UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat de Matemàtiques i Estadística

Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Joint Bachelor Thesis in Mobile
Interdisciplinary Higher Education Center (CFIS-UPC)

Using Trusted Execution Environments for Secure Stream Processing of Medical Data

Spring Semester - May 2019

Author:
Carlos Segarra González[1,2]
carlos.segarra@csem.ch

Supervisors:
José Adrián Rodríguez Fonollosa[1]
jose.fonollosa@upc.edu
Ricard Delgado-Gonzalo[2]
ricard.delgado@csem.ch

[1] Universitat Politècnica de Catalunya BarcelonaTech, Barcelona, Spain
[2] Swiss Center for Electronics and Microtechnology (CSEM), Neuchâtel, CH

In partial fulfillment of the requirements for the
Bachelor's Degree in Mathematics
and
Bachelor's Degree in Telecommunications Technologies and Services Engineering

csem

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Centre de Formació Interdisciplinària Superior

Muchos años después, frente al pelotón de fusilamiento, el coronel Aureliano Buendía había de recordar aquella tarde remota en que su padre lo llevó a conocer el hielo. Macondo era entonces una aldea de veinte casas de barro y cañabrava construidas a la orilla de un río de aguas diáfanas que se precipitaban por un lecho de piedras pulidas, blancas y enormes como huevos prehistóricos. El mundo era tan reciente, que muchas cosas carecían de nombre, y para mencionarlas había que señalarías con el dedo.

Gabriel García Márquez, *Cien años de soledad*

Note from the Author

The work here presented is my Bachelor's Thesis for my joint degree in Mathematics and Telecommunications Engineering within the Interdisciplinary Higher Education Center (CFIS) from the Polytechnic University of Catalonia (UPC). It has been developed during a six-month internship at the Swiss Center for Electronics and Microtechnology (CSEM) under the supervision of Ricard Delgado-Gonzalo, in the Embedded Software group. Professor José Adrián Rodríguez Fonollosa, from the UPC, has co-advised this project, and has been the official tutor with regard to the university.

Parts of this work have been included in two different conference papers which have been accepted and will be published in their respective proceedings. The first one is entitled ***Secure Stream Processing for Medical Data*** [1] and will be presented at the 41st IEEE Engineering in Medicine and Biology Conference (EMBC '19) to be held in Berlin, Germany, from July 23-27 2019. It focuses on a particular medical application our work could be used in. The second one is entitled ***Using Trusted Execution Environments for Secure Stream Processing of Medical Data*** [2] and will be presented in the 19th International Conference on Distributed Applications and Interoperable Systems (DAIS '19) to be held in Copenhagen, Denmark, from June 17-21 2019. It covers our solution's implementation in-depth and evaluates its performance.

Declaration of Authorship

I hereby declare that, except where specific reference is made to the work of others, this Bachelor's thesis has been composed by me and it is based on my own work. None of the contents of this dissertation have been previously published nor submitted, in whole or in part, to any other examination in this or any other university.

Signed: _____

Date: _____

Acknowledgments

These lines end a five year endeavour, during which I have met people, and learnt things that will stick with me for the years to come. Many have given me a hand along the way, but there are a certain few without which this work would have never seen the light. I would like to take this moment and thank them for their help.

First of all, I would like to thank Ricard Delgado and the Swiss Center for Electronics and Microtechnology for hosting me during the six month internship that enabled me to focus completely on my work. Ricard has been an unsurpassable host and, together with Enric Muntané, they made Neuchâtel feel like a home away from home. Secondly, I would like to thank Valerio Schiavoni from the University of Neuchâtel. It was him who proposed the initial topic, it was him who provided the contact with Peter Pietzuch and Pierre-Louis Aublin, who gladly granted me with developer access to their on-going projects, and it was him who, week in and week out, helped me keep the bigger picture and paved the way for the personal success this work has been. Lastly, I would like to thank Professor José Adrián Rodríguez Fonollosa for always answering my doubts and helping me without hesitation.

On a more personal note, I would like to thank my parents for their unconditional love and support throughout all these years. Without their guidance during decisive moments, I would not be writing these lines today. I would also like to express my most sincere gratitude to José Andrés, only he knows what has taken us to be here today, and I owe a good part of it to him. Finally, I must devote some words to Clara. Even though time might now be scarce, she has taught me more about myself than I would have learnt in a lifetime without her.

Carlos Segarra González
Neuchâtel, May 13, 2019

Abstract

Using Trusted Execution Environments for Secure Stream Processing of Medical Data

by Carlos Segarra González

Processing sensitive data, specially medical data 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 information 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.

This Bachelor Thesis presents a secure streaming processing system built on top of Intel SGX. To showcase the viability of this approach, we use it 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 our system introduces a reduced overhead to vanilla Spark while offering strong additional protection guarantees under powerful attackers and threat models.

Keywords: TEE, Trusted Hardware, Stream Processing, Intel SGX, Spark

Resum

Using Trusted Execution Environments for Secure Stream Processing of Medical Data

per Carlos Segarra González

El processat de dades de caràcter personal, especialment aquelles provinents de dominis mèdics, en servidors remots al núvol, és particularment delicat quan es vol preservar la privacitat dels usuaris que les generen. Molt habitualment, aquestes dades provenen de petits sensors que l'usuari du posats i que emeten un flux continu de mesures. Dit volum de mesures no només han de ser processades de manera eficient, sinó també de manera segura, fins i tot contra hipotètics atacants amb accés privilegiat a les màquines al núvol. La recent introducció al mercat de processadors amb Entorns d'Execució Segura (*Trusted Execution Environments*), com ara Intel SGX, faciliten la implementació de solucions més flexibles i lleugeres que les basades en esquemes de criptografia homomòrfica.

Aquest Treball de Final de Grau presenta una plataforma de processament segur de fluxos que es basa en Intel SGX. Per il•lustrar la viabilitat de la plataforma, la usem en un entorn mèdic. En el decurs del treball, dissenyem i implementem un sistema prototip que evaluem amb jocs de dades reals. Els nostres resultats experimentals mostren que la plataforma introdueix un discret ralentiment en comparació amb la implementació estàndard de Spark, tot oferint un nivell de protecció adicional per a les dades dels usuaris.

Paraules Clau: Entorns d'Execució Segura, Computació Segura, Processament de Fluxos, Intel SGX, Spark

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

API Application Programing Interface.

DFT Discrete Fourier Transform.

DSM Data Stream Manager.

ECG Electrocardiogram.

HR Heart Rate.

HRV Heart Rate Variability.

IaaS Infrastructure-as-a-Service.

JVM Java Virtual Machine.

LoC Lines of Code.

LSDS Large Scale Data & Systems.

MEE Memory Encryption Engine.

OS Operating System.

PPG Photoplethysmogram.

RHS Right Hand Side.

SFTP Secure File Transfer Protocol.

SGX Software Guard eXtensions.

SHM Shared Memory.

SSHFS Secure SHell File System.

TA Trusted Application.

TEE Trusted Execution Environment.

TLS Transport Layer Security.

UniNe University of Neuchâtel.

VMM Virtual Machine Monitor.

Chapter 1

Introduction

1.1 Motivation

Internet of Things (IoT) devices are more and more pervasive in our lives [3]. The number of devices owned per user is anticipated to increase by 26× by 2020 [4]. These devices continuously generate a large variety of data. Notable examples include location-based sensors (*e.g.*, GPS), inertial units (*e.g.*, accelerometers, gyroscopes), weather stations, and, the focus of this paper, wearable sensors that monitor human-health data (*e.g.*, blood pressure, heart rate, stress).

Also referred as P4 medicine (Predictive, Preventive, Personalized, and Participatory), personalized health uses this continuous stream of human-health data to make more targeted and effective diagnoses and treatments [5]. Medical decisions are based on the predicted response of each particular user. To implement personalized health ecosystems, a fleet of devices monitoring and processing each person's vital signs is of crucial importance.

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 presents 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 [6] or full-homomorphic [7] encryption), their current computational overhead makes them impractical in real-life scenarios [8].

The recent introduction into the mass market of processors with embedded trusted execution environments (TEEs), *e.g.*, Intel Software Guard Extensions (SGX) [9] (starting from proces-

sors with codename Skylake) or ARM TrustZone [10], 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 [11], Amazon [12], IBM [13], Microsoft [14]) 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. Algorithms for analyzing cardiovascular signals are getting more complex and computationally-intensive. Thus, traditional signal-processing approaches [15] have now evolved to more advanced solutions like deep neural networks [16, 17]. This increase in computational expenditure has moved the processing towards centralized centers (*i.e.*, the cloud) with bigger and more dynamic processing power. In this work we present a system that computes in real time several metrics of the heart-rate variability (HRV) streaming from wearable sensors. While existing stream processing solutions exist [18, 19], 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.

1.2 Contributions

In this section we highlight the contributions the work here presented makes, and also the components and resources we are given privileged access to.

The contributions of this thesis are twofold. First, we design and implement a system that can process cardiac 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 §2). Note that our design is flexible enough to be used with different stream processing systems (as further described later), and other input data streams. Second, we compare our system against the vanilla, non-secure Spark. We perform an exhaustive assessment on the introduced overhead, and we conclude that the introduced slow-down factor is reasonable even for large datasets and high workloads. What leads us to conclude that the technology is almost ready for production environments.

As introduced before, our design has SGX-Spark in its processing core. SGX-Spark is a modification to Spark to run security sensitive code inside SGX (see §2.1). It is an on-going project in the Large-Scale Data & Systems group from the Imperial College London [20]. We have been given early-access to the code in order to use it in our platform and provide a performance assessment.

1.3 Document Structure

The structure of the rest of this thesis is the following one. In Chapter 2 we introduce the preliminaries required to follow the rest of the sections. We divide it in §2.1 where we introduce the concepts that surround the design, implementation, and deployment of the system. And §2.2 where we introduce and motivate the envisioned use-case, and contextualize the data streams we will feed our streaming platform with. In Chapter 3 we cover the state of the art for privacy preserving stream processing engines of medical data. In Chapter 4 we first describe our system's architecture (§4.1, §4.2) and then we go on to introduce our threat model (§4.3) and our system's known vulnerabilities (§4.4). Then, in Chapter 5, we describe how the previously introduced architecture is implemented (§5.2, §5.1) and deployed (§5.3). A complete and exhaustive evaluation of the system is then presented in Chapter 6. In particular, we first describe the evaluation context: hardware settings (§6.1), experiment configuration (§6.2), analyzed metrics (§6.3), and injected workloads (§6.4). To then present the obtained results in §6.5. Lastly, further research lines and the thesis' conclusions are presented in Chapters 7 and 8 respectively.

Chapter 2

Background

In this chapter we introduce general concepts to better understand the design and implementation details of our system. Firstly, we introduce the hardware, software and programming frameworks that we leverage, and then the medical technicalities regarding the data we use and the processing we make of it. In §2.1, we cover technical aspects exploited in the remaining of this work, specifically: we describe the concept of Trusted Execution Environment, the operating principles of Intel SGX and Spark, and lastly we present two key frameworks developed by the *Large Scale Data & Systems* [20] group at the Imperial College London, SGX-LKL and SGX-Spark. In §2.2, we describe the specificities of the data streams from the medical domain our system deals with, how this data streams are obtained, together with the required processing that our system allows to offload on an untrusted cloud provider, and how this processing can be useful in a real use case.

2.1 Technical Background

2.1.1 Trusted Execution Environments and Intel SGX

A *trusted execution environment (TEE)* is an isolated area of a main processor that provides code and data therein contained with confidentiality and integrity guarantees [21]. Confidentiality refers to preventing unauthorized parties from accessing sensitive information and integrity to ensuring that sensitive code and data is not tampered with. An application developed to run and be deployed in a TEE is called a *Trusted Application (TA)*. Trusted Execution Environments have already been available for several years in the main CPU vendors' commodity CPUS. Arm TrustZone has been part of Arm's architecture since v6 for Cortex-A processors (2012) and v8 for Cortex-M (2018). Intel® Software Guard Extensions (SGX) were introduced with the sixth generation of Intel's processors codename *Skylake* in 2015.

In comparison with Arm TrustZone, Intel SGX include a remote attestation protocol, support multiple trusted applications on the same CPU, its SDK is easier to program with, and there is a greater variety of programming frameworks to develop SGX-based TAs. Most importantly, all

the major Infrastructure-as-a-Service (IaaS) providers (Google [11], Amazon [12], IBM [13], Microsoft [14]) are nowadays offering nodes with SGX processors. For these reasons, Intel SGX is our chosen hardware solution to deploy our platform in.

Intel *Software Guard eXtensions* are a set of new instructions and memory access changes added to Intel's architecture. These extensions enable applications to create hardware-protected containers in their address space, referred to as *enclaves*. An enclave provides confidentiality and integrity even in the presence of malicious privileged software such as virtual machine monitors (VMM), BIOS, or operating systems (OS) [22]. At initialization time, the code and data is free for inspection and, once loaded to the enclave, the latter 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. The application can also verify that its secure code is running in a genuine enclave using the same attestation protocol via platform specific keys.

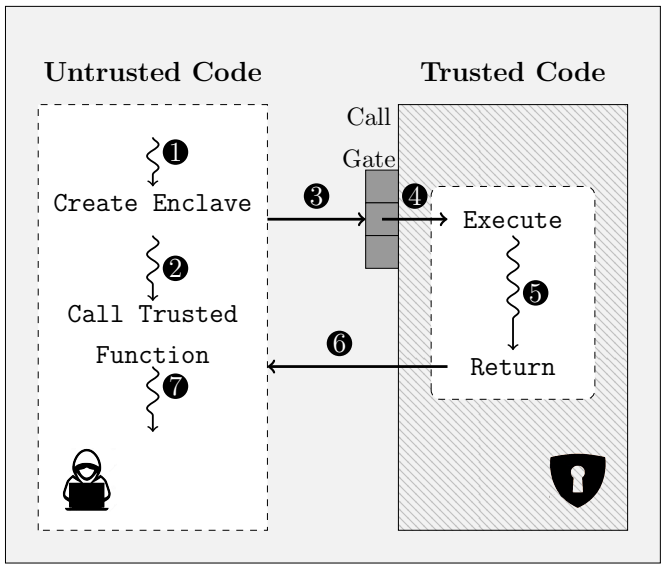

Figure 2.1: Intel SGX execution workflow.

Services using SGX divide its source code in an untrusted and a trusted part. The former deployed outside the enclave and the latter inside. Figure 2.1 breaks down the typical execution workflow of SGX services. After the initial attestation protocol, code in the untrusted region creates an enclave and securely loads trusted code and data inside (Figure-❶). Whenever this untrusted code wants to make use of the enclave, it makes a call to a trusted function (Figure-❸) that gets captured by the call gate mechanism and, after performing sanity and integrity checks (❹), gets executed (❺), the value returned (❻) and the untrusted code can resume execution (❼). It is important to stress that the security perimeter is kept at the CPU package and, as a consequence, all other software including privileged software or even other enclaves are prevented from accessing code and data located inside the enclave. In particular, 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 (MEE)* [23].

