Securing the Execution of ML Workflows across the Compute Continua

Francesc-Josep Lordan Gomis
francesc.lordan@bsc.es
Barcelona Supercomputing Center
Barcelona, Spain

André Martin
Technische Universität Dresden
Dresden, Germany
andre.martin@tu-dresden.de

Daniele Lezzi
daniele.lezzi@bsc.es
Barcelona Supercomputing Center
Barcelona, Spain

ABSTRACT

Cloud computing has become the major computational paradigm for the deployment of all kind of applications, ranging from mobile apps to complex AI algorithms. On the other side, the rapid growth of IoT market has led to the need of processing the data produced by smart devices using their embedded resources. The computing continuum paradigm aims at solving the issues related to the deployment of applications across edge-to-cloud cyber-infrastructures.

This work considers in-memory data protection to enhance security over the compute continua and proposes a solution for the development of distributed applications that handles security in a transparent way for the developer. The proposed framework has been evaluated using an ML application that classifies health data using a pre-trained model. The results show that securing in-memory data incurs no additional effort at development time and the overheads introduced by the encryption mechanisms do not compromise the scalability of the application.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

KEYWORDS

computing continuum, machine learning, security, encryption, distributed systems

ACM Reference Format:
Francesc-Josep Lordan Gomis, André Martin, and Daniele Lezzi. 2023. Securing the Execution of ML Workflows across the Compute Continua . In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23 Companion), April 15–19, 2023, Coimbra, Portugal.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3578245.3584727

1 INTRODUCTION

Artificial Intelligence (AI) and edge computing have emerged as major trends in the ICT industry. AI aims at providing machines with problem-solving and decision-making capabilities. Machine Learning (ML) is a sub-field within AI that develops algorithms through which systems can learn from examples and experience. Experience, which comes in the form of a huge amount of data commonly referred to as the training set, is thus exploited by learning algorithms to build mathematical models to accomplish specific tasks.

Some AI algorithms are resource-demanding; often, smart devices collecting data do not have enough resources to host an AI processing that ends up being offloaded onto the Cloud incurring a high-latency response. This latency does not hinder the training of the model, which often requires using distributed systems, nor the gathering of data composing the training set. Conversely, it drags down any attempt to take decisions on locally gathered data without the need for information to travel in a round trip to a remote data centre; performing AI inference at the edge enables low-latency use cases.

The evolution of Edge Computing has brought a new infrastructure paradigm: the Compute Continua, which covers the wide spectrum from hyper-local IoT sensors to Cloud Data Centers being aware of the many intermediate devices among them. The heterogeneity of software and hardware within the Continua is a challenge for researchers and practitioners that is overcome through software development solutions.

Multi-tenancy is an intrinsic characteristic of the Compute Continua. Generally, IoT devices are specific-purpose and serve a single user – the device owner–; however, the farther a device is from the IoT level, the more shared its resources are among several users. ML workflows often work with privacy-sensitive data such as images obtained from cameras, recordings from microphones, or personal data collected through activity trackers. Shared devices are a potential threat for this information; attackers – either third-party users deploying malicious software on the device or malicious administrators – could take benefit of any vulnerability on the device to access that information. However, not only vulnerabilities open the door for malicious user to get access to confidential data as any user with privileged access to the device can inspect other users data through techniques such as memory dumps. For this reason, software development solutions need to protect this data not only while in transit, often done through secure transport protocols such as HTTPS, but also protect data at rest as well as when in use. Data can be encrypted while stored on a disk. However, at operation time this data is loaded into memory opening a window for attackers to access it.

The work presented in this paper contributes to the current state-of-the-art by providing a software development solution with the capability to develop applications targeting the Compute Continua with the protection of in-memory data in a transparent manner for the developer. In particular, but without loss of generality, this work considers ML applications developed using the COMPSs

ICPE '23 Companion, April 15–19, 2023, Coimbra, Portugal
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0072-9/23/04…$15.00
https://doi.org/10.1145/3578245.3584727

framework: a programming model that aims at easing the development of distributed applications targeting the Compute Continua. For protecting application data, COMPSs' runtime system builds on SCONE, a framework that allows to transform native applications into confidential ones such that the application code runs in a trusted execution environment such as Intel SGX protecting sensitive data when in use during the training or inference. Besides, the article presents the results of the evaluation tests conducted using a prototype implementation that validates the viability of the proposal.

