SGX-PySpark: Secure Distributed Data Analytics

Do Le Quoc
TU Dresden, Scontain UG

Franz Gregor
TU Dresden, Scontain UG

Jatinder Singh
University of Cambridge

Christof Fetzer
TU Dresden, Scontain UG

ABSTRACT

Data analytics is central to modern online services, particularly those data-driven. Often this entails the processing of large-scale datasets which may contain private, personal and sensitive information relating to individuals and organisations. Particular challenges arise where cloud is used to store and process the sensitive data. In such settings, security and privacy concerns become paramount, as the cloud provider is trusted to guarantee the security of the services they offer, including data confidentiality. Therefore, the issue this work tackles is *"How to securely perform data analytics in a public cloud?"*

To assist this question, we design and implement SGX-PySpark– a secure distributed data analytics system which relies on a trusted execution environment (TEE) such as Intel SGX to provide strong security guarantees. To build SGX-PySpark, we integrate PySpark - a widely used framework for data analytics in industry to support a wide range of queries, with SCONE - a shielded execution framework using Intel SGX.

CCS CONCEPTS

• **Information systems → Data analytics**; • **Security and privacy → Distributed systems security**.

KEYWORDS

Confidential computing; data analytics; security; distributed system

ACM Reference Format:
Do Le Quoc, Franz Gregor, Jatinder Singh, and Christof Fetzer. 2019. SGX-PySpark: Secure Distributed Data Analytics. In *Proceedings of the 2019 World Wide Web Conference (WWW '19), May 13–17, 2019, San Francisco, CA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3308558.3314129

1 INTRODUCTION

Cloud-based services are used to collect, process and analyze large amounts of user's personal data, some of it highly sensitive, such at that relating to personal finances, political views, health, and so forth. Indeed, we have seen increasing attention of regulators on issues regarding the way in which personal data is handled and processed - the EU's General Data Protection Regulation a case in point. Thus, confidentiality and integrity of the data processing in clouds are becoming more important, not least because of increased demands for accountability regarding service providers, and the potential serious legal consequences (and fines) for the data mishandling, mismanagement and leakage, and more generally, for failing to implement the appropriate security measures [9]. Service providers must ensure that data is always protected, i.e., at rest, during transmission, and computation. Many organisations make use of public cloud services for the processing to reduce time and computation cost. This setting is vulnerable to many security threats, e.g., data breaches [10]. Concerns are compounded when we consider attacks from inside the cloud provider, where attackers might have root privileges and/or physical access to machines deployed at the service providers' premises. Therefore, to protect the sensitive data and the analytics computation over the data, service providers cannot rely solely on the operating system access control nor their security policy-based mechanisms.

An promising approach to helping resolve these security challenges is to make use of Trusted Execution Environments (TEEs), such as Intel Software Guard Extensions (SGX). Intel SGX protects the confidentiality and integrity of application code and data even against privileged attackers with root access and physical access. In general, Intel SGX provides an isolated secure memory area called *enclaves*, where the code and data can be executed safely. These security guarantees are solely provided by the CPU, thus even if system software is compromised, the attacker can never access the enclave's content. This approach supports data analytics at processor speeds while ensuring the security guarantee for both computation and sensitive data.

While promising at first glance, to build a practical secure data analytics system using TEEs, e.g., Intel SGX, we need to deal with several challenges. ($\mathcal{A}$) In the current version, Intel SGX supports only a limited memory space (~ 94MB) for applications running inside enclaves. Meanwhile, most big data analytics systems (e.g., Hadoop and Apache Spark [1]) are extremely memory-intensive, since these systems are almost always based on Java Virtual Machine (JVM). ($\mathcal{B}$) Intel SGX still suffers from side-channel attacks [12]. These side-channel attacks happen both at memory level [12, 16] and network level [16]. ($\mathcal{C}$) Deployment and bootstrapping of a data analytics framework to run inside enclaves is not trivial, in fact, challenging. Securely transferring configuration secrets such as certificates, encryption keys and passwords to start the framework inside enclaves is complicated because these secrets need to be protected on the network as well as securely moved into the enclaves. ($\mathcal{D}$) Typically, Intel SGX requires users to heavily modify the source code of their application to run inside enclaves. Thus, transparently supporting a unmodified distributed data analytics framework to run inside enclaves is not a trivial task.