2.1.2 Spark and Spark Streaming

Apache Spark is a cluster-computing framework to develop scalable, fault-tolerant, distributed applications. It builds on RDDs, resilient distributed datasets [22], 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 [24] is an extension of Spark's core API that enables scalable, high-throughput, fault tolerant stream (mini-batch) processing of data streams [25]. We leverage on Spark Streaming to perform file-based streaming, by monitoring a filesystem interface outside the enclave and processing new files as they are loaded. In particular, and as detailed later in Chapter 5, we use the *Discretized Streams* API [18].

2.1.3 SGX-LKL and SGX-Spark

Developed at the *Large Scale Data & Systems Group (LSDS)* [20] at the *Imperial College London*, SGX-LKL [26] is a library OS to run unmodified Linux binaries inside enclaves. It provides support for complex applications and managed runtimes enabling in-enclave user-level threading, signal handling, and paging. Namely, it allows the execution of a full *Java Virtual Machine (JVM)* inside an enclave. 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 [27] builds on SGX-LKL. It partitions the code of Spark applications to execute the sensitive parts inside SGX enclaves. Figure 2.2 depicts its architecture. The engine deploys two collaborative Java Virtual Machines (JVM), one outside (Figure 2.2, left) and one inside the enclave (Figure 2.2, right) for the driver, and two more for each worker deployed in the cluster. Spark code outside the enclave accesses only encrypted data. The communication between the JVMs is kept encrypted and is performed through the host OS shared memory (SHM). 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 SGX enclaves. This is, the user must only compile the source code together with SGX-Spark's and, as long as the op-

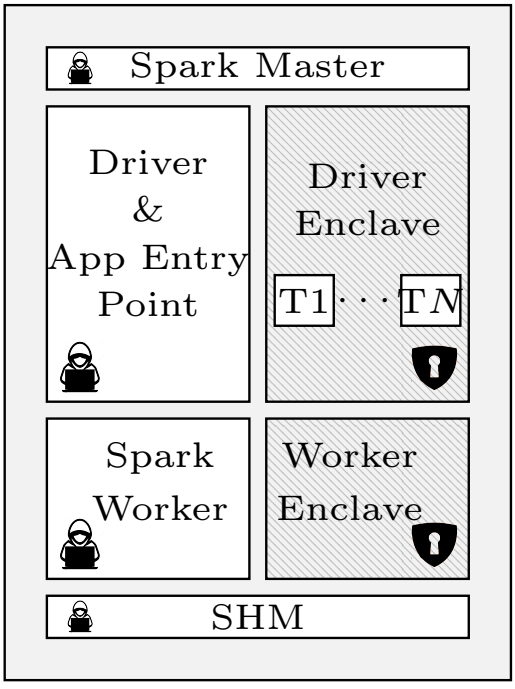

Figure 2.2: SGX-Spark attacker model and collaborative structure scheme.

erators used are supported by the framework, execution is seamlessly deployed inside the enclave with *no* amendments to the vanilla Spark implementation.

2.2 Cardiac 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 Chapter 4, 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). ECG-based systems measure the heart's electrical activity over time and is the chosen method by chest-based sensors [28]. PPG-based systems measure the variation of blood volume over time using LEDs and photodiodes. Although less precise, PPGs are the chosen technique by all wrist-based cardiac monitoring sensors [29].

In both cases, ECG and PPG based, we contemplate the usage of wearable sensors. Wearable technologies are electronic devices that are incorporated into items which can comfortably be worn on the body [30]. Due to space and power constraints, these sensors' memory, computing power, and communication capabilities are limited. As a consequence, to be embedded in a functional ecosystem, they rely on a gateway that forwards the information generated by the sensors to the cloud. This environment is depicted in Figure 2.3.

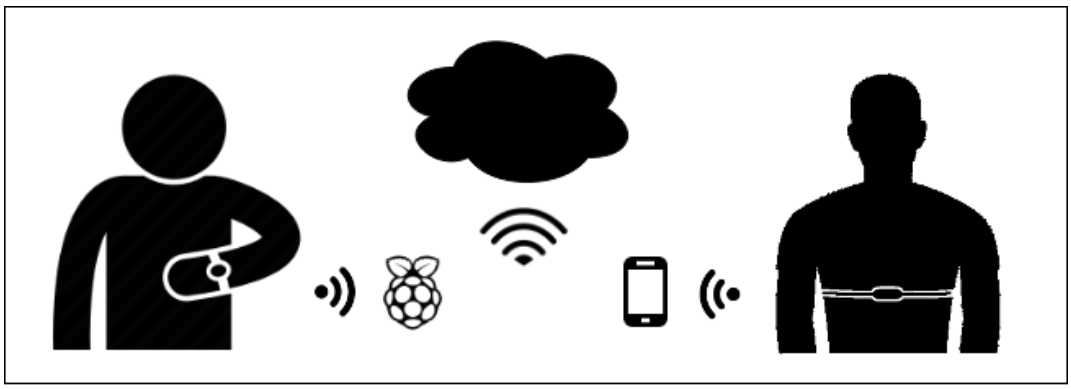

Figure 2.3: Envisioned sensor ecosystem composed of: a user, a wearable device, a gateway with internet connection, and the cloud were results are stored and processed.

The generation of the approximated diagram (ECG or PPG) and the time measures are done inside the sensor. Figure 2.4 depicts a schematic representation of an ECG and the values streamed from the sensor to the gateway: R peak's timestamps and RR intervals.

In our case, we focus on the analysis of the Heart Rate Variability (HRV) [32], that is, the analysis of the variation in the time intervals between heartbeats (a.k.a. RR intervals). The HRV

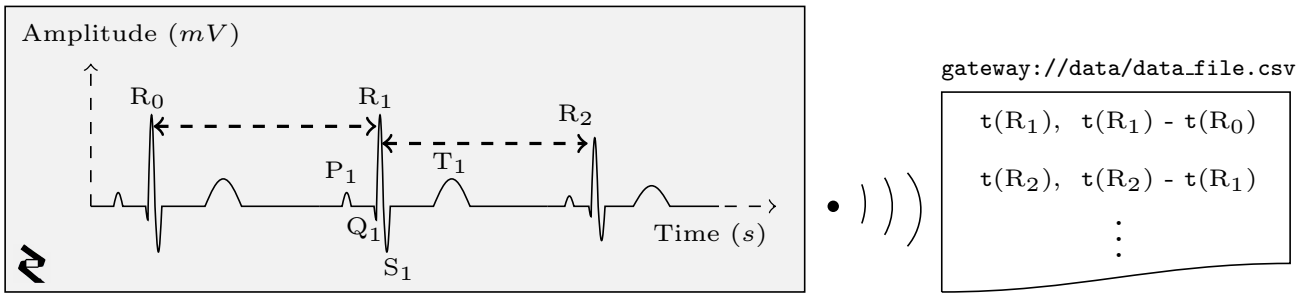

Figure 2.4: Schematic representation of an ECG signal showing three normal beats. A normal electrocardiogram can be broken down in three waves: a *P wave* corresponding to the depolarization of the atria, a *QRS complex* corresponding to the depolarization of the ventricles and a *T wave* corresponding to the repolarization of the ventricle [31]. From an ECG the sensor extracts and streams the R-peaks' timestamp and the time elapsed between them.

is of utmost importance since it has been shown to be a predictor for myocardial infarction [33, 34]. 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. Finally, despite our system being specifically designed for streams with these data features, its modular design (as we later describe in Chapter 4) makes it easy to adapt to other use-cases.

Chapter 3

Related Work

In this Chapter we cover the related work that sets the current State-of-the-Art for privacy preserving stream processing engines of medical data. Together with each piece of work, we provide a brief description and argue why it has or has not been considered for our project. In §3.1 we cover the most notable big data stream processing engines that relate with our system. Note that stream processing is a big field and it is out of the scope of this work to provide a detailed survey. In §3.2 we introduce relevant privacy-preserving processing engines: both stream and batch based. Lastly, in §3.3 we cover other approaches at cardiac data monitoring.

3.1 Stream Processing Engines

Stream processing has recently attracted a lot of attention from academia and industry [35, 36, 37]. Apache Spark [24] is arguably the de-facto standard in this domain, by combining batch and stream processing with a unified API [25]. Apache Spark SQL [38] allows to process structured data by integrating relational processing with Spark's functional programming style. Structured streaming [39] leverages Spark SQL and it compares favorably against the discretized counterpart [25] in terms of performance. However, the former lacks security or privacy guarantees, and hence it was not considered. Furthermore SGX-Spark only has support for the *Discretized Streams* API, and therefore we also rely on it.

3.2 Privacy-Preserving Computation

Opaque [40] 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 our system's. However, *(1)* the Spark master must be located in the client side, for it must be trusted. An scenario that does not fit in our multi-client setting. And *(2)*, it requires changes to the application code, and hence is not transparent to the

user. In `oblivious` mode, *i.e.*, protecting against traffic pattern analysis attacks, it can be up to 46× slower, a slow-down factor not tolerable for the real-time analytics in the scope of our system.

SecureStreams [19] is a reactive framework that exploits Intel SGX to define dataflow processing by pipelining several independent components. In order to use this framework, the user requires a good knowledge of the underlying implementation. Moreover, applications must be written in the Lua programming language, hindering its applicability to legacy systems or third-party programs.

DPBSV [41] 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 *Public Key Infrastructure (PKI)* 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.

TaLoS [42] is a *Transport Layer Security (TLS)* [43] library that terminates connections by maintaining sensitive information inside enclaves. The authors provide custom wrappers for the most common TLS API calls and securely store users and sessions' keys inside enclaves. The embedding with a native Spark, or SGX-Spark, application is not transparent to the end user and hence why we discard it.

Lastly, homomorphic encryption [44] does not rely on trusted execution environments and offers the promise of providing privacy-preserving computations over encrypted data. This is, it generates an encrypted result from two encrypted operands, corresponding to the encrypted value of operating both operands in plain text. The key feature of the scheme is that, at no point, neither the operands nor the result are processed in clear. While several works analyzed the feasibility of homomorphic encryption schemes in cloud environments [45, 46], the performance of homomorphic operations [47] is far from being pragmatic.

3.3 Cardiac Monitoring Systems

For the specific problem of HRV analysis, while periodic monitoring solutions exist [48], 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 we consider. Similarly, another solution [17] uses convolutional networks for the detection of arrhythmias. However, the authors take no considerations with regard to data security and privacy.

This work is one of the first attempts to building a privacy-preserving real-time streaming system specifically designed for medical and cardiac data.

Our system fills the existing research gap by proposing a system that leverages Intel SGX enclaves to compute such analytics over public untrusted clouds without changing the existing Scala-based source code.

Chapter 4

Architecture

The aim of this Chapter is to depict our system's architecture. A high-level abstraction is presented in Figure 4.1, where each different component is represented together with the path data follows. We follow a client-server organization and each functionality is designed to be modular and self-contained for the ease of deployment, evaluation, scalability and availability.

On general lines, the server-side component is executed on untrusted machines (for instance nodes on the cloud) where Intel SGX is available. There, the SGX-Spark engine is deployed and stream processes the data generated by an arbitrary number of clients using a set of medical algorithms. Each client consists of a sensor and a gateway: the former generates samples in a continuous fashion, and the latter aggregates them and periodically sends them to the cloud-based component. Similarly, the gateway fetches the results every fixed time intervals. A filesystem is mounted at the server-side to interface the interaction between the clients and the processing engine. Each client data stream is processed in parallel independently of the chosen algorithm, and results are also stored separately.

The remaining of the chapter is structured as follows: §4.1 details the server-side architecture, §4.2 does the same with the client-side component, in §4.3 we cover the considered threat model and the security assumptions we make in the design and lastly in §4.4 we present our project's known vulnerabilities.

4.1 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 and also stores the algorithm results. This way, the gateway can fetch with the desired frequency the processed information. Currently, the filesystem is mounted and unmounted, respectively at start-up time and upon the shutdown of the service. SGX-Spark monitors the directory and processes new data as it is loaded.

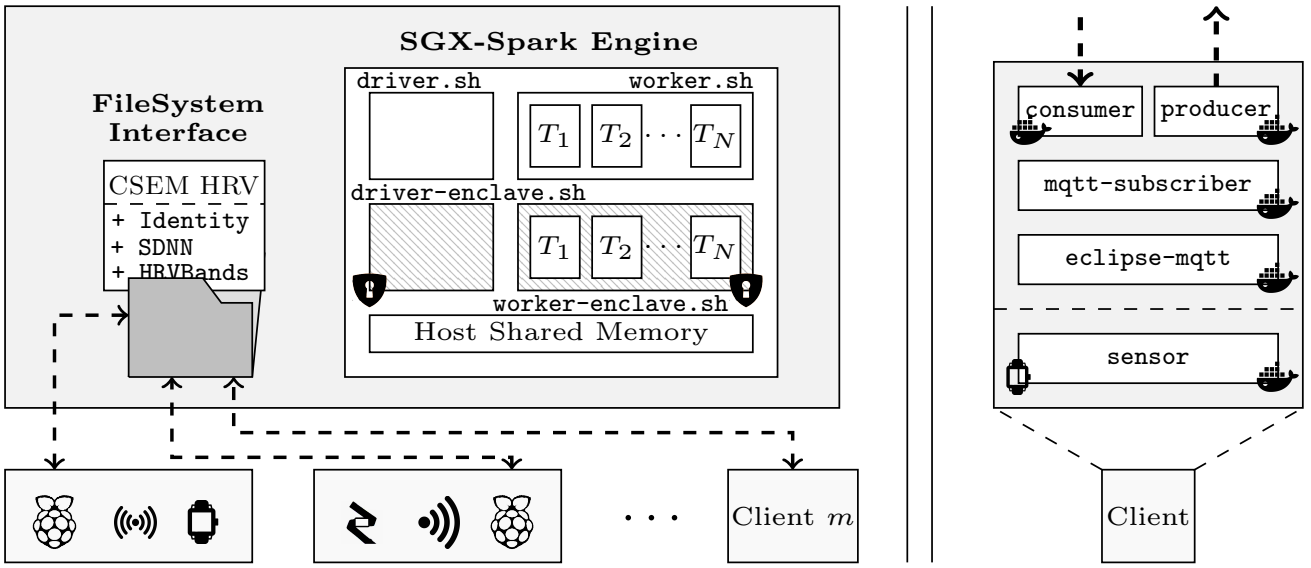

Figure 4.1: (Left) Schematic of the system's main architecture. A set of clients bidirectionally stream data to a remote server. The interaction is done via a filesystem interface. On the server side, SGX-Spark performs secure processing using different HRV analysis algorithms. (Right) Breakdown of a 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.

The streaming engine and the pool of algorithms are compiled together by the same toolchain, yet they are independent. A Spark job deployed in standalone mode executes: the master process for resource management and allocation (not included in Figure 4.1), a driver process that orchestrates the execution, and an arbitrary number of worker processes executing tasks. In the case of SGX-Spark jobs, two Java Virtual Machines (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 Figure 2.2). A process outside the enclave never sees data in clear since sensitive operations are *always* executed in secured environments. Note that, for each newly deployed worker, a new pair of JVMs and a new enclave are also deployed.