The article continues by introducing the two baseline technologies used in this paper. While Section 3 describes the COMPSs programming model and casts a glance over the architecture and operation of its runtime system, Section 4 does the same for the security tool: SCONE. Section 5 discusses the different decisions taken during the design of the solution. To validate the proposal, several tests have been conducted using a prototype implementation; Section 6 presents their results. The last section concludes the article and identifies potential lines for future work.

2 RELATED WORK

2.1 Programming ML Workflows for the Continuum

In the continuum, data processing is needed in three different scenarios. The first scenario is traditional batch jobs triggered by a system administrator, an end-user or a scheduling system to analyse large amounts of at-rest data; for instance, train an ML model or run a large simulation. There are plenty of solutions in the bibliography building on the concept of task-based workflows; while some of these models are domain-specific and defined to target a specific problem (e.g., deal with collections of data [12] or run data analytics [21]), other models are general purpose[2, 7, 17].

The second scenario, called sense-process-actuate, considers that computation is triggered by the infrastructure when sensing an event and the IT platform is expected to process it and provide an appropriate response even activating some of the actuators to produce some physical effect. This is usually handled by submitting a Machine Learning inference operation to a Function-as-a-Service provider [1, 5, 11, 16].

Whereas the first two scenarios consider an eventual triggering of the computation, the last scenario considers the processing of continuous generation of data (stream) aiming at keeping the data stored in the system updated or producing other streams of data with the results. A stream of data could be processed using a workflow manager[22]; however, that would generate a lot of tasks and introduce a management overhead. Developers usually implement their solutions building directly in data distribution frameworks[8, 15]; Dataflows Managers [4, 19] are a more efficient approach to process streams.

In general, the currently available software development solutions target one of these scenarios and focus on tackling the specific challenges set out by the scenario they deal with; however, this technological heterogeneity incurs the adoption of a wide range of programming frameworks and models that hinder the efficient development of solutions targeting complete solutions of the continuum. To the best of our knowledge, Colony [16] is the only framework able to efficiently support the three scenarios with one single programming model thanks to its native support of the COMPSs programming model [17, 19].

2.2 Security considerations the Continuum

There are several approaches to ensure privacy and security in machine learning workflows: Early works in the area of preserving-privacy data mining and processing techniques have often relied on randomizing user data [10, 13, 18]. Although these approaches were promising, they often resulted in a lower accuracy compared to performing processing on plain text data. Furthermore, while these approaches aim to provide privacy of computation itself, they do not address and protect the results such as the trained model that are stored in the cloud or edge, nor do they secure and protect the inference phase.

A first approach that tried to tackle this challenge was published by Graepel et al. [14], which consisted of machine learning algorithms to perform both training and classification on encrypted data. The solution utilizes homomorphic encryption, however, such encryption schemes provide only very restrictive computing operations, and incur high performance overheads.

An alternative approach is the use of shielded executions as they provide strong security guarantees for legacy applications running on un-trusted platforms [9]. Several frameworks have evolved in this area such as the original Intel SGX SDK [3], abstractions such as Haven [9] or even library OS approaches such as Graphene-SGX [20]. Although these approaches/frameworks allow you to run a workload in trusted execution environments such as Intel SGX, they often require source code modifications while our approach can be applied in a transparent fashion through binary transformation techniques lowering the burden on the users.

3 COMP SUPERSCALAR (COMPSS)

COMPSs/PyCOMPSs is a programming framework aiming at easing the development of general-purpose applications targeting the Cloud-Edge-IoT Continuum [16]. The core of the framework is its task-based programming model [17] which provides a sequential fashion of programming that allows its runtime toolkit to orchestrate its execution on top of any distributed infrastructure. PyCOMPSs is an enhanced version of the programming model exploiting the benefits of the Python programming language. The following sections provide a description of the programming model used by the application developers and cast a glance at the architecture of the runtime system, highlighting those parts that are relevant for the protection of application data.

