic protocols. In *42nd IEEE Symposium on Foundations of Computer Science*.
[8] Varun Chandrasekaran, Kamalika Chaudhuri, Irene Giacomelli, Somesh Jha, et al. 2020. Exploring connections between active learning and model extraction. In *29th USENIX Security Symposium*.
[9] Pin-Yu Chen, Huan Zhang, Yash Sharma, Jinfeng Yi, and Cho-Jui Hsieh. 2017. Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models. In *10th ACM workshop on artificial intelligence and security*.
[10] Databricks. [n. d.]. Optimizing Apache Spark UDFs. https://www.databricks.com/session_eu20/optimizing-apache-spark-udfs. (Accessed on 27/10/2022).
[11] Tu Dinh Ngoc, Bao Bui, Stella Bitchebe, Alain Tchana, et al. 2019. Everything you should know about Intel SGX performance on virtualized systems. *ACM on Measurement and Analysis of Computing Systems*.
[12] Muhammad El-Hindi, Tobias Ziegler, Matthias Heinrich, Adrian Lutsch, et al. 2022. Benchmarking the Second Generation of Intel SGX Hardware. In *Data Management on New Hardware*.
[13] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. 2015. Model inversion attacks that exploit confidence information and basic countermeasures. In *22nd ACM SIGSAC Conference on Computer and Communications Security*.
[14] Large-Scale Data & Systems (LSDS) Group. [n. d.]. SGX-Spark. https://github.com/lsds/sgx-spark. (Accessed on 22/10/2022).
[15] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, et al. 2018. Chiron: Privacy-preserving machine learning as a service. *arXiv preprint arXiv:1803.05961*.
[16] Nick Hynes, Raymond Cheng, and Dawn Song. 2018. Efficient deep learning on multi-source private data. *arXiv preprint arXiv:1807.06689*.
[17] Intel. [n. d.]. HiBench is a big data benchmark suite. https://github.com/Intel-bigdata/HiBench. (Accessed on 22/10/2022).
[18] Salman Iqbal, Miss Laiha Mat Kiah, Babak Dhaghighi, Muzammil Hussain, Suleman Khan, Muhammad Khurram Khan, and Kim-Kwang Raymond Choo. 2016. On cloud security attacks: A taxonomy and intrusion detection and prevention as a service. *Journal of Network and Computer Applications*.
[19] Jianyu Jiang, Xusheng Chen, TszOn Li, Cheng Wang, et al. 2020. Uranus: Simple, efficient sgx programming and its applications. In *15th ACM Asia Conference on Computer and Communications Security*.
[20] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. 2017. Adversarial Machine Learning at Scale. In *5th International Conference on Learning Representations*.
[21] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, et al. 2013. Innovative instructions and software model for isolated execution. *Hasp isca*.
[22] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, et al. 2016. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*.
[23] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, et al. 2016. Oblivious multi-party machine learning on trusted processors. In *25th USENIX Security Symposium*.
[24] Ahmed Salem, Apratim Bhattacharya, Michael Backes, Mario Fritz, and other. 2020. Updates-leak: Data set inference and reconstruction attacks in online learning. In *29th USENIX Security Symposium*.
[25] Fahad Shaon, Murat Kantarcioglu, Zhiqiang Lin, and Latifur Khan. 2017. Sgx-bigmatrix: A practical encrypted data analytic framework with trusted processors. In *ACM SIGSAC Conference on Computer and Communications Security*.
[26] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership inference attacks against machine learning models. In *Symposium on Security and Privacy (SP)*.
[27] Emil Stefanov, Marten Van Dijk, Elaine Shi, T-H Hubert Chan, et al. 2018. Path ORAM: an extremely simple oblivious RAM protocol. *J. ACM*.
[28] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and other. 2016. Stealing machine learning models via prediction apis. In *25th USENIX Security Symposium*.
[29] Jean-Baptiste Truong, Pratyush Maini, Robert J Walls, and Nicolas Papernot. 2021. Data-free model extraction. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*.
[30] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-sgx: A practical library OS for unmodified applications on SGX. In *USENIX Annual Technical Conference*.
[31] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, et al. 2016. Apache spark: a unified engine for big data processing. *Commun. ACM*.
[32] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, et al. 2017. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation*.