In the context of building secure data analytics systems using Intel SGX, VC3 [13] is one of the first works that applied SGX technology for Hadoop MapReduce framework. VC3 handles challenges

WWW '19, May 13–17, 2019, San Francisco, CA, USA

ACM ISBN 978-1-4503-6674-8/19/05.
https://doi.org/10.1145/3308558.3314129

($\mathcal{A}$) and ($\mathcal{D}$) by using C/C++ to implement the framework and support unmodified Hadoop. However, the challenge ($\mathcal{B}$) is outside the scope of this work. Recently, Opaque [16] overcomes this issue by introducing a oblivious mechanism to hide access patterns at network level. Opaque provides secure data analytics on Apache Spark framework using Intel SGX. It deals with ($\mathcal{A}$) by reimplementing SQL operators for Spark SQL Catalyst engine [6] using C++. These operators run inside enclaves and communicate with Scala code of Spark using a JNI interface. Opaque supports common operators including map, reduce, filter, sort, aggregation, and join, but not all operators of Apache Spark. This means that it does not handle the challenge ($\mathcal{D}$) completely. In addition, it does not support *remote attestation* to verify the integrity of the code running inside SGX enclaves. It also does not handle the challenge (C) since it does not provide a secrets transferring mechanism for execution inside enclaves. Finally, Opaque requires to run Spark master/driver at client side or in a trusted domain. This might affect significantly the performance of the system.

In this work, we overcome these limitations by building a secure data analytics system called SGX-PySpark. We handle the challenge ($\mathcal{A}$) by using PySpark [3], a system built on top of Apache Spark to support data analytics using Python processes. Instead of running a whole JVM inside an enclave to secure Apache Spark or reimplementing operators in C/C++ as Opaque, we run only Python processes inside enclaves since these processes perform analytics over encrypted data. Thus, our system supports out-of-the-box operators of PySpark (the challenge ($\mathcal{D}$)), i.e., users do not need to modify their source code. To run Python processes inside Intel SGX enclaves, our system makes use of SCONE [4, 7] a shielded execution framework which enables unmodified applications to run inside Intel SGX enclaves.

In addition, SGX-PySpark, with the help of SCONE, supports a remote attestation mechanism to ensure the code and data running inside enclaves are correct and not modified by an attacker. SGX-PySpark also copes with challenge (C) by providing a mechanism to securely transfer secrets (keys and certificates) to Python processes running inside enclaves. To handle challenge ($\mathcal{B}$), SGX-PySpark protects its execution against side channel attacks at memory level using a mechanism integrated with SCONE, called Varys [12]. Finally, the design of SGX-PySpark allows users to run the Spark driver/master in the same infrastructure as workers.

2 BACKGROUND

2.1 Intel SGX

Intel SGX is an ISA extension which is a set of special CPU instructions for Trusted Execution Environments (TEE). These instructions enable applications to create *enclaves* – protected areas in the applications address space to provide strong confidentiality and integrity guarantees against adversaries with privileged root access. Intel SGX enables trusted computing by isolating the environment of each enclave from untrusted applications outside the enclave. In addition, by offering the remote attestation mechanism, Intel SGX allows a remote party to attest the application executing inside an enclave [8].

The enclave memory is acquired from Enclave Page Cache (EPC)—a dedicated memory region protected by an on-chip Memory Encryption Engine (MEE). The MEE transparently encrypts cache lines on cache-line evictions and decrypts and verifies cache lines with on cache-line loads. The EPC cannot be directly accessed by non-enclave applications including operating systems. To support multiple enclaves on a system, the EPC is partitioned into 4KB pages which can be assigned to various enclaves. Currently, the size of EPC is limited to 128MB in which only ~ 94MB can be used for user applications and the rest is used to store SGX metadata. Fortunately, SGX supports a secure paging mechanism to an unprotected memory region even though the paging mechanism may introduce significant overheads.

The EPC is managed as the rest of the physical memory by an operating system (or a hypervisor in virtualized environments). The operating system makes use of SGX instructions to allocate and free EPC pages for enclaves. In addition, the operating system is supposed to expose the enclave services (creation and management) to applications. Since the operating system cannot be trusted, the SGX hardware verifies the correctness of EPC pages allocations and denies any operations that would violate the security guarantees. For example, the SGX hardware will not allow the operating system to allocate the same EPC page for different enclaves.

2.2 SCONE

Our system builds on SCONE [7] – a shielded execution framework to enable unmodified applications to run inside SGX enclaves. In the SCONE platform, the source code of an application is recompiled against a modified standard C library (SCONE libc) to facilitate the execution of system calls. The address space of the application stays within an enclave, and the application only can access the untrusted memory via the system call interface.

