Using Trusted Execution Environments for Secure Stream Processing of Medical Data
(Case Study Paper)

Carlos Segarra[1(✉)], Ricard Delgado-Gonzalo[1], Mathieu Lemay[1], Pierre-Louis Aublin[2], Peter Pietzuch[2], and Valerio Schiavoni[3]

[1] CSEM, Neuchâtel, Switzerland
{cse,rdg,mly}@csem.ch
[2] Imperial College London, London, UK
{p.aublin,prp}@imperial.ac.uk
[3] University of Neuchâtel, Neuchâtel, Switzerland
valerio.schiavoni@unine.ch

Abstract. Processing sensitive data, such as those produced by body sensors, on third-party untrusted clouds is particularly challenging without compromising the privacy of the users generating it. Typically, these sensors generate large quantities of continuous data in a streaming fashion. Such vast amount of data must be processed efficiently and securely, even under strong adversarial models. The recent introduction in the mass-market of consumer-grade processors with Trusted Execution Environments (TEEs), such as Intel SGX, paves the way to implement solutions that overcome less flexible approaches, such as those atop homomorphic encryption. We present a secure streaming processing system built on top of Intel SGX to showcase the viability of this approach with a system specifically fitted for medical data. We design and fully implement a prototype system that we evaluate with several realistic datasets. Our experimental results show that the proposed system achieves modest overhead compared to vanilla Spark while offering additional protection guarantees under powerful attackers and threat models.

Keywords: Spark · Data streaming · Intel SGX · Medical data · Case-study

1 Introduction

Internet of Things (IoT) devices are more and more pervasive in our lifes [22]. The number of devices owned per user is anticipated to increase by 26× by 2020 [19]. These devices continuously generate all large variety of continuous data. Notable examples include location-based sensors (*e.g.*, GPS), inertial units (*e.g.*, accelerometers, gyroscopes), weather stations, and, the focus of this paper, human-health data (*e.g.*, blood pressure, heart rate, stress).

Published by Springer Nature Switzerland AG 2019
J. Pereira and L. Ricci (Eds.): DAIS 2019, LNCS 11534, pp. 91–107, 2019.
https://doi.org/10.1007/978-3-030-22496-7_6

These devices usually have very restricted computing power and are typically very limited in terms of storage capacity. Hence, this continuous processing of data must be off-loaded elsewhere, in particular for storage and processing purposes. In doing so, one needs to take into account potential privacy and security threats that stem inherently from the nature of the data being generated and processed.

Cloud environments represent the ideal environment to offload such processing. They allow deployers to hand-off the maintenance of the required infrastructure, with immediate benefit for instance in terms of scale-out with the workload.

Processing privacy-sensitive data on untrusted cloud platforms present a number of challenges. A malicious (compromised) Cloud operator could observe and leak data, if no countermeasures are taken beforehand. While there are software solutions that allow to operate on encrypted data (*e.g.*, partial [33] or full-homomorphic [24] encryption), their current computational overhead makes impractical in real-life scenarios [25].

The recent introduction into the mass market of processors with embedded trusted execution environments (TEEs), *e.g.*, Intel Software Guard Extensions (SGX) [20] (starting from processors with codename Skylake) or ARM TrustZone [1], offer a viable alternative to pure-software solutions. TEEs protect code and data against several types of attacks, including a malicious underlying OS, software bugs or threats from co- hosted applications. The application's security boundary becomes the CPU itself. The code is executed at near-native execution speeds inside enclaves of limited memory capacity. All the major Infrastructure-as-a-Service providers (Google [21], Amazon [2], IBM [3], Microsoft [37]) are nowadays offering nodes with SGX processors.

We focus on the specific use case of processing data streams generated by health-monitoring wearable devices on untrusted clouds with available SGX nodes. This setting addresses the fact that algorithms for analyzing cardiovascular signals are getting more complex and computation-intensive. Thus, traditional signal-processing approaches [29] have left the way to deep neural networks [43,45]. This increase in computational expenditure has moved the processing towards centralized centers (*i.e.*, the cloud) when scaling up to a large fleet of wearable devices is needed. In order to illustrate the concept, we present a system that computes in real time several metrics of the heart-rate variability (HRV) steaming from wearable sensors. While existing stream processing solutions exist [27,46], they either lack support for SGX or, if they do support it, are tied to very specific programming frameworks and prevent adoption in industrial settings.

The contributions of this case-study paper are twofold. First, we design and implement a complete system that can process heart-specific signals inside SGX enclaves in untrusted clouds. Our design leverages SGX-Spark, a stream processing system that exploits SGX to execute stream analytics inside TEEs (described in detail in Sect. 2). Note that our design is flexible enough to be used with different stream processing systems (as further described later).

Second, we compare the proposed system against the vanilla, non-secure Spark. Our evaluation shows that the current overhead of SGX is reasonable even for large datasets and for high-throughput workloads and that the technology is almost ready for production environments.

This paper is organized as follows. In Sect. 2, we introduce Intel SGX, Spark, and SGX-Spark. The architecture of the proposed system is presented in Sect. 3, while we further provide implementation details in Sect. 4. We evaluate our prototype with realistic workloads in Sect. 5 for which include experimental comparisons also against the vanilla Spark. A summary of related work in the domain of (secure) stream processing is given in Sect. 6. Finally, we present future work (Sect. 7) before concluding in Sect. 8.

2 Background

To better understand the design and implementation details, we introduce some technical background on the underlying technologies that we leverage, as well as some of the specific features interesting for cardiac signals. In Sect. 2.1, we provide background on the technical aspects exploited in the remaining of this paper, specifically describing the operating principles of Intel SGX, Spark and its secure counter-part SGX-Spark. In Sect. 2.2, we describe the specifics of the data streams that the system has to deal with from the medical domain, such as heart-beat monitoring signals, together with the required processing that our system allows to offload on an untrusted cloud provider.