SGX-Spark requires that algorithms are compiled together with the engine so that code can be loaded inside enclaves. However, the specific algorithm that we will execute is currently set at start-up time. It is important to note that each client has its own dedicated data stream assigned, hence being able to choose different algorithms. These will be executed concurrently, each yielding separated results.

4.2 Clients

The client is a combination of: a sensor that constantly generates data, and a gateway that interacts with the remote end (see the right hand side (RHS) of Figure 4.1). For evaluation purposes, the sensor component is replaced by a synthetic data generator that simulates samples. As introduced in §2.2 and depicted in Figure 2.4, the samples produced correspond to RR intervals and their timestamps. The data generator (or fake sensor) streams samples to the gateway which is composed of: a broker and subscriber that receive the samples, a producer that aggregates them, generates files of a fixed size, and streams them to the filesystem interface, and lastly a consumer that fetches the processed data from the remote endpoint. To get a grasp on the volume of data a single client generates, each sample is a couple of bytes and a healthy individual generates between 50 and 180 samples per minute. As a consequence, an average client generates around 230—690 Bytes of data per minute. This is indeed a low throughput but, as shown in Chapter 5, the system can withhold way higher loads.

4.3 Threat Model

In this Section we cover our threat model and the main security assumptions we make. This is, what kind of attacker our system is protected from. Vulnerabilities out of the scope of this project, together with known issues are covered in §4.4. Firstly, we assume that the communication between the gateway and the filesystem is kept protected (*e.g.*, encrypted) using secure transfer protocols (*e.g. Secure File Transport Protocol (SFTP)*, more in Chapter 5). Secondly, we trust the whole client package. Protecting it is out of the scope of this work, but suggestions are given in Chapter 7. Given these assumptions, our threat model is the same as typical systems that rely on SGX. Specifically, we assume an attacker with access to system's privileged software such as the OS, VMM or BIOS. Our security perimeter only includes the internals of the CPU package and the *on-die* memory. Most notably, the system's main memory (DRAM) is left *outside* our security perimeter. As a consequence, the traffic between enclave applications and main memory is kept encrypted and is handled by the MEE [23]. The trusted computing base is Intel's microcode and the code loaded at the enclave, which can be measured and integrity checked. It is worth noting that, any bug or security leak included in the application code loaded in the enclave might compromise our security model.

4.4 Known Vulnerabilities

The threat model described in §4.3 is the same of SGX. As a consequence, and given our security assumptions, it is sufficient to look into the known vulnerabilities of the latter. Providing protection

from attacks outside of SGX's threat model is out of the scope of this project. Intel's MEE is not designed to be an oblivious RAM, therefore an adversary could perform traffic analysis attacks [23] against our system. This is, even if communication between the CPU and the DRAM is kept encrypted, a smart attacker could infer information from patterns in the message exchanges. However, current work enables oblivious computations on enclaves [40]. In March 2017, Schwarz *et. al.* [49] unveiled a side-channel timing attack capable of extracting a full RSA key from an enclave in under 5 minutes. Not long after, various countermeasures were made available [50, 51]. Speculative execution attacks (*Spectre*-like [52]) have also proven to be successful against enclaves [53]. Lastly, *Foreshadow* [54] is another speculative execution attack stronger than its predecessors and specifically targeted against Intel SGX.

Chapter 5

Implementation

The aim of this chapter is to present the implementation details of our system. Each explanation is accompanied, when required, by code snippets to further illustrate the rationale behind our design choices. For the full implementation details we refer to Appendix A. The implementation we will cover is the one the results presented in Chapter 6 are based on. As a consequence, and in order to stress-test the system, we replace real sensors with synthetic data generators in the client package first described in §4.2. All the different components introduced in Chapter 4 are packaged in sets of Docker containers. Large fleets of concurrent users are then simulated using Docker's standalone clusters and mounted using `docker-machine`.

The rest of the chapter is structured as follows. In §5.1 and §5.2 we cover the implementation details of the server-side and the client-side component respectively. Re-using Docker-specific nomenclature, we will refer to logical isolated functionalities packaged in a container as *services*. Sets of services working collaboratively will be gathered and deployed as a *component* using `docker-compose`. Collections (or clusters) of replicated components form *swarms* using `docker-swarm`. Lastly, in §5.3 we cover how each service, component and swarm is deployed.

5.1 Server Implementation

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. To do so, we include our newly added module in the project's `pom.xml` compilation file, in the worker and driver initialization scripts and in the enclave generation `Makefile` so that code is loaded in the enclave at compilation time. The before-mentioned module contains two HRV processing algorithms and a benchmarking one: `SDNN`, `HRVBands`, and `Identity`. The `SDNN` algorithm computes the standard deviation of NN (RR) intervals in a rolling basis, generating one output per 10 seconds worth of samples. The `HRVBands` [55] performs a Discrete Fourier Transform (DFT) of 10 seconds worth of samples, and computes the power of the low frequency and high

frequency components, together with their ratio. Lastly, the `Identity` algorithm copies the input to the output file. It is used to provide a baseline on the overhead the system is introducing. All the application code is implemented in SPARK's binding for the SCALA programming language [56]. We choose this particular binding since it is the only one supported by SGX-SPARK's compiler. To be usable inside SGX enclaves, applications must adhere to the RDD [57] and DStreams [25] API.

The particular implementation of these algorithms relies on basic Spark Streaming operators, and their corresponding Scala counterparts. For instance, the filesystem interface is monitored using DStream's `textFileStream` method as exposed in Listing 5.1. There, variables and code are initialized (lines 3-10) then the input directory is monitored (line 12) and the computation and output are assigned. Note that data is not processed until the Streaming Context is started via the `start()` method (line 22).

```
1  def main(args: Array[String]) {
2        // Initialize variables and Spark Contexts
3        val wdwSize = 10
4        val numClients = args(0).toInt
5        val conf = new SparkConf().setAppName("HRV Toolbox - SDNN")
6        val sc = new SparkContext(conf)
7        val ssc = new StreamingContext(sc, Duration(10000))
8
9        // Initialize the Estimate Class
10       val estimateHRV = new EstimateHRVBands(wdwSize)
11
12       // Monitor Input Directory
13       val dataStreamVec = for (i <- 1 to numClients) yield ssc.textFileStream("csem
     /src/main/resources/csv/"+i.toString+"/
14 ")
15       // Estimate
16       val hrvBandVec = for (dS <- dataStreamVec) yield estimateHRV.estimate(dS)
17       // Save as Output
18       for (i <- 1 to numClients) {
19         hrvBandVec(i-1).saveAsTextFiles("csem/src/main/resources/results/" + i.
     toString + "/sdnn")
20       }
21       // Start Stream Processing
22       ssc.start()
23       ssc.awaitTermination()
24   }
```

Listing 5.1: Snippet illustrating `textFileStream` functionality.

Lastly, the implementations of `SDNN`, `HRVBands`, and `Identity` can be found in Listings A.1, A.2, and A.3 respectively.

5.2 Client Implementation

Clients correspond to body-sensors strapped to the body of a user working collaboratively with a smart gateway, sending gathered data to the cloud-based component. In a real deployment, the sensor service in the client component would be an actual sensor (*e.g.* HR band, smartwatch, or optical HR monitoring device) and the gateway would be implemented in a, for instance, Raspberry Pi. For evaluation purposes (see Chapter 6), these services are simulated and virtualized. This way, we can simulate a situation where multiple clients concurrently use the platform without the hardware burden. As firstly introduced in §4.2, our implementation decouples the client component into into five different services (see Figure 4.1, right).

1. The `sensor` service is a Python script that generates `sample_rate` random samples per second, where a sample is an RR interval and its timestamp, and publishes the information to the `artificial-data` topic in a MQTT [58] broker located in the gateway. Listing 5.2 presents the part of the `sensor` source code where artificial data is generated and published. The full source code is available in Listing A.4. Note the particularity of the sample generation procedure. First, `sample_rate` random points in a $[0, 1)$ uniform distribution are generated (lines 18, 21). Then, current timestamps are inferred from them (line 24) and intervals are computed as successive differences (line 25). These intervals are, in fact, the RR intervals.

```
1  def loop(sample_rate):
2      """Endless Sample Generation
3
4      This module is the endless loop that generates <sample_rate>
5      new samples per second and publishes them to a MQTT topic.
6
7      Parameters
8      ----------
9      sample_rate : float
10         How many samples are generated per second
11     """
12     killer = Killer()
13     global CL_ID
14     past_ts = time.time()
15     while 1:
16         value = ""
17         # First we generate <sample_rate> timestamps in one second
18         tstamps = [i for i in random.uniform(low=time.time(),
19                                              high=time.time() + 1,
20                                              size=(sample_rate))]
21         tstamps.sort()
22         # Second we compute the intervals between them
```

```
23          for i in tstamps:
24              tmp = datetime.fromtimestamp(i).strftime('%Y-%m-%d %H:%M:%S')
25              value += tmp+','+str(int((i - past_ts) * 1e6))+'\n'
26              past_ts = i
27          publish.single("artificial-data-{}".format(CL_ID), value)
28          time.sleep(1)
29          if killer.kill_now:
30              print("Exiting gracefully!")
31              break
```

Listing 5.2: Snippet illustrating the artificial data generation in the `sensor` service.

2. The `eclipse-mqtt` service is the `mosquitto` [59] implementation of the MQTT broker. We rely on the public Docker image at Docker Hub [60]. It handles the samples generated by `sensor` and delivers them to the `mqtt-subscriber` when the latter subscribes to the `artificial-data` topic.

3. The `mqtt-subscriber` service is a PYTHON script that subscribes to the `artificial-data` MQTT topic and whenever it gathers `FILE_LINES` (line 26) samples it generates a `.csv` file at a specified location. The key functionality of the service, sample gathering and file generation, is illustrated in Listing 5.3. The full service code is available in Listing A.5.

```
1  def on_message(mosq, obj, msg):
2      """Callback method for the mosquitto client.
3
4      Whenever the MQTT client receives a new sample it triggers this method.
5      When the current count reaches the threshold, a new csv file is generated
6      and stored in the local data directory.
7
8      Parameters
9      ----------
10     Parameters are those of the standard implementation of the MQTT client
11     for Python.
12     """
13     global count
14     global curr_str
15     global FILE_LINES
16     global CL_ID
17     global LOCAL_DATA_DIR
18     global killer
19     print("Received message")
20     dcd_msg = msg.payload.decode("utf-8").split("\n")
21     whole_msg = [i.split(",") for i in dcd_msg if i != ""]
22     for tmp in whole_msg:
```

```
23          d = datetime.strptime(tmp[0], '%Y-%m-%d %H:%M:%S')
24          time_stamp = time.mktime(d.timetuple())
25          rr = tmp[1]
26          if count == FILE_LINES:
27              global TS
28              print("It took: {} s".format(time.time() - TS))
29              curr_str += "{},{}".format(time_stamp, rr)
30              ts = int(time.time()*1e3)
31              filename = "{}{}_".format(LOCAL_DATA_DIR, CL_ID)+str(ts)+".csv"
32              try:
33                  with open(filename, "w+") as f:
34                      f.write(curr_str)
35              except FileNotFoundError as e:
36                  if killer.kill_now:
37                      mosq.disconnect()
38                      print("Exiting gracefully!")
39                      break
40              count = 0
41              curr_str = ""
42          else:
43              curr_str += "{},{}\n".format(time_stamp, rr)
44              count += 1
45      else:
46          if killer.kill_now:
47              mosq.disconnect()
48              print("Exiting gracefully!")
```

Listing 5.3: Snippet of the data gathering and file generation in the `mqtt-subscriber` service.

4. The `producer` service is a PYTHON script that monitors a local directory and, whenever a new file is stored, sends it to the remote filesystem. The monitoring functionality is depicted in Listing 5.4 and the full source code is available in Listing A.6. We keep track of the current files in the directory with a dictionary (lines 175 and 180) and we compute the difference between the pre and the post (line 181). We then transfer the new files over SFTP (line 188).

```
1  def monitor(ssh, sftp, *args):
2      """Sending daemon
3
4      This method starts an infinite loop that every <SEND_PERIOD> looks for
5      newly added data files and copies them to a remote directory.
6
7      Parameters
8      ----------
9      ssh : paramiko.SSHClient
10         SSH Client connected to the remote host.
```

```
    sftp : paramiko.SFTPClient
        SFTP Client connected to the remote host.
    """
    global LOCAL_DATA_DIR
    path_to_watch = LOCAL_DATA_DIR
    before = dict([(f, None) for f in os.listdir(path_to_watch)])
    global SEND_PERIOD
    killer = Killer()
    while 1:
        time.sleep(SEND_PERIOD)
        after = dict([(f, None) for f in os.listdir(path_to_watch)])
        added = [f for f in after if f not in before]
        if added:
            for f in added:
                local_file = LOCAL_DATA_DIR + str(f)
                global REMOTE_DATA_DIR
                remote_file = REMOTE_DATA_DIR + "/" + str(f)
                sftp.put(local_file, remote_file)
                os.remove(local_file)
                # If there are a lot of files, not checking inside also leads
                # to error with code 137
                if killer.kill_now:
                    print("Exiting gracefully!")
                    break
        before = after
        if killer.kill_now:
            print("Exiting gracefully!")
            break
    for d in os.listdir(LOCAL_DATA_DIR):
        tmp_file = LOCAL_DATA_DIR + str(d)
        os.remove(tmp_file)
```

Listing 5.4: Snippet illustrating the local directory monitoring in the `producer` service.

5. Similarly, the `consumer` service is a PYTHON script that monitors the remote filesystem and, whenever a result file is stored, fetches it to the local result directory. The monitoring functionality is illustrated in Listing 5.5 and the whole source code is available in Listing A.7. It is, in essence, symmetric to the one presented for the `producer` service. The only difference is that, since `sftp.get()` does not support recursive invocations (*e.g.* `-R` flags), specific care must be taken to fetch all the different files placed in the different directories.

```
def monitor(ssh, sftp):
    """Fetching daemon

```

```
    This method starts an infinite loop that every <FETCH_PERIOD> looks for
    newly added directories and copies them to a local result directory.

    Parameters
    ----------
    ssh : paramiko.SSHClient
        SSH Client connected to the remote host.
    sftp : paramiko.SFTPClient
        SFTP Client connected to the remote host.
    """
    global REMOTE_RESULT_DIR
    global FETCH_PERIOD
    global LOCAL_RESULT_DIR
    #print("CONSUMER: user -> {}".format(getpass.getuser()))
    path_to_watch = REMOTE_RESULT_DIR
    before = dict([(f, None) for f in sftp.listdir(path_to_watch)])
    killer = Killer()
    while 1:
        time.sleep(FETCH_PERIOD)
        after = dict([(f, None) for f in sftp.listdir(path_to_watch)])
        added = [f for f in after if f not in before]
        if added:
            for f in added:
                local_dir = LOCAL_RESULT_DIR + str(f) + "/"
                remote_dir = REMOTE_RESULT_DIR + "/" + str(f) + "/"
                try:
                    os.mkdir(local_dir)
                    get_all(sftp, remote_dir, local_dir)
                except FileExistsError as e:
                    print("Fetching a file that already exists!")
        before = after
        if killer.kill_now:
            print("Exiting gracefully!")
            break
    for d in os.listdir(LOCAL_RESULT_DIR):
        tmp_dir = LOCAL_RESULT_DIR + d
        for f in os.listdir(tmp_dir):
            tmp_file = tmp_dir + "/" + f
            os.remove(tmp_file)
        os.rmdir(tmp_dir)
```

Listing 5.5: Snippet illustrating the remote filesystem monitoring in the `consumer` service.