SCONE uses the compiler-based approach to prepare and build native applications for executing inside SGX enclaves. SCONE applies its mechanism into GNU Compiler Collection (GCC) tool-chain to change the compiling process such that it can build position independent, statically linked code, and eventually linked with the starter program. Therefore, SCONE natively supports C/C++ applications. For Python applications e.g., PySpark executors, we need to compile the CPython/PyPy interpreter with SCONE to run these Python processes inside SGX enclaves. Similarly, to run a Java application inside an enclave, we compile JVM with SCONE.

2.3 PySpark

PySpark is built on top of Apache Spark [1] to provide the Python API for users. Thus, before explaining PySpark, it is useful to understand what is Apache Spark. Apache Spark [1] is an open-source large-scale data analytics framework. Today, it has become the most popular and widely used big data framework in both academia and industry. Comparing to earlier frameworks such as Hadoop MapReduce, Spark is much faster in processing large-scale datasets since it enables the in-memory computing concept where intermediate data is cached in memory to reduce latency [11].

For the in-memory computation, Spark introduces the core abstraction – Resilient Distributed Datasets (RDDs) [15] for distributed data-parallel computing. An RDD is an immutable and fault-tolerant

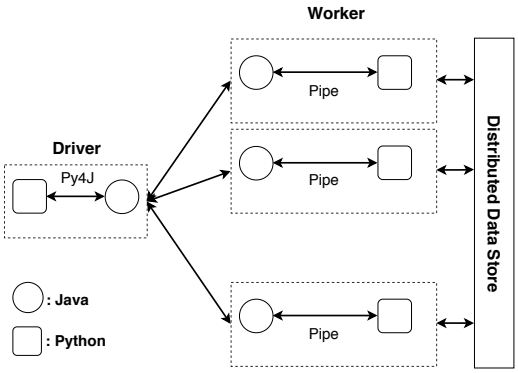

Figure 1: PySpark's architecture.

collection of elements (objects) that is distributed or partitioned across a set of nodes in a cluster [1].

RDDs support two types of operations: `transformations` and `actions`. Transformations return a new RDD, such as *map()* and *filter()*, and actions are operations performed on RDDs that return the output to the driver program or store it in a persistent storage system (e.g *reduce()*, *count()*, *collect()*, *saveAsTextFile()*, etc). Transformations are performed lazily, i.e., they are computed whenever an action operation is invoked [11].

In a deployment, Spark contains a main program called *driver* which coordinates the processes (tasks) executed in other nodes (i.e., workers) in a cluster. When a Spark application is submitted to the driver, it will split the job into tasks and perform scheduling to dispatch tasks to workers. The workers then spawn processes (called *executors*) to handle received tasks.

Since Python supports powerful libraries such as scipy, numpy, scikit-learn, and pandas with a simple and concise syntax, it is a favorite language of data scientists. For that reason, PySpark has been proposed as a Spark programming model to Python. Figure 1 shows the high-level architecture of PySpark.

PySpark extends the Spark runtime to enable executing Python programs on top of Spark. Typically, a PySpark job (a Python process) is submitted, and a JVM is started to communicate with the Python process using Py4J [2]. The Python process creates a *SparkContext* object in the JVM and the SparkContext orchestrates the computation as the regular Spark framework (to reuse almost the same Spark infrastructure). However, the difference is that in PySpark, the executors are Python processes. Each Python process (per CPU core) takes care of the execution of the assigned tasks. Each worker (JVM-based program) submits received tasks to the pool of Python processes and communicates with them using Pipes. The Python processes perform the computation and store back resulting data to an RDD (as pickle objects) in the JVM.

3 SGX-PYSPARK

Our idea to design SGX-PySpark— a secure distributed data analytics system using Intel SGX, is quite simple. We execute only sensitive parts, i.e., the computation parts that process the input sensitive data, inside enclaves. The computation parts outside of enclaves can only access encrypted data. In general, we first encrypt the input data and upload the encrypted data into a distributed storage in an untrusted infrastructure (e.g., a public cloud). Thereafter, SGX-PySpark decrypts and processes the encrypted data inside enclaves in a distributed manner.

Figure 2 illustrates SGX-PySpark's architecture. SGX-PySpark consists of two main components: *(i)* Configuration and attestation service (CAS) component and *(ii)* PySpark with integration with the SCONE library to run inside enclaves. SGX-PySpark maintains the native layout of PySpark (see §2.3).