2.1 Technical Background

Trusted Execution Environments and Intel SGX. A trusted execution environment (TEE) is an isolated area of the processor that offers code and data's confidentiality and integrity guarantees. TEEs are nowadays available in commodity CPUs, such as Arm TrustZone and Intel®SGX.

In comparison with Arm TrustZone, SGX includes a remote attestation protocol, support multiple trusted applications on the same CPU, and its SDK is easier to program with. As mentioned earlier, all the major IaaS providers offer SGX-enabled instances on their cloud offering, hence we decided to base the design of our system on top of it. Briefly, the SGX extensions are a set of instructions and memory access extensions. These instructions enable applications to create hardware-protected areas in their

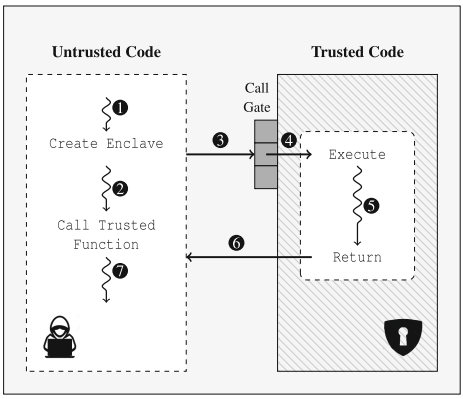

Fig. 1. Intel SGX execution workflow.

address space, also known as, *enclaves* [31]. At initialization time, the content loaded is measured (via hashing) and sealed. An application using an enclave identifies itself through a remote attestation protocol and, once verified, interacts with the protected region through a call gate mechanism. In particular, Fig. 1 breaks down the typical execution workflow of SGX applications. After the initial attestation protocol, code in the untrusted region creates an enclave and securely loads trusted code and data inside (Fig.-❶). Whenever this untrusted code wants to make use of the enclave, it makes a call to a trusted function (Figs.-❷, -❸) that gets captured by the call gate mechanism and, after performing sanity and integrity checks (Fig.-❹), gets executed (Fig.-❺), the value returned (Fig.-❻) and the untrusted code can resume execution (Fig.-❼). The security perimeter is kept at the CPU package and, as a consequence, all other software including privileged software, OS, hypervisors or even other enclaves are prevented from accessing code and data located inside the enclave. Most notably, the systems' main memory is left untrusted and the traffic between CPU and DRAM over the protected address range is managed by the *Memory Encryption Engine* [26].

Spark and Spark Streaming. Spark is a cluster-computing framework to develop scalable, fault-tolerant, distributed applications. It builds on RDDs, resilient distributed datasets [46], a read-only collection distributed over a cluster that can be rebuilt if one partition is lost. It is implemented in Scala and provides bindings for Python, Java, SQL and R. Spark Streaming [47] is an extension of Spark's core API that enables scalable, high-throughput, fault tolerant stream (mini-batch) processing of data streams [16]. The proposed system leverages Spark Streaming to perform file-based streaming, by monitoring a filesystem interface outside the enclave.

SGX-LKL and SGX-Spark. SGX-LKL [11] is a library OS to run unmodified Linux binaries inside enclaves. Namely, it provides system support for managed runtimes, *e.g.*, a full JVM. This feature enables the deployment of Spark, and Spark Streaming applications to leverage critical computing inside Intel SGX with minimal to no modifications to the application's code. SGX-Spark [15] builds on SGX-LKL. It partitions the code of Spark applications to execute the sensitive parts inside SGX enclaves. Figure 2 depicts its architecture. Basically, it deploys two collaborative Java Virtual Machines (JVM), one outside (Fig. 2, *Spark Driver*) and one inside the enclave (Fig. 2, *Driver Enclave*) for the driver, and two more for each worker deployed. Spark code outside the enclave

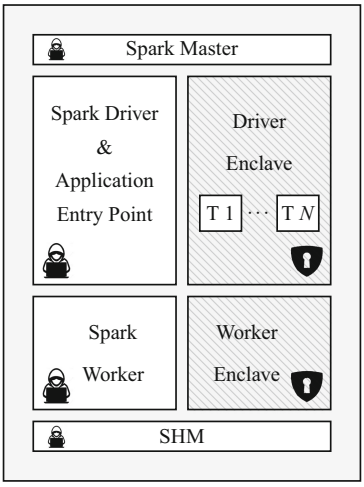

Fig. 2. SGX-Spark attacker model and collaborative structure scheme.

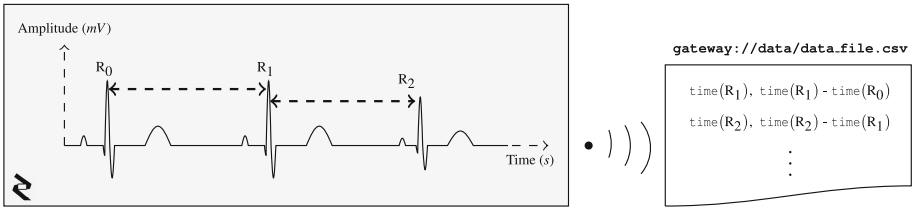

Fig. 3. Schematic representation of an ECG signal. It shows three normal beats and the information transferred from the sensor to the gateway. The most relevant part of the ECG wave are the R peaks and the time elapsed between them. The RR intervals together with the R peaks' timestamp are sent from the sensor to the gateway.

accesses only encrypted data. The communication between the JVMs is kept encrypted and is performed through the host OS shared memory. SGX-Spark provides a compilation toolchain, and it currently supports the vast majority of the native Spark operators, allowing to transparently deploy and run existing Spark applications into the SGX enclaves.