All the services implemented in Python, amount to a total of 888 Lines of Code (LoC). As introduced before, for the service deployment we rely on Docker. The corresponding `Dockerfiles`

and `docker-compose` are presented in §5.3. Furthermore, the deployment of each service is also wrapped in a `Dockerfile` and included in an image. Lastly, we remark that the communication between the client and the server happens via SSH/SFTP to ensure transport layer security when transferring user's data.

5.3 Deployment

To ease scalability and reproducibility of both server and client, deployment is orchestrated by a single script detached from both execution environments. This is done through a combination of `bash` scripts that act as entry points for the `Dockerfile` and `docker-compose` file. We differentiate between the server deployment (§5.3.1), the client deployment (§5.3.2) and the overall deployment and evaluation (§5.3.3).

5.3.1 Server Execution Deployment

Specifying the remote location; the SGX-Spark engine, the streaming algorithm, and the filesystem interface are initialized. Depending on whether we want to enable enclaves or not (`SGX_MODE`, line 24), a set of scripts (lines 25 and 26) are executed in the remote server using `ssh.exec_command()`. The script then remains passive until a `SIGTERM` signal is captured, and all processes are killed (lines 41-49). A snippet of the main functionality is attached in Listing 5.6 and the full source code is available in Listing 5.3.1. Note that the code is intended to be launched from a location different from the server (which we leave untrusted).

```
1  def main(ssh, sftp):
2      """Main execution method
3
4      This method contains the main deployment of UC1. It basically starts each
5      and every process, launches the benchmarking, and cleans everything when
6      execution has finished.
7
8      Notes
9      -----
10     All the sleeps are placed due to experienced errors race conditions. As a
11     consequence their arguments are completely empirical.
12
13     Arguments
14     ---------
15     ssh : paramiko.SSHClient
16         SSH Client to the remote server.
17     sftp : paramiko.SFTPClient
```

```
        SFTP Client to the remote server.
    """
    global ALGORITHM
    global SGX_MODE
    global NUM_CLIENTS
    algo_name = "./launch-csem-{}.sh {}".format(ALGORITHM, NUM_CLIENTS)
    if SGX_MODE:
        script_list = ["./master.sh", "./worker.sh", "./worker-enclave.sh",
                       "./driver-enclave.sh", algo_name]
    else:
        script_list = ["./master.sh", "./worker.sh", algo_name]
    for script in script_list:
        _in, _out, _err = ssh.exec_command("cd sgx-spark; {} &".format(script))
        time.sleep(3)
    tmp_command = "dstat -l --nocolor --noheaders 10 > cpu_load.csv"
    _, _out, __ = ssh.exec_command("cd sgx-spark; {} &".format(tmp_command))
    _out.readlines()
    killer = Killer()
    while 1:
        time.sleep(1)
        if killer.kill_now:
            print("Killing Gracefully!")
            break
    for script in script_list:
        _, _out, __ = ssh.exec_command("kill $(pgrep -f '{}')".format(script))
        _out.readlines()
    _, _out, __ = ssh.exec_command("kill $(pgrep -f 'java')")
    _out.readlines()
    _, _out, __ = ssh.exec_command("kill $(pgrep -f 'dstat')")
    _out.readlines()
    _, _out, __ = ssh.exec_command("rm /dev/shm/*")
    _out.readlines()
    return 0
```

Listing 5.6: Main method of the Server-Side Deployment Script.

5.3.2 Client Execution Deployment

The client deployment procedure is more complicated than the server's one since we support executions of variable number of clients, which are instantiated at start-up time. In short, when the number of clients is set, the platform starts a standalone Docker cluster (or *swarm*). However, once started with the scalability tests, we recalled that the number of clients a cluster whose network relies on Docker's default `bridge` adapter can support was too small. At 20 clients, the

engine ran out of local IP addresses and executions would systematically fail. As a consequence, we decided to virtualize a local standalone cluster using `docker-machine`. This latter functionality enables the deployment of several Docker engines in a single computing instance. To do so, a virtual machine must be started for each engine, and special care must be taken to set up the network, so that all containers are reachable from the others.

Even though the full script is rather long, given its complexity and how crucial it is to the system's performance, we attach it entirely in Listing 5.7. We firstly deploy a name discovery service based on the Consul [61] docker image, the aim of which is to keep track of all the generated virtual machines, and assign an IP address to each one (lines 27-36). Secondly, we start $\lceil$`NUM_CLIENTS`$/20\rceil$ virtual machines and register them with the name discovery service (lines 40-60). These set of virtual machines will form our Docker Swarm. As a consequence, it is important to elect one (first by default) as our `swarm-master`. Note that, in order to speed up the start-up time, the images for all the client services are compressed and pre-loaded to the virtual machines (lines 62-66). In third place, we create the overlay network (lines 70-71). Then, the local directories for each client in each virtual machine must be mounted and mirrored outside the virtual machine so that we can process the results. This is done using SSHF in lines 76-85. Lastly, each client is deployed using the specific `docker-compose` file (see Listing A.9).

```
1  #!/bin/bash
2
3  set -a
4
5  # Predefined enviroment variables
6  source docker-env.env
7
8  # Pull MQTT image
9  docker-compose pull mqtt
10
11 # Max Containers per Host
12 MAX_CONTAINERS=2
13
14 # Input Parameters
15 SAMPLE_RATE=${1}
16 NUM_CLIENTS=${2}
17 FILE_LINES=${3}
18
19 NUM_HOSTS=$(( $(( $NUM_CLIENTS / $MAX_CONTAINERS )) + 1))
20
21 # To introduce an abstraction layer, even if there is no real need for an
22 # overlay network (the bridge itself suffices) we create an external machine
23 # anyways. This may be an overkill for very small executions but is introduced
24 # to preserve scalability transparent to the user.
```

```
25
26 # We first create the Consul discovery service (key/value store)
27 docker-machine create \
28     -d virtualbox \
29     --virtualbox-boot2docker-url ~/tmp/boot2docker.iso \
30     mh-keystore
31 eval "$(docker-machine env mh-keystore)"
32 docker run -d \
33   --name consul \
34   -p "8500:8500" \
35   -h "consul" \
36   consul agent -server -bootstrap -client "0.0.0.0"
37
38 # We then create the Swarm cluster the first node (id == 1) will be the swarm
39 # manager.
40 for HOST_ID in $(seq 1 ${NUM_HOSTS});
41 do
42     if [ $HOST_ID == 1 ]; then
43         docker-machine create \
44             -d virtualbox \
45             --virtualbox-boot2docker-url ~/tmp/boot2docker.iso \
46             --swarm --swarm-master \
47             --swarm-discovery="consul://$(docker-machine ip mh-keystore):8500" \
48             --engine-opt="cluster-store=consul://$(docker-machine ip mh-keystore)
    :8500" \
49             --engine-opt="cluster-advertise=eth1:2376" \
50             "sgx-csem-host-$HOST_ID"
51     else
52         docker-machine create \
53             -d virtualbox \
54             --virtualbox-boot2docker-url ~/tmp/boot2docker.iso \
55             --swarm \
56             --swarm-discovery="consul://$(docker-machine ip mh-keystore):8500" \
57             --engine-opt="cluster-store=consul://$(docker-machine ip mh-keystore)
    :8500" \
58             --engine-opt="cluster-advertise=eth1:2376" \
59             "sgx-csem-host-$HOST_ID"
60     fi
61     eval $(docker-machine env "sgx-csem-host-$HOST_ID")
62     docker load -i ~/tmp/consumer.tar
63     docker load -i ~/tmp/deployment.tar
64     docker load -i ~/tmp/fake-sensor.tar
65     docker load -i ~/tmp/mqtt-sub.tar
66     docker load -i ~/tmp/producer.tar
67 done
```

```
68
69 # We now create the overlay network
70 eval $(docker-machine env --swarm sgx-csem-host-1)
71 docker network create --driver overlay --subnet=10.0.0.0/16 sgx-csem-net
72
73 for CL_ID in $(seq 1 ${NUM_CLIENTS});
74 do
75     # Prepare environment
76     H_ID=$(( $(( $CL_ID / $MAX_CONTAINERS )) + 1))
77     mkdir -p ${LOCAL_DATA_DIR}${CL_ID}
78     mkdir -p ${LOCAL_RESULT_DIR}${CL_ID}
79     eval $(docker-machine env "sgx-csem-host-$H_ID")
80     docker-machine ssh "sgx-csem-host-$H_ID" mkdir -p "results/${CL_ID}"
81     docker-machine mount "sgx-csem-host-$H_ID":"/home/docker/results/$CL_ID" \
82         ${LOCAL_RESULT_DIR}${CL_ID}
83     docker-machine ssh "sgx-csem-host-$H_ID" mkdir -p "data/${CL_ID}"
84     docker-machine mount "sgx-csem-host-$H_ID":"/home/docker/data/$CL_ID" \
85         ${LOCAL_DATA_DIR}${CL_ID}
86
87     # Deploy containers (Add a -d at the end for detach mode)
88     cat ${COMPOSE_FILE} | envsubst | docker-compose -f - -p \
89         "${PROJECT_NAME}-${CL_ID}" up &
90     sleep 5
91 done
92
93 # Return to default environment before exiting
94 eval $(docker-machine env -u)
```

Listing 5.7: Client Cluster Deployment Script.

5.3.3 Deployment

The main entry point for the platform's deployment is included in Listing 5.8. It first loads a set of environment variables and parses the input (lines 6,8-12), it then deploys the client cluster as detailed in §5.3.2, and it lastly deploys the server container as detailed in §5.3.1.

```
1 #!/bin/bash
2
3 set -a
4
5 # Predefined enviroment variables
6 source execution.env
7
8 ALGORITHM=${1}
```

```
SGX_MODE=${2}
SAMPLE_RATE=${3}
NUM_CLIENTS=${4}
FILE_LINES=${5}

# Deploy Client-Side container
cd ${CLIENT_DIR}
bash ${CLIENT_DIR}/deploy-client.sh ${SAMPLE_RATE} ${NUM_CLIENTS} ${FILE_LINES}
cd ${WDIR}

# Deploy Server-Side Containers
cat ${COMPOSE_FILE} | envsubst | docker-compose -f - -p "${PROJECT_NAME}" up &
```

Listing 5.8: Main entry point for a single execution.

Additionally, a PYTHON script wraps the launcher presented in Listing 5.8 in order to conveniently orchestrate execution batches for evaluation purposes. The same script (attached in Listing A.10) also includes the queries to obtain the required metrics for the different benchmarking scenarios that we will present in Chapter 6.

Chapter 6

Evaluation

The aim of this Chapter is to present the experimental evaluation of our system. Firstly, in §6.1 we introduce the hardware used for the benchmarking. We differentiate between the machines and software used in the Server-side §6.1.1 and those used in the Client-side §6.1.2 since both components have very different requirements. Secondly, we cover the different experimental configurations considered in §6.2. The latter require a given number of metrics that we present in §6.3. The workloads used to assess the performance are presented and justified in §6.4. Lastly, our results are presented and analyzed in §6.5. In a nutshell, our experiments answer the following questions: *(i)* is the design of the platform sound? *(ii)* is our implementation efficient, *(iii)* what is the overhead of SGX, and *(iv)* is it scalable?

6.1 Hardware Settings

6.1.1 Server

The server side is where the filesystem interface is hosted and where the SGX-Spark jobs run. Therefore, it must be be equipped with Intel SGX and enclave mode must be enabled. The component runs on a machine located at the University of Neuchâtel's (UniNe) cluster 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 [62], and SGX-LKL [26]. We use an internal release of the SGX-Spark framework, which is currently under development.

6.1.2 Client

For evaluation purposes, and as introduced in the Implementation Chapter (5), the whole set of clients is deployed as a standalone Docker swarm in a single computing instance. The machine we are using is located in UniNe's cluster and is equipped with two AMD EPYC 7281 16-Core Processor which, taking hyperthreading into consideration, account for a total of 64 cores and 64

GiB RAM. It is deployed with 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 (we rely on Consul [61]). Inter-container communication is established using the `overlay` network driver. To optimize start up time we provide all the images via `.tar` files that are loaded to the VM, skipping image building time, and we use a local copy of `virtualbox`'s disk image. In order to gather the results, we mount a Docker volume on each client and mirror the VM to the real filesystem using *Secure SHell File System (SSHFS)*. 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.

6.2 Experimental Configuration

In order to evaluate the overhead SGX-Spark and our system introduce, we compare three different settings or execution modes:

1. The **vanilla Spark** mode acts as our baseline. It executes the algorithm using a standard distribution of the Spark cluster-computing framework.

2. The **SGX-Spark w/o Enclaves** mode is an SGX-Spark execution but with the enclaves disabled. This is, the engine sets up the collaborative JVM scheme communicating over SHM, but neither runs inside an enclave.

3. The **SGX-Spark w/ Enclaves** mode is the execution mode of our system. It runs unmodified Spark applications leveraging SGX for sensitive computations.

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. We increment the input workload (see §6.4) and measure the impact it has on the execution time (see §6.3). In batch mode, a result file is generated once the processing is finished. 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 minutes of the service operating with a specific execution mode, client configuration, and input workload. We execute our experiments 5 times and report average values together with their standard deviations.

6.3 Analyzed Metrics

To assess performance, scalability, and efficiency, we consider two different metrics depending on whether we run in *batch* or *stream* mode: **elapsed time** and **average batch processing time** [1].

1. **Elapsed Time** measures the time it takes the engine to process the input file. In short, it is our system's execution time in *batch* mode. To obtain this metric it is sufficient with introducing logs (measuring time) in the application code.

2. **Average Batch Processing Time** takes the average of the time it takes the platform to process each 10-second-long chunk of the input data stream. To obtain this value we query Spark's REST API [63]. We use Python's `requests` package as depicted in Listing 6.1. Since the server, the clients, and the deployment might be hosted in different locations, the easiest way to query the REST API instantiated by the master process at the server is to establish a SSH tunnel and forward the contents in the server's 4040 port to `localhost`. This is done in line 9 using an additional Python script that we attach in Listing B.1 (see also Listing B.2). Note that in the latter there are configuration-specific parameters. Once the port forwarding is established, we set up and execute the request (lines 11-14) and then process the results (lines 16-20). It is worth mentioning that, since Spark keeps an historic of the batch processing times, it is sufficient with doing only one query right before we terminate the 5-minute-long execution.

```
1  def query_batch_processing(filedir):
2      """Query Spark's REST API
3
4      This method queries sparks API for the batch processing time. It firsts
5      launches a port forwarding daemon in order to be able to perform the
6      request and then queries the information. Note that this is done only
7      once at the end of the execution (right before killing it).
8      """
9      proc = subprocess.Popen("python3 port_forward.py".split(" "))
10     time.sleep(2)
11     app_id = requests.get('http://localhost:4040/api/v1/applications/').json()
12     app_id = app_id[0]['id']
13     req = 'http://localhost:4040/api/v1/applications/{}/jobs/'.format(app_id)
14     r = requests.get(req).json()
15     time_format = "%Y-%m-%dT%H:%M:%S.%f"
16     times = [ datetime.strptime(r[i]['completionTime'][:-3],
17                                 time_format).timestamp() -
```

[1] 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 seconds in our experiments). The time it takes the engine to process the data therein contained is denoted as batch processing time.

```
18                datetime.strptime(r[i]['submissionTime'][:-3],
19                                  time_format).timestamp()
20                for i in reversed(range(len(r))) if 'completionTime' in r[i] ]
21    proc.kill()
22    filename = filedir + "batch_processing_times.csv"
23    with open(filename, "w+") as f:
24        for num, val in enumerate(times):
25            if num == 0:
26                f.write("{} {}".format(num, val))
27            else:
28                f.write("\n{} {}".format(num, val))
29    return 0
```

Listing 6.1: Snippet illustrating a query to Spark's REST API.