To secure data analytics computations on top of PySpark, SGX-PySpark runs the driver and the Python processes inside Intel SGX enclaves using the SCONE library. We need to ensure the confidentiality and integrity of the driver, since it is responsible for splitting and scheduling tasks in the system. We protect the Python processes since they are the sensitive computation parts since they directly decrypt and process the input data. Note that in SGX-PySpark, we encrypt not only the input sensitive data but also the computation over it (Python code of analytics jobs).

3.1 Trusted Enclave-based Driver

In PySpark, when we submit a job, the Python driver program makes use of Py4J to start a JVM (Spark driver) and create a JavaSparkContext. This JavaSparkContext orchestrates the job as a regular Spark framework. The JVM-based process takes the responsibility to convert a submitted job into tasks. In detail, it first converts the logical DAG of operations in the submitted job into a physical execution plan, i.e., it divides the DAG into a number of stages. Thereafter, it divides these stages into smaller tasks. Next, the Spark scheduler distributes these tasks to executors (Python processes) deployed on worker nodes for execution (see §2.3).

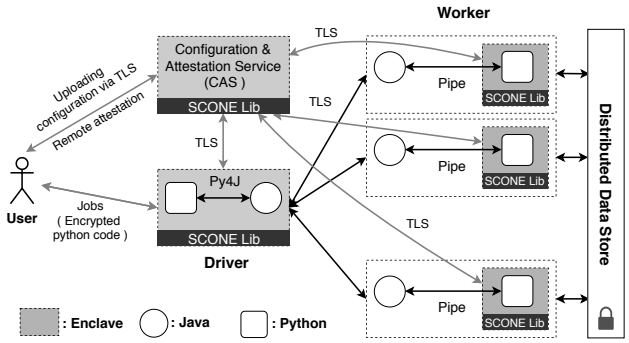

Figure 2: SGX-PySpark's architecture.

To provide secure data analytics, we need to protect the driver as it is the command center of the whole system. To achieve this, in SGX-PySpark, we recompile JVM running in the driver node with the SCONE library to execute the whole driver inside an SGX enclave.

3.2 Trusted Enclave-based Executors

To execute the analytics job (Python code), PySpark launches Python processes (worker processes) and communicates with them using pipes to transfer the user-specified Python code and processed data. Since these Python processes interact directly with the input

data, we protect them against malicious activities, i.e., ensure the integrity and confidentiality by running them inside enclaves with the help of the SCONE platform. Note that, in SGX-PySpark both input data and computation (Python code) are encrypted before upload to the system. They are decrypted inside enclaves using keys transparently obtained from CAS (see §3.3).

3.3 Configuration and Remote Attestation Service

In SGX-PySpark, we need to make sure that the shared secrets such as certificates, passwords to start PySpark, or keys for encrypting/decrypting the input data and computations can never be revealed to untrusted components. Furthermore, we need to securely transfer these secrets to the driver and executors (Python processes) running inside enclaves (see challenge (C) in §1). To achieve these goals, we extend SCONE with a configuration and attestation service (CAS) that transfers security secrets only to the components that have authenticated themselves successfully against it. CAS enhances the Intel attestation service [8] to bootstrap and establish trust across the machines running SGX-PySpark and maintain a secure configuration for the system. In detail, CAS remotely attests the driver and worker processes of PySpark running inside enclaves, before providing encryption/decryption keys and other configuration parameters. CAS is itself launched inside an SGX enclave. A user of SGX-PySpark, first encrypts the input data and his data analytics job and then uploads the secrets (cryptographic keys) and system configurations to CAS. A basic implementation of CAS is presented in [14]. In addition, to guarantee the correctness of computation in SGX-PySpark, we design and implement an auditing service in CAS to keep result logs during runtime. This auditing service protects our data analytics system against rollback attacks.

3.4 Network and File System Protection

File system shield. To protect integrity and confidentiality of the Python codes and the sensitive input data stored on disk, we design in SGX-PySpark a file system shield using SCONE library. In the case SGX-PySpark would write computation results to a file, the shield encrypts the contents before writing. The shield ensures the integrity of these files by keeping their metadata inside CAS (see §3.3). The keys to encrypt the computation content are different from the secrets used by the SGX implementation. They are instead a part of configuration that is uploaded to CAS by users at the startup time of the data analytics system.