2.2 Heart Rate Variability Analysis

The data streams used for the evaluation and the algorithms compiled with SGX-Spark belong to the medical domain and motivate the real need for confidentiality and integrity. As further explained in Sect. 3, our use case contemplates a scenario where multiple sensors track the cardiac activity of different users. The two most standard procedures for monitoring heart activity are electrocardiograms (ECG) and photoplethysmograms (PPG). An ECG measures the heart's electrical activity and is the method used by, for instance, chest bands. A PPG is an optical measure of the evolution of blood volume over time and is the method used by wrist-based sensors [34]. Both procedures lead to an approximation of R peaks' timestamps and the intervals between them (RR intervals). The generation of the approximated diagram and the time measures are done inside the sensor. Figure 3 depicts a schematic representation of an ECG and the values streamed from the sensor to the gateway: R peak's timestamps and RR intervals. With healthy individuals' heart rate (HR) averaging between 60 to 180 beats per minute (bpm), the average throughput per client is between 23 and 69 bytes per second. An interesting use case of RR processing, besides HR approximation, is the study of Heart Rate Variability (HRV). HRV [30] is the variation in the time intervals between heartbeats and it has been proven to be a predictor of myocardial infarction. Finally, despite the proposed system being specifically designed for streams with these data features, its modular design (as we later describe in Sect. 3) makes it easy to adapt to other use-cases.

3 Architecture

The architecture of the proposed system is depicted in Fig. 4. It is composed of a server-side component which executes on untrusted machines (*e.g.*, nodes on the cloud), where Intel SGX is available. The clients are distributed among remote locations. Each client is a sensor generating samples, and a gateway aggregating and sending them periodically every n seconds to the cloud-based component. Similarly, clients fetch the results at fixed time intervals (*i.e.*, every 5 s in our deployments). The interaction between the clients and the server-side components of the system happens over a filesystem interface. Each client data stream is processed in parallel by the SGX-Spark job. In the reminder, we further detail these components.

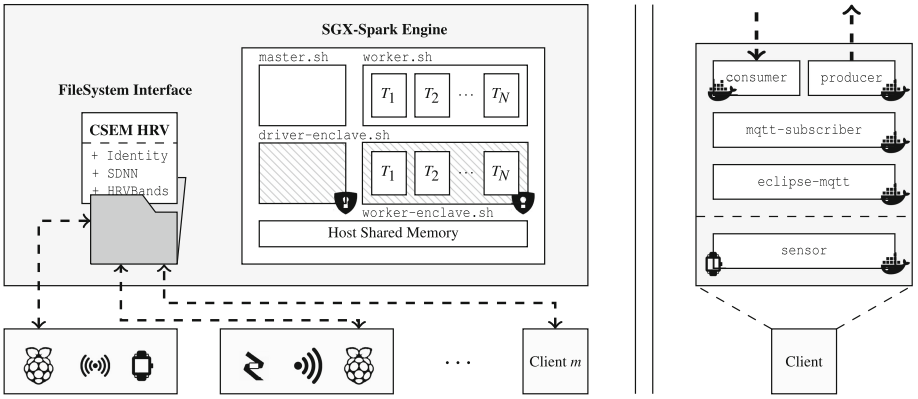

Fig. 4. (Left) Schematic of the system's main architecture. A set of clients bidirectionally stream data to a remote server. The interaction is done via a filesystem interface. On the server side, SGX-Spark performs secure processing using different HRV analysis algorithms. (Right) Breakdown of a packaged client: it includes a `sensor` and gateway that wrap four different microservices (MQTT broker, `mqtt-subscriber`, `consumer`, `producer`) to interact with the remote end.

3.1 Server-Side

The server-side component is made by three different modules: a filesystem interface, the SGX-Spark engine, and a set of algorithms to analyze HRV. The filesystem interface acts as a landing point for the batches of data generated by each client. It is monitored by the SGX-Spark engine. Currently, it is mounted and unmounted, respectively at start-up time and upon the shutdown of the service. The streaming engine and the pool of algorithms are compiled together by the same toolchain, yet they are independent. The Spark engine (deployed in standalone mode) executes: the master process, the driver process, and an arbitrary number of workers. In the case of SGX-Spark jobs, two JVMs are

deployed per driver and worker process: one inside an enclave and one outside. The communication between JVMs is kept encrypted and is done through the host OS shared memory (see Fig. 2). For each JVM pair, SGX-Spark will initialize a new enclave. The specific algorithm that the system will execute is currently set at start-up time, although several concurrent ones can be executed, each yielding separated results.

3.2 Clients

The client is a combination of: (1) a data generator that simulates a sensor and (2) a gateway that interacts with the remote end. The data generator streams RR intervals. These samples are gathered by the gateway, which stacks and sends them for processing in a file-based streaming fashion. The typical size of these batches is in the 230–690 Bytes range. Each gateway is composed by: a message broker that handles the samples, a service that handles data pre-processing and batch sending, and a fetcher that directly *greps* from the server's filesystem.

3.3 Threat Model

We assume that the communication between the gateway and the filesystem is kept protected (*e.g.*, encrypted) using secure transfer protocols (more in Sect. 4). Given this assumption, the threat model is the same as typical systems that rely on SGX. Specifically, we assume the system software is untrusted. Our security perimeter only includes the internals of the CPU package. The trusted computing base is Intel's microcode as well as and the code loaded at the enclave, which can be measured and integrity can be checked. We assume that in our case the client package is trusted and tamper-proof. We focus on protecting the areas *outside* user's control. However, if the client package is deployed in, for instance, a Raspberry Pi, the Trusted Computing Base (TCB) could be further reduced using Arm TrustZone and Op-Tee [9].