We study the variability of these two parameters as we modify the input workload according to §6.4.

6.4 Workload

As firstly introduced in Chapter §2.2, the clients inject streams as cardiac signals corresponding to RR intervals and their timestamps. Each client 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

Table 6.1: Different input loads used for Batch Mode (BM) and Streaming Mode (SM). 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 / sec)**	**Input Load**
BM - Small Load	$\{44, 89, 178, 356, 712, 1424\}$	$\{1, 2, 4, 8, 16, 32\}$ kB
SM - Small Load	$\{44, 89, 178, 356, 712, 1424\}$	$\{1, 2, 4, 8, 16, 32\}$ kB / sec
BM - Big Load	$\{44, 89, 178, 356, 712, 1424\} * 1024$	$\{1, 2, 4, 8, 16, 32\}$ MB
SM - Big Load	$\{44, 89, 178, 356, 712, 1424\} * 1024$	$\{1, 2, 4, 8, 16, 32\}$ MB / sec

possible bottlenecks, we scale up the output rate of these signals with the goal of inducing more aggressive workloads into the system. We do so in detriment of medical realism, since arbitrary input workloads do not relate to any medical situation or condition. Table 6.1 shows the variations used to evaluate the various execution modes. On the leftmost column, BM and SM stand for *batch* and *stream* mode execution. For each mode, we consider a small load and a big load evaluation setting. We present both the `sample_rate` parameter passed to the `sensor` component and the total workload that amounts to (in multiples of bytes per second).

6.5 Results

In this Section we present and analyze the evolution of the different metrics (§6.3) as we vary the input workload (§6.4) for the different execution modes (§6.2). We firstly present the results for *batch* mode in §6.5.1 and then we do the same for *stream* mode in §6.5.2.

6.5.1 Batch Execution

The exact configuration for the study of the elapsed time as we vary the input file size is: one client, one master, one driver, one worker, and a variable input file that progressively increases in size. We measure the elapsed time of each execution and present the average and standard deviation of a total of five experiments with the same configuration. Results obtained are included in Figure 6.1.

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 (and hence our system), 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 hundreds of concurrent clients (considering processing time as the only bottleneck).

6.5.2 Stream Execution

Similarly as done in §6.5.1, we scale the load of the data streams that feed the platform and study the evolution of the average batch processing time. 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 Figure 6.2.

We obtain results for vanilla Spark, and SGX-Spark without enclaves, and we estimate them for SGX-Spark with enclaves. We observe similar behaviours as those in Figure 6.1. 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 behaviour 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. This is due to the fact that the average of streaming batch processing times smoothens the initial overhead of starting the Spark engine. Furthermore, in *stream* mode, data loading times are hidden under previous batches' execution times, and, again, smoothened by the average.

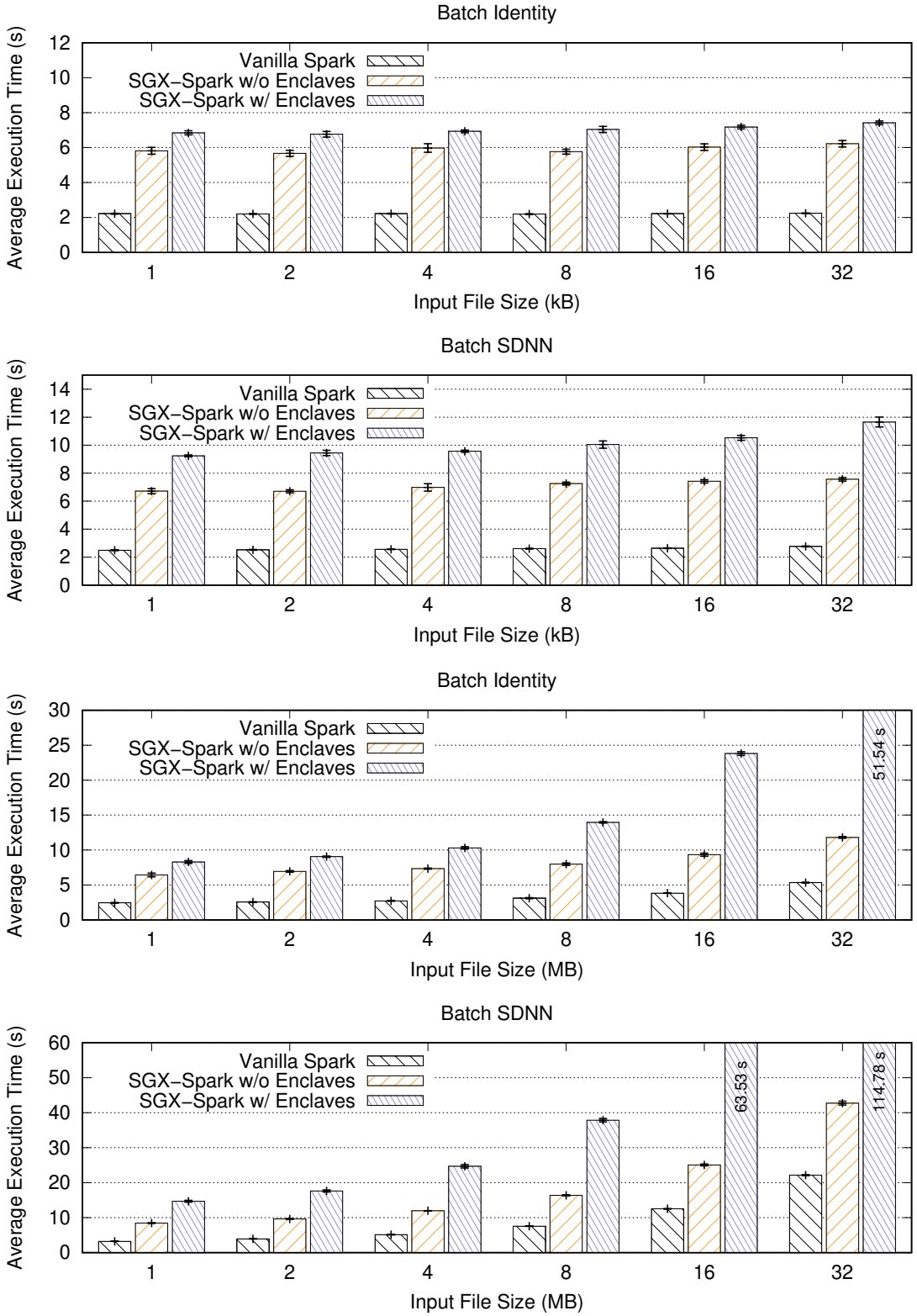

Figure 6.1: Evolution of the average elapsed time, together with its standard deviation, as we increase the size of the input file. We compare the three different execution modes for each algorithm. Mode SGX-Spark w/ enclaves is the mode the platform runs in.

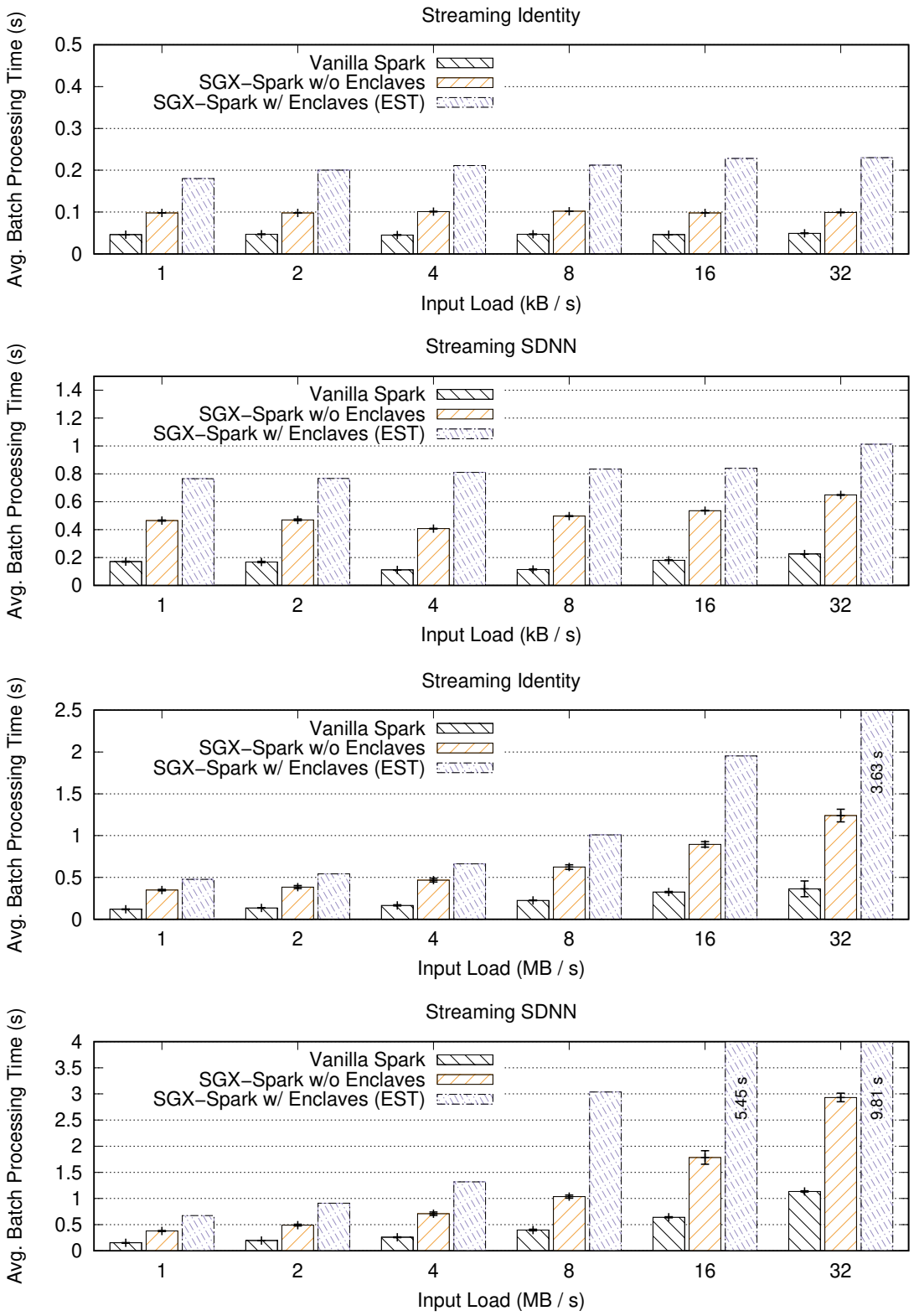

Figure 6.2: Evolution of the average batch processing time as we increase the input stream size. We compare the results of the three different execution modes. Note that those corresponding to SGX-Spark w/ enclaves are estimated basing on the results in Figure 6.1 and the slow-down with respect to the other execution modes.

Chapter 7

Future Work

There are several dimensions along which the current project can be improved. We differentiate between possible improvements to the current architecture as presented in Chapter 4, and further research lines to enrich the privacy-preserving computing state of the art.

With regard to the project here presented, there are different issues that would need to be addressed in the coming work. Firstly, the current SGX-Spark implementation, as mentioned in Chapter 6, does not yet have support for in-enclave streaming. As a consequence, the estimated results should be validated once the Streaming API is supported by SGX-Spark. Secondly, results concerning client scalability should also be included in the evaluation chapter. Following the deployment technique (standalone `Docker` cluster) indicated in §5.3, we have managed to run experiments with hundreds of clients. However, due to time constraints, we have not been able to provide an exhaustive assessment of the system's scalability, *i.e.* how many clients it can run in parallel, neither of how the overhead introduced by SGX-Spark might affect the number of clients our platform can handle simultaneously. On the same lines, we would like to study the cost of deploying our system over public cloud infrastructures such as AWS Confidential Computing. Lastly, we intend to deploy the platform in a real use case, this is, using real data produced by real users and streamed through a smart gateway.

On a more general note, we think that our threat model, as presented in §4.3, could be very much improved by securing the client package. To do so, we envision the use of ARM TrustZone, widely available in low-power devices (*e.g.*, Raspberry PI), to reduce the TCB in the client-side of the architecture whilst still leveraging on Trusted Execution Environments. Finally, were TEEs to be used to secure the client package, a point-to-point, TEE-to-TEE, communication could be leveraged instead of relying on standard secure transfer protocols. This way, data would never leave the TEE (to be included in a SFTP package and sent over the network), hence reducing the overall attack surface. Even though generic TEE-to-TEE point-to-point communication protocols might not be mature enough yet (for generic TEEs on each endpoint), there are for instance solutions for enclave-to-enclave [42] secure link establishment.

Chapter 8

Conclusions

In this thesis, we have presented a proof of concept of a streaming platform that grants executions on remote, untrusted, servers or clouds with data and code confidentiality and integrity. We provide end-to-end protection transparently to the developer since we run unmodified Apache Spark applications inside Intel SGX's enclaves.

Our design easily scales to different types of data generators, data streams, or even processing algorithms. It only relies on SGX-Spark, a stream processing framework. However, this dependency could also be overlooked since the server component in our architecture is also pluggable and modular.

We have quantified the impact on overall system performance when protecting health sensitive data from an untrusted cloud provider. More precisely, when performing an HRV analysis, for files smaller than 4 MB, it introduces a x4-5 slow-down when compared to vanilla Apache Spark both in batch and streaming execution mode. A good part of this slow-down though, is due to the collaborative JVM structure adopted in SGX-Spark. For a matter of fact, a slow-down of only x1.5-2 is introduced when moving from SGX-Spark with enclaves disabled to enclave-enabled mode. Regardless, these results are already competitive when compared to other S.o.A privacy-preserving processing engines [40]. Therefore, we consider our platform to be mature enough to be introduced in a production environment, since it complies with current data protection regulations whilst still maintaining a reasonable performance, and keeping the costs to use the cloud infrastructure reasonable. Lastly, the work here presented is a work in progress for there are still several issues to address that could further improve the overall performance and the overall security of the platform, see Chapter 7.

To put it in a nutshell, this thesis covers the motivation, design, implementation, and analysis of a proof of concept of a privacy-preserving streaming platform. The quality of the results here obtained is backed by the publication of two conference papers [1, 2].