3.1 Programming Syntax

PyCOMPSs provides a sequential programming model to develop applications, hiding the complexity of the underlying infrastructure. The programming model includes the definition of the tasks and of the constraints (through Python annotations) to drive the scheduling phase at execution time. The application developer provides a sequential Python script whose functions are annotated through decorators; these annotations are used by the runtime to run those parts of the code as asynchronous parallel tasks code. When executed, the user code (the annotated part) is intercepted and analysed by the runtime, generating an execution graph.

The PyCOMPSs programming model provides a set of Python decorators that allow the user to identify the function/methods whose calls will be considered tasks and a small API for synchronisation. The main decorator is the `@task decorator`. This decorator can be placed on top of any function, instance method or class method and is used to identify the function's input/output parameters and return peculiarities.

The code snippet in Listing 1 depicts the selection of the function `train` to convert its invocations into tasks with the `@task` decorator across lines 2-5. The method takes as input three collections of objects (`x_list`, `y_list`, `id_list`) and one object (`random_state`), and produces 4 results.

```
1   @constraint(computing_units = "4")
2   @task(x_list = {Type: COLLECTION_IN},
3       y_list = {Type: COLLECTION_IN},
4       id_list = {Type: COLLECTION_IN},
5       returns = 4)
6   def train(x_list, y_list, id_list, random_state):
7       x, y, ids = _merge(x_list, y_list, id_list)
8
9       clf = SVC(random_state = random_state)
10      clf.fit(X = x, y = y.ravel())
11
12      ...
13
14      return sv, sv_labels, sv_ids, clf
```

Listing 1: Sample PyCOMPSs application selecting the `train` method to become a task

Besides, PyCOMPSs also supports the task's constraint definition. To this end, it provides the `@constraint` decorator, which also needs to be placed on top of the stack of decorators. In the previous example, the `@constraint` decorator in line 1 indicates that tasks for this function require 4 cores to run. Constraints are also used to let the developers provide hints on the fault tolerance at task level thus allowing them to discard parts of a workflow that do not lead to relevant results or that fail for some reason, without affecting the main application.

3.2 Runtime System

When a COMPSs application is deployed across the Continua, each node belonging to the infrastructure runs a daemon process, known as Agent, that handles the different computation requests that arrive through its API. When a monitoring system detects that new data has been collected and its analysis must be triggered or when an external user decides to run a computation, they submit to this API a request to run a function execution. Upon the reception of this request, the API submits to the Runtime engine a task that encapsulates the execution of the main code of the computation. As depicted in Figure 1, this engine is composed of four main components:

Resource Manager monitors the availability of computing resources (embedded on the device or available as agents on remote nodes).

Task Scheduler picks the resources and time-lapse to host the execution of each task while meeting dependencies among them and guaranteeing exclusive access to the assigned resources.

Data Manager keeps track of the data values located in the node and establishes a data sharing mechanism across the whole infrastructure

Executor Engine handles the execution of tasks on the resources embedded on the local device (CPU, GPU, FPGA or any other accelerator).