3.4 Known Vulnerabilities

As for the known vulnerabilities, SGX (in particular the memory encryption engine) is not designed to be an oblivious RAM. As a consequence and adversary can perform traffic analysis attacks [26]. Moreover, side-channel attacks [38] and speculative execution attacks (*Spectre*-like [13] and *Foreshadow* [42]) have still successful against enclaves and will require in-silicon fixes.

4 Implementation

This section presents the further implementation details. To stress-test our evaluation, we replaced real sensors with synthetic data generators. Additionally, we deploy a large number of Docker containers [5] to mimic a fleet of concurrent clients.

4.1 Server-Side

We rely on the original SGX-Spark implementation, and we only modify it to support a different in-enclave code deployment path, so that the `.jar` archive is available inside the enclaves and the shared memory. The application code is implemented in the Scala programming language [14]. Applications must adhere to the RDD API [10] to be usable inside the SGX enclaves. We use SGX-Spark via Structured Streaming jobs, and must also adhere to the same API. We have implemented two state-of-the-art HRV analysis algorithms, namely `SDNN` and `HRVBands` [39]. The SDNN algorithm measures the standard deviation of NN (RR in our case) intervals. HRVBands performs frequency domain calculations: high-frequency (HF) power, low-frequency (LF) power and HF to LF ratio. For the sake of performance comparison, we also include results using an identity algorithm, simply reading the input data stream and outputting it. The implementation of these algorithms rely on basic Spark Streaming operators, and their corresponding Scala implementations. We use the file-based data stream input for Spark streaming.[1]

4.2 Clients

Clients correspond to body-sensors strapped to the body of a user. These are connected to a gateway, (*e.g.*, a Raspberry Pi) packaged together. Our implementation decouples the clients into into five different microservices (see Fig. 4, right). For evaluation purposes, the `sensor` is a Python service that generates random RR intervals. These are published into the MQTT queue [6,8] following a uniform time distribution. The gateway is composed by a MQTT queue and broker service. We rely on `eclipse-mosquitto`[2], a `mqtt-sub` service that subscribes to the specific topic and generates data files with several samples, and a `producer` and `consumer` services that interact with the remote filesystem. These components are implemented in Python, and consist of 888 Lines of Code (LoC). Our prototype relies on Docker to facilitate the deployment of new clients, and on `docker-compose` [4] to easily group orchestrate their deployment. The communication between the client and the server happens via SSH/SecureFTP to ensure transport layer security when transferring user's data.

4.3 Deployment

To ease scalability and reproducibility of both server and client, deployment is orchestrated by a single script detached from both execution environments. Specifying the remote location, the SGX-Spark engine, the streaming algorithm and the filesystem interface are initialized either container-based or on metal. Specifying the number of simulated users and their location, a cluster of clients is dynamically started. On execution time, a Spark streaming service located

[1] https://spark.apache.org/docs/2.2.0/api/java/org/apache/spark/streaming.
[2] https://hub.docker.com/_/eclipse-mosquitto/.

in a remote server with a master process and an arbitrary number of Spark workers (or executors) interacts with a standalone Docker Swarm composed by the cluster of clients, a name discovery service and an overlay network. This architecture scales to hundreds of clients.

5 Evaluation

In this section, we present the experimental evaluation. We first present the evaluation settings for both the client and the server components. Then, we describe the metrics of interest on which we focus our experiments. Finally, we present our results. Our experiments answer the following questions: *(i)* is the design of the proposed system sound? *(ii)* is our implementation efficient, *(iii)* what is the overhead of SGX, and *(iv)* is it scalable?

5.1 Settings

Clients. Each client (*e.g.*, a body sensor in real-life) is emulated by a standalone Docker application. We deploy them on a quad-16core (64 hardware cores) AMD EPYC 7281 with 64 GiB of RAM running Ubuntu v18.04 LTS (kernel 4.15.0-42-generic). The client containers are built and deployed using Docker (v18.09.0) and `docker-compose` (v1.23.2). We use `docker-machine` (v0.16.0) with the `virtualbox` disk image. Each machine hosts 20 clients, the maximum number of services supported by its local network, and it registers itself to the Swarm via a name discovery service running on another machine. Inter-container communication rely on the `overlay` network driver. We pull the latest images available on Docker Hub for the Consul name discovery service (v1.4) and the `eclipse-mosquitto` (v1.5) message broker.

Server. The server components run on host machines with Intel ® Xeon ® CPU E3-1270 v6 @ 3.80 GHz with 8 cores and 64 GiB RAM. We use Ubuntu 16.04 LTS (kernel 4.19.0-41900-generic) and the official Intel ® SGX driver v2.0 [7], and SGX-LKL [11]. We use an internal release of the SGX-Spark framework.

5.2 Experiment Configurations

We compare the results of 3 different systems (or execution modes): the vanilla Spark (our baseline), the SGX-Spark system with enclaves disabled (*i.e.* collaborative JVMs communicating over SHM which run outside the SGX enclaves) and SGX-Spark with enclaves enabled. The latter mode is the one the proposed system runs in. The current implementation of SGX-Spark (still under development) does not provide support for Spark's `Streaming Context` inside enclaves. To overcome this temporary limitation, we evaluate the `SDNN` and `Identity` algorithms in batch and stream mode. For the former, all three different execution

modes are supported. For the latter, we present estimated results for SGX-Spark with enclaves enabled, basing the computation time on the batch execution times and the additional overhead against the other modes. The algorithms are fed with a data file or a data stream, respectively. In the streaming scenario, an output file is generated every ten seconds. In a multi-client scenario, each client has a separated data stream (or file) and consequently a different result file. A streaming execution consists of 5 min of the service operating with a specific configuration. We execute our experiments 5 times and report average and standard deviations.