Bibliography

[1] C. Segarra, E. Muntané, M. Lemay, V. Schiavoni, and R. Delgado-Gonzalo. Secure stream processing for medical data. In *41st IEEE Engineering in Medicine and Biology Conference (EMBC '19)*, 2019.

[2] C. Segarra, R. Delgado-Gonzalo, M. Lemay, P. Aublin, P. Pietzuch, and V Schiavoni. Using trusted execution environments for secure stream processing of medical data. In *19th International Conference on Distributed Applications and Interoperable Systems (DAIS '19)*, 2019.

[3] Gartner. Leading the IoT Gartner Insights on how to lead in a connected world. 2017.

[4] M. Barbosa, S. B. Mokhtar, P. Felber, F. Maia, M. Matos, R. Oliveira, E. Riviere, V. Schiavoni, and S. Voulgaris. SAFETHINGS: Data security by design in the IoT. In *IEEE EDCC'17*.

[5] G. P. Cumming. Connecting & collaborating - Healthcare for the 21st century. In *Proceedings of the Second European Workshop on Practical Aspects of Health Informatics [PAHI], Trondheim, Norway*, 2014.

[6] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT'99*.

[7] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. In *CRYPTO'12*.

[8] Ch. Göttel, R. Pires, I. Rocha, S. Vaucher, P. Felber, M. Pasin, and V. Schiavoni. Security, performance and energy trade-offs of hardware-assisted memory protection mechanisms. In *IEEE SRDS'18*.

[9] V. Costan and S. Devadas. Intel SGX explained. *IACR'16*.

[10] ARM TrustZone Developer. `https://developer.arm.com/technologies/trustzone`.

[11] Barb Darrow. Google Is First in Line to Get Intel's Next-Gen Server Chip. `http://for.tn/2lLdUtD`, February 2017. Accessed on: 2018-03-05.

[12] Coming Soon: Amazon EC2 C5 Instances, the next generation of Compute Optimized instances. `http://amzn.to/2nmIiH9`, November 2016. Accessed on: 2018-03-05.

[13] 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/`. Accessed: 2019-01-28.

[14] Mark Russinovich. Introducing Azure Confidential Computing. `https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/`, September 2017. Accessed on: 2018-03-05.

[15] A. Kumar, F. Shaik, B. A. Rahim, and D. S. Kumar. *Signal and image processing in medical applications*. Springer, 2016.

[16] Z. Xiong, M. Nash, E. Cheng, V. Fedorov, M. Stiles, and J. Zhao. ECG signal classification for the detection of cardiac arrhythmias using a convolutional recurrent neural network. *Physiological Measurement*, August 2018.

[17] J. Van Zaen, O. Chételat, M. Lemay, E. M. Calvo, and R. Delgado-Gonzalo. Classification of cardiac arrhythmias from single lead ECG with a convolutional recurrent neural network. In *Proceedings of the 12th International Joint Conference on Biomedical Engineering Systems and Technologies (BIOSTEC'19)*, 2019. In press.

[18] Apache Foundation. Spark streaming programming guide. `https://spark.apache.org/docs/2.2.0/streaming-programming-guide.html`. Accessed: 2019-01-15.

[19] Aurélien Havet, Rafael Pires, Pascal Felber, Marcelo Pasin, Romain Rouvoy, and Valerio Schiavoni. SecureStreams: A reactive middleware framework for secure data stream processing. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, pages 124–133. ACM, 2017.

[20] Imperial College London. Large-Scale Data & Sytems Group at the Imperial College London website. `https://lsds.doc.ic.ac.uk/`. Accessed: 2019-01-18.

[21] GlobalPlatform. Introduction to Trusted Execution Environments. Technical report, GlobalPlatform, 2018.

[22] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 10:1–10:1, New York, NY, USA, 2013. ACM.

[23] Sh. Gueron. A memory encryption engine suitable for general purpose processors. *IACR Cryptology ePrint Archive*, 2016:204, 2016.

[24] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[25] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *Presented as part of the*. USENIX, Submitted.

[26] SGX-LKL on Github. `https://github.com/lsds/sgx-lkl`. Accessed: 2019-01-15.

[27] D3.2 SecureCloud: Specification and Implementation of Reusable Secure Microservices. `https://www.securecloudproject.eu/wp-content/uploads/D3.2.pdf`, 2017.

[28] T. Tamura and W. Chen. *Seamless Healthcare Monitoring: Advancements in Wearable, Attachable, and Invisible Devices*. Springer, 2018.

[29] J. Parak, A. Tarniceriu, Ph. Renevey, M. Bertschi, R. Delgado-Gonzalo, and I. Korhonen. Evaluation of the beat-to-beat detection accuracy of PulseOn wearable optical heart rate monitor. In *Proceedings of the 37th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC'15)*, pages 8099–8102, 2015.

[30] K. Tehrani and M Andrew. Wearable technologies and wearable devices: Everything you need to know. `http://www.wearabledevices.com/what-is-a-wearable-device/`. Accessed: 2019-04-25.

[31] L. S. Lilly. *Pathophysiology of heart disease: A collaborative project of medical students and faculty*. Lippincott Williams & Wilkins, 2001.

[32] A. Camm, M. Malik, J. Bigger, G. Breithardt, S. Cerutti, R. Cohen, Ph. Coumel, E. Fallen, H. Kennedy, and R. E. Kleiger. Heart rate variability: Standards of measurement, physiological interpretation and clinical use. Task Force of the European Society of Cardiology and the North American Society of Pacing and Electrophysiology. *Circulation*, 93(5):1043–1065, 1996.

[33] R. E. Kleiger, J. P. Miller, J. Th. Bigger, and A. J. Moss. Decreased heart rate variability and its association with increased mortality after acute myocardial infarction. *The American Journal of Cardiology*, 59(4):256–262, 1987.

[34] J. Th. Bigger, J. L. Fleiss, R. C. Steinman, L. M. Rolnitzky, R. E. Kleiger, and J. N. Rottman. Frequency domain measures of heart period variability and mortality after myocardial infarction. *Circulation*, 85:164–171, February 1992.

[35] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter Pietzuch. SABER: Window-Based Hybrid Stream Processing for Heterogeneous Architectures. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 555–569, New York, NY, USA, 2016. ACM.

[36] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S McKinley, and Felix Xiaozhu Lin. StreamBox: Modern stream processing on a multicore machine. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 617–629, 2017.

[37] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 374–389. ACM, 2017.

[38] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational data processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.

[39] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. Structured streaming: A seclarative API for real-time applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 601–613, New York, NY, USA, 2018. ACM.

[40] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 283–298, Berkeley, CA, USA, 2017. USENIX Association.

[41] Deepak Puthal, Surya Nepal, Rajiv Ranjan, and Jinjun Chen. DPBSV – An Efficient and Secure Scheme for Big Sensing Data Stream. In *Proceedings of the 2015 IEEE Trustcom/BigDataSE/ISPA - Volume 01*, TRUSTCOM '15, pages 246–253, Washington, DC, USA, 2015. IEEE Computer Society.

[42] Pierre-Louis Aublin, Florian Kelbert, Dan O 'keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch. TaLoS: Secure and transparent tls termination inside sgx enclaves. Technical report, Imperial College London, 05 2017.

[43] T. Dierks and C. Allen. The tls protocol version 1.2, 2008.

[44] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.

[45] Sai Deep Tetali, Mohsen Lesani, Rupak Majumdar, and Todd Millstein. MrCrypt: Static Analysis for Secure Cloud Computations. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 271–286, New York, NY, USA, 2013. ACM.

[46] Julian James Stephen, Savvas Savvides, Vinaitheerthan Sundaram, Masoud Saeida Ardekani, and Patrick Eugster. STYX: Stream Processing with Trustworthy Cloud-based Execution. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 348–360, New York, NY, USA, 2016. ACM.

[47] Christian Göttel, Rafael Pires, Isabelly Rocha, Sebastien Vaucher, Pascal Felber, Marcelo Pasin, and Valerio Schiavoni. Security, performance and energy trade-offs of hardware-assisted memory protection mechanisms. In *Proceedings of the 37th IEEE Symposium on Reliable Distributed Systems (SRDS'18)*, pages 133–142, 2018.

[48] P. Renevey, R. Delgado-Gonzalo, A. Lemkaddem, C. Verjus, S. Combertaldi, B. Rasch, B. Leeners, F. Dammeier, and F. Kübler. Respiratory and cardiac monitoring at night using a wrist wearable optical system. In *Proceedings of the 40th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC'18)*, pages 2861–2864, July 2018.

[49] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using sgx to conceal cache attacks. In Michalis Polychronakis and Michael Meier, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24, Cham, 2017. Springer International Publishing.

[50] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, Urs Müller, and Ahmad-Reza Sadeghi. Dr.sgx: Hardening sgx enclaves against cache attacks with data location randomization. 09 2017.

[51] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, pages 217–233, Berkeley, CA, USA, 2017. USENIX Association.

[52] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[53] Spectre Attack SGX on Github. `https://github.com/lsds/spectre-attack-sgx`. Accessed: 2019-01-16.

[54] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018.

[55] F. Shaffer and J. P. Ginsberg. An overview of heart rate variability metrics and norms. *Frontiers in Public Health*, 5:258, 2017.

[56] The Scala Programming Language. `https://www.scala-lang.org/`. Accessed: 2019-02-04.

[57] RDD Programming Guide. `https://spark.apache.org/docs/latest/rdd-programming-guide.html`. Accessed: 2019-02-04.

[58] MQTT Communication Protocol. `http://mqtt.org/`. Accessed: 2019-02-04.

[59] Eclipse Paho MQTT Implementation. `https://www.eclipse.org/paho/`. Accessed: 2019-02-04.

[60] Eclipse Mosquitto Image on Docker Hub. `https://hub.docker.com/_/eclipse-mosquitto/`. Accessed: 2019-01-16.

[61] Consul Image on Docker Hub. `https://hub.docker.com/_/consul/`. Accessed: 2019-01-16.

[62] Intel Software Guard Extension for Linux OS Driver on GitHub. `https://github.com/intel/linux-sgx-driver`. Accessed: 2019-02-05.

[63] Spark Documentation: REST API. `https://spark.apache.org/docs/latest/monitoring.html#rest-api`. Accessed: 2019-02-05.

Appendix A

Implementation Code Snippets

A.1 Server-Side Algorithms

```
/** Estimation of the Standard Deviation between NN intervals.
 *
 *  @constructor create a new estimation suite.
 *  @param wdwSize duration in seconds of the window in which we compute
 *  the standard deviation.
 */
class EstimateSDNN(wdwSize: Int) {

  /** Estimation of the SDNN with a fixed window in streaming mode
   *
   *  @return returns a DStream containing a tuple per line containing the time
   *  block/interval (starting from zero) and the SDNN value.
   */
  def sdnn(dataStream: DStream[String]): DStream[(String, Double)] = {
    val operatorByKey = new StreamOperators(wdwSize)
    operatorByKey.sumSquareByKey(dataStream)
      .join(operatorByKey.cardinalByKey(dataStream))
      .map(udfDivide)
      .join(operatorByKey.avgByKey(dataStream))
      .map(udfSqrtDif)
      .map(udfTimesWindow)
  }

}
```

Listing A.1: Implementation of the `SDNN` algorithm.

```
package org.apache.spark.csem.utils
```

```
2
3  import org.apache.spark.rdd.RDD
4  import org.apache.spark.streaming.dstream.DStream
5  import org.apache.spark.csem.utils.StreamOperators
6
7  import scala.math.{pow, cos, sin, Pi, floor}
8
9  /** Set of signal processing operators defined on (K, V), (String, Float)
10  *  DStreams. Currently the methods included in this suite are: DFT,
11  *
12  *  @constructor create a new operator suite.
13  *  @param wdwSize duration in seconds of the window in which we compute
14  *  the standard deviation.
15  */
16 class StreamSignalProcessing(wdwSize: Int) extends Serializable {
17
18   /** Preprocesses the input data stream (which contains lines from the csv
19    *  input file) to the following format, (k, v) where:
20    *  - k: is the 10 second block identifier (key)
21    *  - v: iterable with all the rr's from that block indexed.
22    *
23    *  @return A data stream with the specified format.
24    */
25   def preProcess(dataStream: DStream[String]):
26     DStream[(String, Iterable[(Double, Int)], Int)] = {
27       dataStream
28         .map(line => (floor(line.split(",")(0).toDouble/10).toInt.toString,
29           line.split(",")(1).toDouble))
30         .groupByKey()
31         .map(line => (line._1, line._2.zipWithIndex, line._2.size))
32   }
33
34   /** Compute the squared module of the Discrete Fourier Transform of the
35    *  samples contained in the DStream.
36    *
37    *  @return returns a DStream containing a tuple per line containing the time
38    *  block/interval (starting from zero) and the sum of all the values
39    *  in that same interval.
40    */
41   def modDFT2(dataStream: DStream[String]):
42     DStream[(String, Double, Double, Double)] = {
43
44     // Auxiliary methods to compute the squared modulus of the DFT
45     def cosDFT(line: (String, Iterable[(Double, Int)], Int), f_index: Int):
46       Double = {
```

```
47        line._2
48          .map(x => x._1*cos(2*Pi*x._2*f_index/line._3))
49          .reduce((a,b) => (a+b))
50      }
51      def sinDFT(line: (String, Iterable[(Double, Int)], Int), f_index: Int):
52        Double = {
53        line._2
54          .map(x => x._1*sin(2*Pi*x._2*f_index/line._3))
55          .reduce((a,b) => (a+b))
56      }
57
58      // Compute the Square Modulus of the DFT
59      def modDFT(value: (Double, Int),
60        line: (String, Iterable[(Double, Int)], Int)): (Double, Int) = {
61          (pow(cosDFT(line, value._2), 2) + pow(sinDFT(line, value._2), 2),
62            value._2)
63      }
64
65      // Predicate to know if a value belongs to the High Freq. Component
66      def isHighFreq(value: (Double, Int), cardinal: Int): Boolean = {
67        val thr_low = (0.15 * cardinal).toInt
68        val thr_high = (0.4 * cardinal).toInt
69        value._2 match {
70          case x if thr_low until thr_high contains x => true
71          case _ => false
72        }
73      }
74
75      // Predicate to know if a value belongs to the Low Freq. Component
76      def isLowFreq(value: (Double, Int), cardinal: Int): Boolean = {
77        val thr_low = (0.04 * cardinal).toInt
78        val thr_high = (0.15 * cardinal).toInt
79        value._2 match {
80          case x if thr_low until thr_high contains x => true
81          case _ => false
82        }
83      }
84
85      val tmp = preProcess(dataStream)
86        .map( l => (l._1, l._2.map( list => modDFT(list, l) ), l._3) )
87
88      // High Frequency Component
89      val highFreq = tmp
90        .map(line => (line._1, line._2
91          .filter( x => isHighFreq(x, line._3) )
```

```
92          .map( x => x._1 )
93          .foldLeft(0.toDouble)( (a,b) => (a + b) ), line._3))
94        .map( line => (line._1, line._2 / pow(line._3, 2) * 2) )
95
96      // Low Frequency Component
97      val lowFreq = tmp
98        .map(line => (line._1, line._2
99          .filter(x => isLowFreq(x, line._3))
100         .map( x => x._1 )
101         .foldLeft(0.toDouble)( (a,b) => (a + b) ), line._3))
102       .map( line => (line._1, line._2 / pow(line._3, 2) * 2) )
103     lowFreq
104       .join(highFreq)
105       .map( x => (x._1, x._2._1, x._2._2, x._2._1 / x._2._2) )
106   }
107 }
108
109
110 /** Estimation of the frequency bands given a HRV sequence.
111  *
112  *  @constructor create a new estimation suite.
113  *  @param wdwSize duration in seconds of the window in which we estimate
114  *  the frequency bands.
115  */
116 class EstimateHRVBands(wdwSize: Int) {
117   /** Estimation of the SDNN with a fixed window in streaming mode
118    *
119    *  @return returns a DStream containing a tuple per line containing the time
120    *  block/interval (starting from zero) and the SDNN value.
121    */
122   def estimate(dataStream: DStream[String]):
123       DStream[(String, Double, Double, Double)] = {
124     val processingByKey = new StreamSignalProcessing(wdwSize)
125     processingByKey.modDFT2(dataStream)
126   }
127 }
```