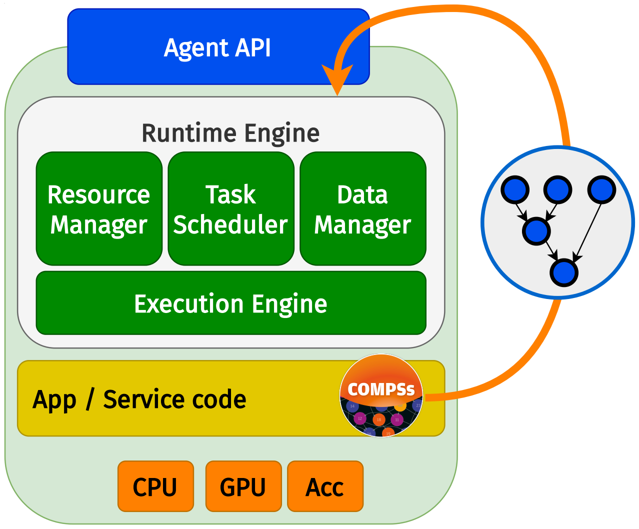

Figure 1: Components of the Agent running the COMPSs runtime

Upon the reception of the task, the runtime engine analyses the data values on which the task operates and identifies potential data dependencies with previously-submitted tasks. With this information, the runtime builds a directed acyclic graph representing the task dependencies; each node in the graph represents a task to run and the arrows among the nodes correspond to the data dependencies where the origin node represents the task producing and the end is the consumer task. When a task has no incoming arrows or all the incoming arrows correspond to a completed task, it means that the task is dependency-free; and thus, it is ready for execution. Once the runtime is aware of all the dependencies of a task and their status, the Task Scheduler plans its execution considering the availability of the local computing devices and the availability of the computing resources in other agents previously configured. If the Task Scheduler decides that the task should be offloaded onto another agent, it forwards the task to the remote agent through the API so the remote agent handles the task in the same manner. Conversely, if the Task Scheduler decides to host the task execution in the local computing devices, it requests the Data Manager to fetch all the necessary input data values and submits the task execution to the Execution Engine.

The Execution Engine is a multi-process component following a master-worker architecture as depicted in Figure 2. The master part is executed within the same process as the rest of the runtime. Its purpose is to receive the different tasks to execute and orchestrate

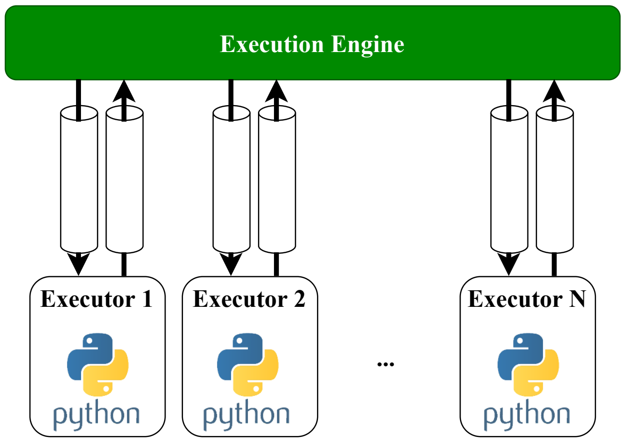

Figure 2: Diagram of the architecture of the Executor Engine component

their execution on the resource of the device via the worker processes. COMPSs allows combining the execution of tasks developed in different languages; it supports the native execution of Java, C and Python functions, but it also supports the execution of a task as an external binary or even a container. Upon the reception of the first task of each natively-supported language, the master deploys several worker processes that will deal with the actual execution of the tasks.

These workers, also known as *executors*, are simple processes that are waiting for their master to submit a task execution, they load the necessary input data values from the disk to memory, and execute the task's specific function operating on those values. When the local computing devices execute a code programmed following the COMPSs/PyCOMPSs programming model, new tasks are spawned and submitted back to the runtime engine so that the runtime handles their execution in the same way as it was done for the main task. At the end of the function execution, the worker writes the result values into the disk, lets the master know about the completion of the task, and waits for a request to execute another task. By hosting each executor in a different process, the execution of one task is isolated from other executions keeping its own memory space. The inter-process communication between the master and the executors builds on named pipes to establish a bi-directional channel connecting the master with each executor.

4 TRUSTED EXECUTION & SCONE

Intel SGX is a relatively new technology that combines a processor mode and a set of processor instructions to provide Trusted Execution Environments (TEEs). SGX ensures that processes are isolated from each other by utilizing a dedicated and cryptographically protected memory region (so called *enclave*) [3]. Enclaves use a contiguous memory region as a block of protected memory borrowed from the Dynamic Random Access Memory (DRAM) as Processor Reserved Memory (PRM). The PRM comprises the Enclave Page Cache (EPC), a set of 4KB memory pages, and enclave meta data. The PRM is neither accessible from other applications nor privileged code such as the operating system or the hypervisor. The Memory Encryption Engine (MEE), part of the processor, encrypts and authenticates data for non-PRM memory and protects EPC pages.