Metrics. To assess performance, scalability, and efficiency, we consider average batch processing times for streaming jobs, and elapsed times for batch executions. Note that we mention *batch* in two different contexts: batch execution (one static input and static output) and streaming batches. Spark Streaming divides live input data in chunks called *batches* determined by a time duration (10 s in our experiments). The time it takes the engine to process the data therein contained is denoted as batch processing time. In order to obtain all batch processing times, we rely on the internal Spark's REST API [12]. Since the GET request fetches the historic of batch processing times for the running job, one single query right before finishing the execution provides all the sufficient informations for our computations. In order to obtain the elapsed times for batch executions, a simple logging mechanism suffices.

Workload. The clients inject streams as cardiac signals, as shown earlier (Sect. 2.2). Each signal injects a modest workload into our system (230–690 bytes per minute). Hence, to assess the efficiency and the processing time as well as to uncover possible bottlenecks, we scale up the output rate of these signals with the goal of inducing more aggressive workloads. We do so in detriment of medical realism, since arbitrary input workloads do not relate to any medical situation or condition. Table 1 shows the variations used to evaluate the various execution modes.

Table 1. Different input loads used for Batch Executions (BE) and Streaming Executions (SE). We present the sample rate they simulate (*i.e.* how many RR intervals are streamed per second) and the overall file or stream size (Input Load).

Experiment	**`s_rate` (samples/s)**	**Input Load**
BE - Small Load	$\{44, 89, 178, 356, 712, 1424\}$	$\{1, 2, 4, 8, 16, 32\}$ kB
SE - Small Load	$\{44, 89, 178, 356, 712, 1424\}$	$\{1, 2, 4, 8, 16, 32\}$ kB/s
BE - Big Load	$\{44, 89, 178, 356, 712, 1424\} * 1024$	$\{1, 2, 4, 8, 16, 32\}$ MB
SE - Big Load	$\{44, 89, 178, 356, 712, 1424\} * 1024$	$\{1, 2, 4, 8, 16, 32\}$ MB/s

5.3 Results

Batch Execution: Input File Size. The configuration for the following experiments is: one client, one master, one driver, one worker, and a variable input file that progressively increases in size. We measure the processing (or elapsed) time of each execution and present the average and standard deviation of experiments with the same configuration. The results obtained are included in Fig. 5.

From the bar plot we highlight that the variance between execution times among same execution modes as we increase the input file size is relatively low. However, it exponentiates as we reach input files of 4–8 MB. We also observe that the slow-down factor between execution modes remains also quite static until reaching the before mentioned load threshold. SGX-Spark with enclaves, if input files are smaller than 4 MB, increases execution times x4-5 when compared to vanilla Spark and x1.5-2 when compared to SGX-Spark with enclaves disabled. Note that, since a single client in our real use case streams around 230 to 690 bytes per minute, the current input size limitation already enables several concurrent clients.

Streaming Execution: input load. As done previously, we scale the load of the data streams that feed the system. We deploy one worker, one driver and one client, query the average batch processing time to Spark's REST API, and present the results for the `Identity` and `SDNN` algorithms. Results are summarized in Fig. 6.

We obtain results for vanilla Spark, and SGX-Spark without enclaves, and we estimate them for SGX-Spark with enclaves. We observe similar behavior as those in Fig. 5. Variability among same execution modes when increasing the input stream size is low until reaching values of around 4 to 8 MB per second. Similarly, the slow-down factor from vanilla Spark to SGX-Spark without enclaves remains steady at around x2-2.5 until reaching the load threshold. As a consequence, it is reasonable to estimate that the behavior of SGX-Spark with enclaves will preserve a similar slow-down factor ($\times$4-$\times$5) when compared with vanilla Spark in streaming jobs. Similarly, the execution time will increase linearly with the input load after crossing the load threshold of 4 MB. Note as well how different average batch processing times are in comparison with elapsed times, in spite of relatively behaving similar. The average of streaming batch processing times smoothens the initial overhead of starting the Spark engine, and data loading times are hidden under previous batches' execution times.