Network shield. In SGX-PySpark, to protect the confidentiality of secrets transfered from CAS to computation components running inside enclaves, we need to protect the network communication between CAS and the components to make sure that an attacker cannot observe the network traffic to steal the secrets. To achieve this security requirement, we enhance SGX-PySpark by designing the network shield that wraps the communication between our system components in TLS connections and ensures that all data passes to the connection and is TLS-encrypted. The certificates for TLS connections are saved in a configuration file protected by our file system shield.

4 DEMONSTRATIONS

In this section, we demonstrate how a user can securely perform data analytics using SGX-PySpark[1]. For demonstration purposes, we consider a simple and classical workload of "wordcount". Assume that the user wants to perform data analytics (e.g. wordcount) over a sensitive input data.

4.1 Protecting Data

A straightforward way to securely process the input data is that the user encrypts the data before uploading it to an untrusted domain such as a public cloud. In this demo, we show that this mechanism is not enough to protect secrets inside the input data, since the user needs to decrypt the encrypted data and then process it in memory; an attacker, especially attackers with privileges, can just dump the memory content to steal the secrets. In SGX-PySpark, the input data is encrypted using the file system shield, and then decrypted and processed inside SGX enclaves which cannot be accessed even by strong attackers with root access.

4.2 Protecting Computation

When the user submit a job (Python code) to the system, an attacker might learn what kind of computation the user wants to perform. We show in this demo that by using SGX-PySpark, the user can further encrypt his computation using the same file system shield before uploading to the untrusted domain (e.g., a cloud) for execution. The secret key for decryption is transferred into executors running inside SGX enclaves using the same mechanism as in §4.1.

4.3 Benchmarks

Beside the wordcount workload, we also make use of a standard data analytics benchmark (i.e.,TPC-H [5]) to demonstrate that SGX-PySpark supports a wide range of queries as native PySpark. Figure 3 presents the latency comparison between SGX-PySpark with native PySpark in processing TPC-H queries. The performance overhead incurred when running Python processes inside enclaves is not significant compared to the native PySpark. The reason for this is that the main overhead of SGX-PySpark is introduced by communication between Python processes and JVMs in workers.

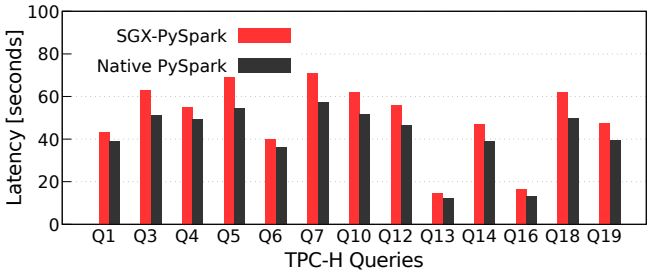

Figure 3: TPC-H benchmark.

Acknowledgments. This work is supported by the European Unions Horizon 2020 research and innovation programme under

[1]The demo repository is available here: https://github.com/doflink/sgx-pyspark-demo

grant agreements No. 777154 (ATOMSPHERE) and No. 780681 (LEGaTO).

REFERENCES

[1] Apache Spark. https://spark.apache.org. Accessed: Jan, 2019.

[2] Py4J. http://py4j.sourceforge.net. Accessed: Jan, 2019.

[3] PySpark. http://spark.apache.org/docs/2.2.0/api/python/pyspark.html. Accessed: Jan, 2019.

[4] Scontain Technology. https://sconedocs.github.io/. Accessed: Jan, 2019.

[5] TPC-H Benchmark. http://www.tpc.org/tpch/. Accessed: Jan, 2019.

[6] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the International Conference on Management of Data (SIGMOD)*.

[7] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[8] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086.

[9] General Data Protection Regulation. 2016. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46. *Official Journal of the European Union (OJ)* (2016).

[10] Tim Greene. Biggest data breaches of 2015. https://www.networkworld.com/article/3011103/security/biggest-data-breaches-of-2015.html. Accessed: Jan, 2019.

[11] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. 2015. *Learning Spark: Lightning-Fast Big Data Analysis.* " O'Reilly Media, Inc.".

[12] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. 2018. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *2018 USENIX Annual Technical Conference (USENIX ATC)*.

[13] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *Proceedings of the Symposium on Security and Privacy (SP)*.

[14] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. 2018. ShieldBox: Secure Middleboxes using Shielded Execution. In *Proceedings of the Symposium on SDN Research (SOSR)*.

[15] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*.

[16] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI)*.