Listing A.2: Implementation of the `HRVBands` algorithm.

```
1 /** Main application/class that launches the identity application.
                                    [64/180]
2  *
3  *  @note We implement the identity application that
4  *  basically does nothing. This way we will be able to measure easily the
```

```
 *  system's overhead.
 */
object Identity extends Logging {

  def main(args: Array[String]) {

    // Initialize Variables and Spark Context
    val numClients = args(0).toInt
    val conf = new SparkConf().setAppName("HRV Toolbox - Identity")
    val sc = new SparkContext(conf)
    val ssc = new StreamingContext(sc, Duration(10000))

    // Map Output to Input
    val dataStreamVec = for (i <- 1 to numClients) yield ssc.textFileStream("csem/
   src/main/resources/csv/"+i.toString+"/
")
    for (i <- 1 to numClients) {
    dataStreamVec(i-1).saveAsTextFiles("csem/src/main/resources/results/" + i.
   toString + "/identity")
    }

    ssc.start()
    ssc.awaitTermination()
  }
}
```

Listing A.3: Implementation of the `Identity` algorithm.

A.2 Client-Side Services Source Code

```
"""Fake MQTT Generator

This module emulates a sensor that publishes data to a MQTT topic with a given
sample rate.
"""
// Method names are those required to be overridden by Paho MQTT
import paho.mqtt.publish as publish
import signal
import sys
import time
from datetime import datetime
from numpy import random

```

```
14
15 // SIGTERM Signal Handler
16 class Killer:
17   kill_now = False
18   def __init__(self):
19     signal.signal(signal.SIGTERM, self.exit_gracefully)
20
21   def exit_gracefully(self, signum, frame):
22     self.kill_now = True
23
24
25 // Callback to be executed at service start up time
26 def run(sample_rate):
27     if sample_rate == 0:
28         raise ZeroDivisionError("Can not set a 0 sample rate")
29     loop(int(sample_rate))
30
31
32 def loop(sample_rate):
33     """Endless Sample Generation
34
35     This module is the endless loop that generates <sample_rate>
36     new samples per second and publishes them to a MQTT topic.
37
38     Parameters
39     ----------
40     sample_rate : float
41         How many samples are generated per second
42     """
43     killer = Killer()
44     global CL_ID
45     past_ts = time.time()
46     while 1:
47         value = ""
48         # First we generate <sample_rate> timestamps in one second
49         tstamps = [i for i in random.uniform(low=time.time(),
50                                              high=time.time() + 1,
51                                              size=(sample_rate))]
52         tstamps.sort()
53         # Second we compute the intervals between them
54         for i in tstamps:
55             tmp = datetime.fromtimestamp(i).strftime('%Y-%m-%d %H:%M:%S')
56             value += tmp+','+str(int((i - past_ts) * 1e6))+'\n'
57             past_ts = i
58         publish.single("artificial-data-{}".format(CL_ID), value)
```

```
        time.sleep(1)
        if killer.kill_now:
            print("Exitting gracefully!")
            break


if __name__ == "__main__":
    """Standalone entry point"""
    if len(sys.argv) < 2:
        raise TypeError("No Sample Rate provided")
    CL_ID = sys.argv[2]
    run(sys.argv[1])
```

Listing A.4: Implementation of the `sensor` service.

```
"""MQTT Subscriber and CSV Generator

This module subscribes to the data topic in a MQTT queue and, whenever it
stacks a given amount of samples it generates a new csv file.

Attributes
----------
count : int
    Global counter.
curr_str : str
    Global string accumulator.
"""
import paho.mqtt.client as mqtt
from datetime import datetime
import time
import signal
import sys
import os


count = 0
curr_str = ""


class Killer:
  kill_now = False
  def __init__(self):
    signal.signal(signal.SIGTERM, self.exit_gracefully)

```

```
30    def exit_gracefully(self, signum, frame):
31        self.kill_now = True
32
33
34    def on_message(mosq, obj, msg):
35        """Callback method for the mosquitto client.
36
37        Whenever the MQTT client receives a new sample it triggers this method.
38        When the current count reaches the threshold, a new csv file is generated
39        and stored in the local data directory.
40
41        Parameters
42        ----------
43        Parameters are those of the standard implementation of the MQTT client
44        for Python.
45        """
46        global count
47        global curr_str
48        global FILE_LINES
49        global CL_ID
50        global LOCAL_DATA_DIR
51        global killer
52        print("Received message")
53        dcd_msg = msg.payload.decode("utf-8").split("\n")
54        whole_msg = [i.split(",") for i in dcd_msg if i != ""]
55        for tmp in whole_msg:
56            d = datetime.strptime(tmp[0], '%Y-%m-%d %H:%M:%S')
57            time_stamp = time.mktime(d.timetuple())
58            rr = tmp[1]
59            if count == FILE_LINES:
60                global TS
61                print("It took: {} s".format(time.time() - TS))
62                curr_str += "{},{}".format(time_stamp, rr)
63                ts = int(time.time()*1e3)
64                filename = "{}{}_".format(LOCAL_DATA_DIR, CL_ID)+str(ts)+".csv"
65                try:
66                    with open(filename, "w+") as f:
67                        f.write(curr_str)
68                except FileNotFoundError as e:
69                    if killer.kill_now:
70                        mosq.disconnect()
71                        print("Exitting gracefully!")
72                        break
73                count = 0
74                curr_str = ""
```

```
75          else:
76              curr_str += "{},{}\n".format(time_stamp, rr)
77              count += 1
78      else:
79          if killer.kill_now:
80              mosq.disconnect()
81              print("Exitting gracefully!")
82
83
84  def start_mqtt():
85      """Start the MQTT client"""
86      global CL_ID
87      mqttc = mqtt.Client()
88      mqttc.on_message = on_message
89      mqttc.connect("localhost", port=1883)
90      mqttc.subscribe("artificial-data-{}".format(CL_ID))
91      mqttc.loop_forever()
92
93
94  if __name__ == "__main__":
95      """Standalone execution entry point"""
96      try:
97          FILE_LINES = int(sys.argv[1])
98      except IndexError as e:
99          print("Have not provided a FILE_LINES value!")
100         sys.exit(1)
101     killer = Killer()
102     CL_ID = sys.argv[2]
103     LOCAL_DATA_DIR = sys.argv[3]
104     start_mqtt()
105     TS = time.time()
```

Listing A.5: Implementation of the `mqtt-subscriber` service.

```
1  """CSV Producer
2
3  This module monitors a local data directory and perodically sends the newly
4  added csv files to a remote directory in order to be processed.
5
6  Attributes
7  ----------
8  """
9  import os
10 import time
```

```
import signal
import sys
from paramiko import SSHClient, SSHConfig, ProxyCommand, SFTPClient
from paramiko.util import log_to_file
from server_connect import server_connect


class Killer:
  kill_now = False
  def __init__(self):
    signal.signal(signal.SIGTERM, self.exit_gracefully)

  def exit_gracefully(self, signum, frame):
    self.kill_now = True


def monitor(ssh, sftp, *args):
    """Sending daemon

    This method starts an infinite loop that every <SEND_PERIOD> looks for
    newly added data files and copies them to a remote directory.

    Parameters
    ----------
    ssh : paramiko.SSHClient
        SSH Client connected to the remote host.
    sftp : paramiko.SFTPClient
        SFTP Client connected to the remote host.
    """
    global LOCAL_DATA_DIR
    path_to_watch = LOCAL_DATA_DIR
    before = dict([(f, None) for f in os.listdir(path_to_watch)])
    global SEND_PERIOD
    killer = Killer()
    while 1:
        time.sleep(SEND_PERIOD)
        after = dict([(f, None) for f in os.listdir(path_to_watch)])
        added = [f for f in after if f not in before]
        if added:
            for f in added:
                local_file = LOCAL_DATA_DIR + str(f)
                global REMOTE_DATA_DIR
                remote_file = REMOTE_DATA_DIR + "/" + str(f)
                sftp.put(local_file, remote_file)
                os.remove(local_file)
```

```
                # If there are a lot of files, not checking inside also leads
                # to error with code 137
                if killer.kill_now:
                    print("Exitting gracefully!")
                    break
        before = after
        if killer.kill_now:
            print("Exitting gracefully!")
            break
    for d in os.listdir(LOCAL_DATA_DIR):
        tmp_file = LOCAL_DATA_DIR + str(d)
        os.remove(tmp_file)


if __name__ == "__main__":
    """Standalone entry point

    The scripts is executed as a subprocess and as a consequence in standalone
    mode. This is done to prevent it from blocking the rest of the execution
    and to prevent from working with Python's threading library.
    """
    ssh, sftp = server_connect()
    LOCAL_DATA_DIR = sys.argv[1]
    REMOTE_DATA_DIR = sys.argv[2]
    SEND_PERIOD = int(sys.argv[3])
    sftp.mkdir(REMOTE_DATA_DIR)
    monitor(ssh, sftp)
```

Listing A.6: Implementation of the `producer` service.

```
"""Result Consumer Daemon

This module periodically fetches data from a remote directory containing the
results of the computation and copies them over SFTP to a local directory.

Attributes
----------
FETCH_PERIOD : int
    How often does the module fetch the newly generated directories.
"""
import os
import getpass
import sys
import time
```

```
15 import signal
16 from paramiko import SSHClient, SSHConfig, ProxyCommand, SFTPClient
17 from paramiko.util import log_to_file
18 from server_connect import server_connect
19
20
21 class Killer:
22   kill_now = False
23   def __init__(self):
24     signal.signal(signal.SIGTERM, self.exit_gracefully)
25
26   def exit_gracefully(self, signum, frame):
27     self.kill_now = True
28
29
30 def get_all(sftp, remote_dir, local_dir):
31     """Copy a whole directory
32
33     The aim of this module is to emulate the -r option in scp. Note that this
34     is not recursive and only works in a folder with a known and given
35     structure.
36
37     Parameters
38     ----------
39     sftp : paramiko.SFTPClient
40         SFTP Client connected to the remote host.
41     remote_dir : str
42         Path of the directory we are copying from.
43     local_dir : str
44         Path of the directory we are copying to.
45     """
46     for f in sftp.listdir(remote_dir):
47         remote_file = remote_dir + str(f)
48         if (not f.startswith('.')) and (not f.startswith('_')):
49             local_file = local_dir + str(f)
50             sftp.get(remote_file, local_file)
51
52
53 def monitor(ssh, sftp):
54     """Fetching daemon
55
56     This method starts an infinite loop that every <FETCH_PERIOD> looks for
57     newly added directories and copies them to a local result directory.
58
59     Parameters
```

```
60      ----------
61      ssh : paramiko.SSHClient
62          SSH Client connected to the remote host.
63      sftp : paramiko.SFTPClient
64          SFTP Client connected to the remote host.
65      """
66      global REMOTE_RESULT_DIR
67      global FETCH_PERIOD
68      global LOCAL_RESULT_DIR
69      #print("CONSUMER: user -> {}".format(getpass.getuser()))
70      path_to_watch = REMOTE_RESULT_DIR
71      before = dict([(f, None) for f in sftp.listdir(path_to_watch)])
72      killer = Killer()
73      while 1:
74          time.sleep(FETCH_PERIOD)
75          after = dict([(f, None) for f in sftp.listdir(path_to_watch)])
76          added = [f for f in after if f not in before]
77          if added:
78              for f in added:
79                  local_dir = LOCAL_RESULT_DIR + str(f) + "/"
80                  remote_dir = REMOTE_RESULT_DIR + "/" + str(f) + "/"
81                  try:
82                      os.mkdir(local_dir)
83                      get_all(sftp, remote_dir, local_dir)
84                  except FileExistsError as e:
85                      print("Fetching a file that already exists!")
86          before = after
87          if killer.kill_now:
88              print("Exitting gracefully!")
89              break
90      for d in os.listdir(LOCAL_RESULT_DIR):
91          tmp_dir = LOCAL_RESULT_DIR + d
92          for f in os.listdir(tmp_dir):
93              tmp_file = tmp_dir + "/" + f
94              os.remove(tmp_file)
95          os.rmdir(tmp_dir)
96
97
98  if __name__ == "__main__":
99      """Standalone entry point
100
101     The scripts is executed as a subprocess and as a consequence in standalone
102     mode. This is done to prevent it from blocking the rest of the execution
103     and to prevent myself from working with Python's threading library.
104     """
```

```
    ssh, sftp = server_connect()
    LOCAL_RESULT_DIR = sys.argv[1]
    REMOTE_RESULT_DIR = sys.argv[2]
    sftp.mkdir(REMOTE_RESULT_DIR)
    FETCH_PERIOD = int(sys.argv[3])
    monitor(ssh, sftp)
```

Listing A.7: Implementation of the `consumer` service.