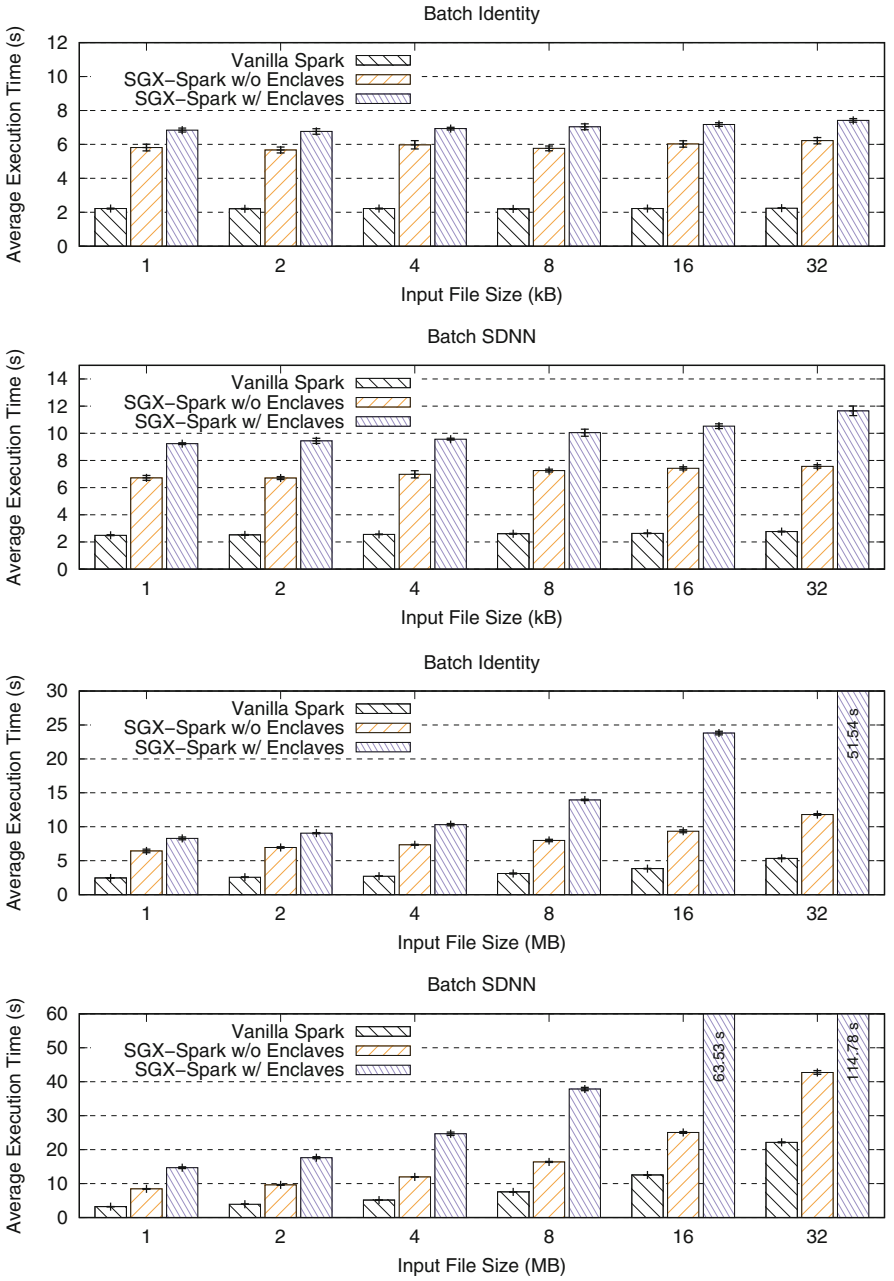

Fig. 5. Evolution of the average elapsed time, together with its standard deviation, as we increase the size of the input file. We compare the three different execution modes for each algorithm. Mode SGX-Spark w/ enclaves is the mode our system runs in.

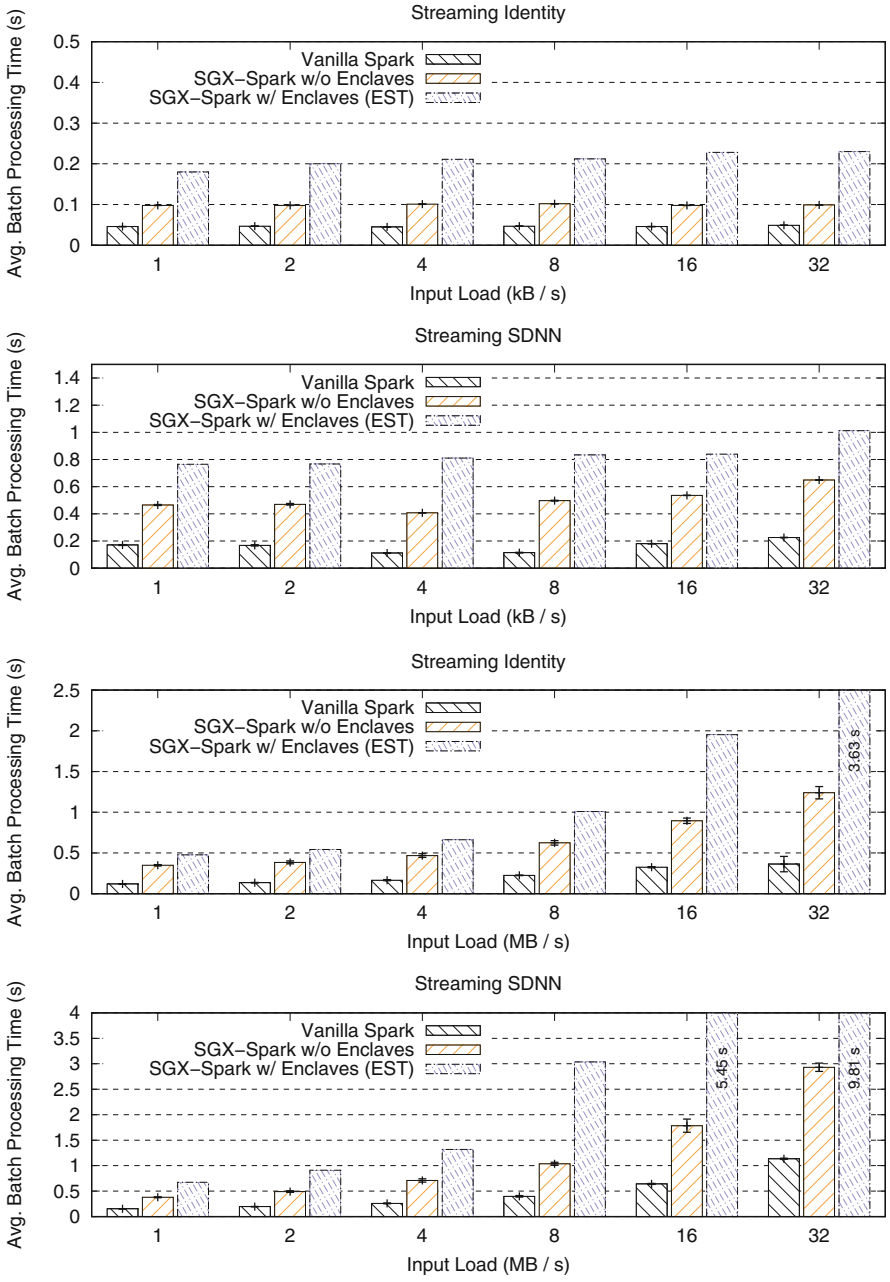

Fig. 6. Evolution of the average batch processing time as we increase the input file size. We compare the results of the three different execution modes. Note that those corresponding to SGX-Spark w/ enclaves are estimated basing on the results in Fig. 5 and the slow-down with respect to the other execution modes.

6 Related Work

Stream processing has recently attracted a lot of attention from academia and industry [28,32,44]. Apache Spark [46] is arguably the de-facto standard in this domain, by combining batch and stream processing with a unified API [48]. Apache Spark SQL [18] allows to process structured data by integrating relational processing with Spark's functional programming style. Structured streaming [17] leverages Spark SQL and it compares favorably against the discretized counterpart [47]. However, the former lacks security or privacy guarantees, and hence it was not considered. The proposed system relies on SGX-Spark, as it directly extends Spark with SGX support.

Opaque [49] is a privacy-preserving distributed analytics system. It leverages Spark SQL and Intel SGX enclaves to perform computations over encrypted Spark DataFrames. In `encryption` mode, Opaque offers security guarantees similar to the proposed system. However, *(1)* the Spark master must be co-hosted with the client, a scenario not supported by our multi-client setting and *(2)* it requires changes to the application code. In `oblivious` mode, *i.e.*, protecting against traffic pattern analysis attacks, it can be up to 46× slower, a factor not tolerable for the real-time analytics in our setting. SecureStreams [27] is a reactive framework that exploits Intel SGX to define dataflow processing by pipelining several independent components. Applications must be written in the Lua programming language, hindering its applicability to legacy systems or third-party programs. DPBSV [35] is a secure big data stream processing framework that focuses on securing data transmission from the sensors or clients to the *Data Stream Manager (DSM)* or server. Its security model requires a PKI infrastructure and a dynamic prime number generation technique to synchronously update the keys. In spite of using trusted hardware on the DSM end for key generation and management, the server-side processes all the data in clear, making the framework not suitable for our security model.

Homomorphic encryption [23] does not rely on trusted execution environments and offers the promise of providing privacy-preserving computations over encrypted data. While several works analyzed the feasibility of homomorphic encryption schemes in cloud environments [40,41], the performance of homomorphic operations [25] is far from being pragmatic.

Further, for the specific problem of HRV analysis, while periodic monitoring solutions exist [36], they are focused on embedded systems. As such, since they off-load computation to third-party cloud services, these solutions simply overlook the privacy concerns that the proposed system considers.

To the best of our knowledge, there are no privacy-preserving real-time streaming systems specifically designed for medical and cardiac data. The proposed system fills this gap by leveraging Intel SGX enclaves to compute such analytics over public untrusted clouds without changing the existing Java- or Scala-based source code.

7 Future Work

The current prototype can be improved along several dimensions. First, we envision to support clients running inside ARM TrustZone: this TEE is widely available in low-power devices (*e.g.*, Raspberry PI), hence makes an ideal candidate to reduce the TCB in the client-side of the architecture. Second, we intend to improve the plug-in mechanism for additional analysis of the data, as currently a given algorithm is set at deploy-time, while it is expected to load/unload those at runtime. Thirdly, we intend to study the cost of deployment of such system over public cloud infrastructures such as AWS Confidential Computing.

8 Conclusion

We presented a stream-processing architecture and implementation that leverage Spark-SGX to overcome privacy concerns of deploying such systems over untrusted public clouds. Its design allows to easily scale to different types of data generators (*e.g.*, the clients). The processing components that execute on the cloud rely on SGX-Spark, a stream processing framework that can executes Spark jobs within SGX enclaves. Our evaluation shows that for typical signal processing, despite an observed overhead of 4×–5× induced by the current experimental version of SGX-Spark, the performance is still practical. This suggests that it will be possible in a near-future to deploy such systems on a production-ready environment with performances that can easily satisfy even strict Service Level Agreements, while keeping maintaining the costs to use the cloud infrastructure reasonable. We intend to release the code as open-source.

Acknowledgements. We are grateful to the members of the LSDS Team (https://lsds.doc.ic.ac.uk/) at Imperial College London to have provided us early access to SGX-Spark.

References

1. ARM TrustZone Developer. https://developer.arm.com/technologies/trustzone
2. Coming Soon: Amazon EC2 C5 Instances, the next generation of Compute Optimized instances. http://amzn.to/2nmIiH9
3. Data-in-use protection on IBM Cloud using Intel SGX. https://www.ibm.com/blogs/bluemix/2018/05/data-use-protection-ibm-cloud-using-intel-sgx/
4. Docker Documentation: Docker Compose. https://docs.docker.com/compose/
5. Docker: What is a Container? https://www.docker.com/resources/what-container
6. Eclipse Paho MQTT Implementation. https://www.eclipse.org/paho/
7. Intel Software Guard Extension for Linux OS Driver on GitHub. https://github.com/intel/linux-sgx-driver
8. MQTT Communication Protocol. http://mqtt.org/
9. Open Portable Trusted Execution Environment. https://www.op-tee.org
10. RDD Programming Guide. https://spark.apache.org/docs/latest/rdd-programming-guide.html