A.3 Deployment Scripts

```
#!/usr/bin/python3
"""Remote Side Standalone Launcher

This module implements the remote-side, standalone, launcher.
"""
import os
import sys
import time
import signal
from server_connect import server_connect


class Killer:
  kill_now = False
  def __init__(self):
    signal.signal(signal.SIGTERM, self.exit_gracefully)

  def exit_gracefully(self, signum, frame):
    self.kill_now = True


def main(ssh, sftp):
    """Main execution method

    This method contains the main deployment of UC1. It basically starts each
    and every process, launches the benchmarking, and cleans everything when
    execution has finished.

    Notes
    -----
    All the sleeps are placed due to experienced errors race conditions. As a
    consequence their arguments are completely empirical.
```

```
33
34      Arguments
35      ---------
36      ssh : paramiko.SSHClient
37          SSH Client to the remote server.
38      sftp : paramiko.SFTPClient
39          SFTP Client to the remote server.
40      """
41      global ALGORITHM
42      global SGX_MODE
43      global NUM_CLIENTS
44      algo_name = "./launch-csem-{}.sh {}".format(ALGORITHM, NUM_CLIENTS)
45      if SGX_MODE:
46          script_list = ["./master.sh", "./worker.sh", "./worker-enclave.sh",
47                         "./driver-enclave.sh", algo_name]
48      else:
49          script_list = ["./master.sh", "./worker.sh", algo_name]
50      for script in script_list:
51          _in, _out, _err = ssh.exec_command("cd sgx-spark; {} &".format(script))
52          time.sleep(3)
53      tmp_command = "dstat -l --nocolor --noheaders 10 > cpu_load.csv"
54      _, _out, __ = ssh.exec_command("cd sgx-spark; {} &".format(tmp_command))
55      _out.readlines()
56      killer = Killer()
57      while 1:
58          time.sleep(1)
59          if killer.kill_now:
60              print("Killing Gracefully!")
61              break
62      for script in script_list:
63          _, _out, __ = ssh.exec_command("kill $(pgrep -f '{}')".format(script))
64          _out.readlines()
65      _, _out, __ = ssh.exec_command("kill $(pgrep -f 'java')")
66      _out.readlines()
67      _, _out, __ = ssh.exec_command("kill $(pgrep -f 'dstat')")
68      _out.readlines()
69      _, _out, __ = ssh.exec_command("rm /dev/shm/*")
70      _out.readlines()
71      return 0
72
73
74  if __name__ == "__main__":
75      """Entry point for standalone executions
76
77      This entry point contains the command line argument parsing.
```

```
    """
    ssh, sftp = server_connect("/home/user/logfile.log")
    ALGORITHM = sys.argv[1]
    SGX_MODE = int(sys.argv[2])
    NUM_CLIENTS = int(sys.argv[3])
    main(ssh, sftp)
```

Listing A.8: Server-Side Deployment Script.

```
version: '3'

services:
    consumer:
        build: ./services/consumer/
        container_name: "sgx-csem-client-consumer-${CL_ID}"
        image: consumer:latest
        command: "${CONTAINER_RESULT_DIR} ${REMOTE_RESULT_DIR}/${CL_ID} ${
    FETCH_PERIOD}"
        user: "${myUID}:${myGID}"
        volumes:
            - "/home/docker/results/${CL_ID}:${CONTAINER_RESULT_DIR}"
              #    - "${LOCAL_RESULT_DIR}${CL_ID}:${CONTAINER_RESULT_DIR}"
    producer:
        build: ./services/producer/
        container_name: "sgx-csem-client-producer-${CL_ID}"
        image: producer:latest
        command: "${CONTAINER_DATA_DIR} ${REMOTE_DATA_DIR}/${CL_ID} ${SEND_PERIOD}"
        volumes:
            - "/home/docker/data/${CL_ID}:${CONTAINER_DATA_DIR}"
              #- "${LOCAL_DATA_DIR}${CL_ID}:${CONTAINER_DATA_DIR}"
    mqtt-sub:
        build: ./services/mqtt_sub/
        container_name: "sgx-csem-client-mqtt-sub-${CL_ID}"
        image: mqtt-sub:latest
        network_mode: "sgx-csem-net"
        user: "${myUID}:${myGID}"
        depends_on:
            - "mqtt"
        volumes:
            - "/home/docker/data/${CL_ID}:${CONTAINER_DATA_DIR}"
              #- "${LOCAL_DATA_DIR}${CL_ID}:${CONTAINER_DATA_DIR}"
        command: "${FILE_LINES} ${CL_ID} ${CONTAINER_DATA_DIR}"
    fake-sensor:
        build: ./services/fake-sensor/
```

```
        container_name: "sgx-csem-client-fake-sensor-${CL_ID}"
        image: fake-sensor:latest
        network_mode: "sgx-csem-net"
        command: "${SAMPLE_RATE} ${CL_ID}"
        depends_on:
            - "mqtt"
    mqtt:
        image: eclipse-mosquitto
        container_name: "sgx-csem-mqtt-${CL_ID}"
        network_mode: "sgx-csem-net"
        logging:
            driver: none
```

Listing A.9: Client Docker Compose Script.

```
#!/usr/bin/python3
"""Benchmarking and Evaluation launcher.

This module includes all the diferent benchmarking scenarions considered in the
project. It also deploys all of them in a sequential fashion.
"""
import itertools
import sys
import os
import time
import subprocess
from datetime import datetime
import requests
import glob
import shutil
from server_connect import server_connect


def query_batch_processing(filedir):
    """Query Spark's REST API
    This method queries sparks API for the batch processing time. It firsts
    launches a port forwarding daemon in order to be able to perform the
    request and then queries the information. Note that this is done only
    once at the end of the execution (right before killing it).
    """
    proc = subprocess.Popen("python3 port_forward.py".split(" "))
    time.sleep(2)
    app_id = requests.get('http://localhost:4040/api/v1/applications/').json()
    app_id = app_id[0]['id']
```

```
30      req = 'http://localhost:4040/api/v1/applications/{}/jobs/'.format(app_id)
31      r = requests.get(req).json()
32      time_format = "%Y-%m-%dT%H:%M:%S.%f"
33      times = [ datetime.strptime(r[i]['completionTime'][:-3],
34                                  time_format).timestamp() -
35                datetime.strptime(r[i]['submissionTime'][:-3],
36                                  time_format).timestamp()
37                for i in reversed(range(len(r))) if 'completionTime' in r[i] ]
38      proc.kill()
39      filename = filedir + "batch_processing_times.csv"
40      with open(filename, "w+") as f:
41          for num, val in enumerate(times):
42              if num == 0:
43                  f.write("{} {}".format(num, val))
44              else:
45                  f.write("\n{} {}".format(num, val))
46      return 0
47
48
49  def query_energy(filedir, elapsed_time = 0, energy_ini = 0):
50      """ Query UniNe's Powe Control System
51
52      The idea is to compute consumed energy vs runtime so this are the values
53      that we will pack. For further reference below are the values to query
54      for other things.
55      pdu_val    |    variable measured
56         1       |    Voltage AC rms (V)
57         2       |    Current AC rms (A)
58         8       |    Total energy active (kWh)
59        10       |    Resettable energy (?) active (kWh)
60      DISCLAIMER: The current result is in kWh!
61      """
62      port_fw = "python3 port_forward.py"
63      port_fw += " -lp 8080 -ru powercontrol.maas -rp 80 -rh hoernli-2"
64      proc = subprocess.Popen("{}".format(port_fw).split(" "))
65      time.sleep(2)
66      req = 'http://localhost:8080/statusjsn.js?components=16384'
67      # In this case we choose value 8 -> Energy
68      r = requests.get(req).json()['sensor_values'][1]['values'][3][8]['v']
69      # Obtain teh shit
70      # subprocess.call("kill $(pgrep -f '{}')".format(port_fw), shell=True)
71      proc.kill()
72      if energy_ini:
73          filename = filedir + "energy_consumption.csv"
74          with open(filename, "w+") as f:
```

```
75              f.write("{} {}\n".format(elapsed_time, float(r) - energy_ini))
76      else:
77          return float(r)
78
79
80  def query_memory(filedir):
81      ssh, sftp = server_connect("./logfile.log")
82      local_file = filedir + "cpu_load.csv"
83      remote_file = "/home/ubuntu/sgx-spark/cpu_load.csv"
84      sftp.get(remote_file, local_file)
85      sftp.remove(remote_file)
86      ssh.close()
87      sftp.close()
88
89
90  # TODO: This will eventually have to change when we containerize the client
91  def query_active_clients(filedir, total_clients):
92      ssh, sftp = server_connect("./logfile.log")
93      filename = filedir + "clients.csv"
94      remote_dir = "/home/ubuntu/sgx-spark/csem/src/main/resources/csv/"
95      active_clients = len(set([l.split('_')[0] for l in sftp.listdir(remote_dir)]))
96      with open(filename, "w+") as f:
97          f.write("Active clients:\t {}\n".format(active_clients))
98          f.write("Total clients:\t {}\n".format(total_clients))
99      ssh.close()
100     sftp.close()
101
102
103 def main():
104     """Launcher.
105
106     This method defines all the diferent benchmarking environments, the
107     parameters they take and deploys each and every execution.
108     """
109     # General variables
110     #n_versions = 1
111     #sr = [1]
112     #batch_number = 200
113     #batch_duration = 10
114
115     # Evaluation Variables
116     # Test:
117     #n_clients = [100]
118     #algorithms = ["sdnn"]
119     #mode = [1]
```

```
120     # Client Scalability:
121     #n_clients = [1, 50, 250, 500]
122     #algorithms = ["sdnn", "identity", "hrvbands"]
123     #mode = [1, 0]
124     # Maximal Throughput Many Files
125     n_versions = 1
126     sr = [100, 200, 300, 400, 500]
127     batch_number = 20
128     batch_duration = 10
129     n_clients = [1]
130     algorithms = ["sdnn"]
131     mode = [0]
132 
133     for elem in itertools.product(*[algorithms, mode, sr]):
134         for version in range(n_versions):
135             for client in n_clients:
136                 print("Starting execution with the following parameters:")
137                 print("\t - Replica: {} of {}".format(version + 1, n_versions))
138                 print("\t - Number of clients: {}".format(client))
139                 print("\t - Algorithm: {}".format(elem[0]))
140                 print("\t - SGX Enabled: {}".format(elem[1]))
141                 print("\t - Sample Rate: {}".format(elem[2]))
142                 # The dynamic file size is set so that one file is generated
143                 # roughly every 10 seconds, so simply sample_rate * 10 lines.
144                 fl = elem[2] * 10
145                 subprocess.call("./deploy-single-evaluation.sh {} {} {} {} {}".
    format(
146                                 elem[0], elem[1], elem[2], client, fl).split(" "))
147                 tmp = "/{}_ncli_{}_sgx_{}_sr_{}_v{}/".format(elem[0], client,
148                                                              elem[1], elem[2],
149                                                              version)
150                 filedir = OUTPUT_DIR + tmp
151                 os.makedirs(filedir, exist_ok=True)
152                 try:
153                     energy_ini = query_energy(filedir)
154                     elapsed_time = batch_number*batch_duration
155                     time.sleep(elapsed_time)
156                     query_batch_processing(filedir)
157                     query_energy(filedir, elapsed_time, energy_ini)
158                     query_memory(filedir)
159                     query_active_clients(filedir, client)
160                 except Exception as e:
161                     print("An exception occured during the data grepping!")
162                     print(e)
163                 finally:
```

```
164                         subprocess.call("./kill-execution.sh")
165
166
167 if __name__ == "__main__":
168     OUTPUT_DIR = sys.argv[1]
169     main()
```

Listing A.10: Benchmarking and Experiment Deployment Script.

Appendix B

Evaluation Code Snippets

```
#!/usr/bin/python3
"""SSH Tunnel Daemon

This module starts a daemon that establishes a SSH tunnel from the local box
to the remote one. This is done to be able to query Spark's UI. It is
essentially a local port forwarding.
"""
import os
import sys
import argparse
import socket
from paramiko import SSHClient, SSHConfig, ProxyCommand
from paramiko.util import log_to_file
from forward import forward_tunnel


def run(**kwargs):
    """Main method

    Starts a subprocess that SSH's to the remote host and establishes the
    tunnel/port forwarding. In short, you will see at localhost:local_port
    the contents found at remote_url:remote_port as seen by remote_host.

    Attributes
    ----------
    local_port : int
        Local port towards which the connection is forwarded.
    remote_url : str
        Remote url you are accessing from the remote host.
    remote_port : int
        Port you will be accessing the remote url through.
    remote_host : str
```

```
        Host through which you are establishing the tunnel. Note that this
        host should appear in your .ssh/config and, due to particularities
        of the implementation, it must include a ProxyCommand to be used.
    """
    ssh = SSHClient()
    ssh_config = SSHConfig()
    ssh.load_system_host_keys()
    with open(os.path.expanduser("~/.ssh/config")) as f:
        ssh_config.parse(f)
    try:
        details = ssh_config.lookup(kwargs['remote_host'])
    except Exception as e:
        logger.error("The remote host {} is not listed in your .ssh/config" +
                     " file!".format(kwargs['remote_host']))
    log_to_file("logfile.log")
    ssh.connect(hostname=details['hostname'], username=details['user'],
                sock=ProxyCommand(details['proxycommand']))
    forward_tunnel(kwargs['local_port'], kwargs['remote_url'],
                   kwargs['rem_port'], ssh.get_transport())


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("-lp", "--local_port", help="Local forwarded port",
                        type=int, nargs='?', default=4040, const=4040)
    parser.add_argument("-ru", "--remote_url", help="End URL (from remote)",
                        type=str, nargs='?', default='localhost',
                        const='localhost')
    parser.add_argument("-rp", "--rem_port", help="Remote used port", type=int,
                        nargs='?', default=4040, const=4040)
    parser.add_argument("-rh", "--remote_host", help="Remote host", type=str,
                        nargs='?', default="eiger-9", const="eiger-9")
    args = parser.parse_args()
    run(**vars(args))
```

Listing B.1: Python Script to Set Up a Port Forwarding Daemon.

```
#!/usr/bin/env python

# Copyright (C) 2003-2007  Robey Pointer <robeypointer@gmail.com>
#
# This file is part of paramiko.
#
# Paramiko is free software; you can redistribute it and/or modify it under the
```

```
# terms of the GNU Lesser General Public License as published by the Free
# Software Foundation; either version 2.1 of the License, or (at your option)
# any later version.
#
# Paramiko is distributed in the hope that it will be useful, but WITHOUT ANY
# WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR
# A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for more
# details.
#
# You should have received a copy of the GNU Lesser General Public License
# along with Paramiko; if not, write to the Free Software Foundation, Inc.,
# 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA.

"""
Sample script showing how to do local port forwarding over paramiko.

This script connects to the requested SSH server and sets up local port
forwarding (the openssh -L option) from a local port through a tunneled
connection to a destination reachable from the SSH server machine.
"""

import os
import socket
import select

try:
    import SocketServer
except ImportError:
    import socketserver as SocketServer

import sys
from optparse import OptionParser

import paramiko


SSH_PORT = 22
DEFAULT_PORT = 4000


class ForwardServer(SocketServer.ThreadingTCPServer):
    daemon_threads = True
    allow_reuse_address = True


```

```
class Handler(SocketServer.BaseRequestHandler):
    def handle(self):
        try:
            chan = self.ssh_transport.open_channel(
                "direct-tcpip",
                (self.chain_host, self.chain_port),
                self.request.getpeername(),
            )
        except Exception as e:
            return
        if chan is None:
            return

        while True:
            r, w, x = select.select([self.request, chan], [], [])
            if self.request in r:
                data = self.request.recv(1024)
                if len(data) == 0:
                    break
                chan.send(data)
            if chan in r:
                data = chan.recv(1024)
                if len(data) == 0:
                    break
                self.request.send(data)

        peername = self.request.getpeername()
        chan.close()
        self.request.close()


def forward_tunnel(local_port, remote_host, remote_port, transport):
    # this is a little convoluted, but lets me configure things for the Handler
    # object.  (SocketServer doesn't give Handlers any way to access the outer
    # server normally.)
    class SubHander(Handler):
        chain_host = remote_host
        chain_port = remote_port
        ssh_transport = transport

    ForwardServer(("", local_port), SubHander).serve_forever()
```

Listing B.2: Python Script to Set Up a SSH Tunnel.