11. SGX-LKL on Github. https://github.com/lsds/sgx-lkl
12. Spark Documentation: REST API. https://spark.apache.org/docs/latest/monitoring.html#rest-api
13. Spectre Attack SGX on Github. https://github.com/lsds/spectre-attack-sgx
14. The Scala Programming Language. https://www.scala-lang.org/
15. D3.2 SecureCloud: Specification and Implementation of Reusable Secure Microservices (2017). https://www.securecloudproject.eu/wp-content/uploads/D3.2.pdf
16. Apache Foundation: Spark streaming programming guide. https://spark.apache.org/docs/2.2.0/streaming-programming-guide.html
17. Armbrust, M., et al.: Structured streaming: a declarative API for real-time applications in Apache Spark. In: ACM SIGMOD 2018 (2018)
18. Armbrust, M., et al.: Spark SQL: relational data processing in Spark. In: ACM SIGMOD 2015 (2015)
19. Barbosa, M., et al.: SAFETHINGS: data security by design in the IoT. In: IEEE EDCC 2017 (2017)
20. Costan, V., Devadas, S.: Intel SGX explained. IACR 2016 (2016)
21. Darrow, B.: Google is first in line to get Intel's next-gen server chip. http://for.tn/2lLdUtD
22. Gartner: Leading the IoT Gartner Insights on how to lead in a connected world (2017)
23. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: ACM STOC 2009 (2009)
24. Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the AES circuit. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 850–867. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32009-5_49
25. Göttel, C., et al.: Security, performance and energy trade-offs of hardware-assisted memory protection mechanisms. In: IEEE SRDS 2018 (2018)
26. Gueron, S.: A memory encryption engine suitable for general purpose processors. IACR 2016
27. Havet, A., et al.: SecureStreams: a reactive middleware framework for secure data stream processing. In: ACM DES 2017 (2017)
28. Koliousis, A., et al.: SABER: window-based hybrid stream processing for heterogeneous architectures. In: ACM SIGMOD 2016 (2016)
29. Kumar, A., Shaik, F., Rahim, B.A., Kumar, D.S.: Signal and Image Processing in Medical Applications. Springer, Heidelberg (2016). https://doi.org/10.1007/978-981-10-0690-6
30. Malik, M.: Heart rate variability: standards of measurement, physiological interpretation, and clinical use. Circulation **93**, 1043–1065 (1996)
31. McKeen, F., et al.: Innovative instructions and software model for isolated execution. In: HASP 2013 (2013)
32. Miao, H., Park, H., Jeon, M., Pekhimenko, G., McKinley, K.S., Lin, F.X.: StreamBox: modern stream processing on a multicore machine. In: USENIX ATC 2017 (2017)
33. 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
34. Parák, J., Tarniceriu, A., Renevey, P., Bertschi, M., Delgado-Gonzalo, R., Korhonen, I.: Evaluation of the beat-to-beat detection accuracy of PulseOn wearable optical heart rate monitor. In: IEEE EMBC 2015 (2015)
35. Puthal, D., Nepal, S., Ranjan, R., Chen, J.: DPBSV - an efficient and secure scheme for big sensing data stream. In: IEEE TRUSTCOM 2015 (2015)

36. Renevey, P., et al.: Respiratory and cardiac monitoring at night using a wrist wearable optical system. In: IEEE EMBC 2018 (2018)
37. Russinovich, M.: Introducing Azure Confidential Computing. https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/
38. Schwarz, M., Weiser, S., Gruss, D., Maurice, C., Mangard, S.: Malware guard extension: using SGX to conceal cache attacks. In: Polychronakis, M., Meier, M. (eds.) DIMVA 2017. LNCS, vol. 10327, pp. 3–24. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-60876-1_1
39. Shaffer, F., Ginsberg, J.P.: An overview of heart rate variability metrics and norms. Front. Pub. Health **5**, 258 (2017). https://doi.org/10.3389/fpubh.2017.00258
40. Stephen, J.J., Savvides, S., Sundaram, V., Ardekani, M.A., Eugster, P.: STYX: stream processing with trustworthy cloud-based execution. In: ACM SoCC 2016 (2016)
41. Tetali, S.D., Lesani, M., Majumdar, R., Millstein, T.: MrCrypt: static analysis for secure cloud computations. In: ACM OOPSLA 2013 (2013)
42. Van Bulck, J., et al.: Foreshadow: extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In: USENIX Security 2018 (2018)
43. Van Zaen, J., Chételat, O., Lemay, M., Calvo, E.M., Delgado-Gonzalo, R.: Classification of cardiac arrhythmias from single lead ECG with a convolutional recurrent neural network. In: BIOSTEC 2019 (2019)
44. Venkataraman, S., et al.: Drizzle: fast and adaptable stream processing at scale. In: ACM OSP 2017 (2017)
45. Xiong, Z., Nash, M., Cheng, E., Fedorov, V., Stiles, M., Zhao, J.: ECG signal classification for the detection of cardiac arrhythmias using a convolutional recurrent neural network. Physiol. Measur. **39**, 094006 (2018)
46. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: USENIX HotCloud 2010 (2010)
47. Zaharia, M., Das, T., Li, H., Shenker, S., Stoica, I.: Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In: USENIX HotCloud 2012 (2012)
48. Zaharia, M., et al.: Apache spark: a unified engine for big data processing. Commun. ACM 2016 **59**, 56–65 (2016)
49. Zheng, W., Dave, A., Beekman, J.G., Popa, E.A., Gonzalez, J.E., Stoica, I.: Opaque: an oblivious and encrypted distributed analytics platform. In: USENIX NSDI 2017 (2017)