Although the Intel SGX SDK provides software developers interfaces to run applications in enclaves in its entirety, the usage of such interfaces is cumbersome as it requires developers to provide glue code for every possible system call an application may use. To overcome this burden, frameworks such as SCONE [6] or Haven [9] evolved that simplify or even completely eliminate manual steps required to run applications in TEEs using Intel SGX. SCONE achieves this by providing a custom standard c-library which contains glue code as well as all enclave- and system-call handling routines. Additionally, SCONE provides ready to use functionality for attestation and secret provisioning.

In order to enable applications to facilitate the hardware features provided by Intel SGX, applications must be either cross-compiled using SCONE's cross compiler or sconified where an existing docker image is modified and patched in such a way that native applications can run seamlessly in an enclave such as provided by Intel SGX. During the cross compilation process, the application is instrumented with several additional instructions and code that first pre-allocates memory in an enclave for loading the application code and data into it afterwards. It furthermore attests the previously loaded code and data to ensure that the application code and data is the expected one and has not been tampered during enclave loading with prior of its execution.

As mentioned previously, an alternative approach to the cross compilation is the use of prebuilt docker images or a so-called sconify-CLI tool which performs binary transformation/patching of existing docker images. In this process, the original glibc will be replaced with a modified one which contains the instrumented code which usually the cross compiler would weave in as described in the cross compilation process. The patched glibc code not only allows an application to be loaded and executed inside of an enclave, it also provides mechanisms to encrypt data at rest and in transit. This is achieved by intercepting the system calls such as read and write and en-/de-crypting the buffers provided as arguments before writing them out onto the underlying file system. This mechanisms can be used not only for writing and reading files but also to establish secure connections using TLS. In order to do so, the so called SCONE runtime will furthermore perform transparently TLS handshakes and as well as perform a mutual authentication ensuring that the respective counterparts can be trusted.

In order to harness the security mechanisms such as the file system and network protection shield, the confidential application will furthermore establish a secure connection the configuration and attestation service (CAS) which stores policies and information such as what volumes, i.e., directories on a local disk should be transparently en-/de-crypted and what other peers processes can establish secure connections with this process. The CAS instance will furthermore inject secrets as well as certificates preventing human access to these secrets unless absolutely necessary.

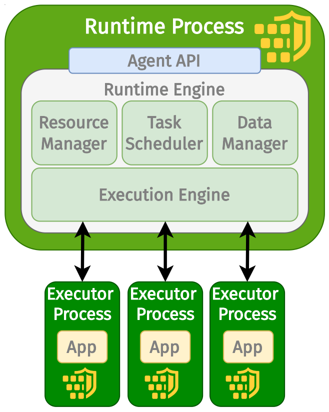

Figure 3: Diagram of the architecture components protected with SCONE with the first approach

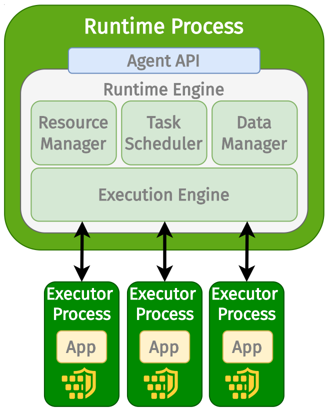

Figure 4: Diagram of the architecture of the solution with a selection of the components protected with SCONE

5 INTEGRATION

The work presented in this article pursues enhancing a software development solution for the Compute Continua by protecting in-memory application data in a transparent manner for both, the application developer and its end user. SCONE is able to provide this protection by instrumenting the memory accesses of a binary to keep the heap space of its host process encrypted with no implications for the developer or the end user. Hence, any software development solution – for instance, COMPSs – could achieve the purpose of this work if it is extended to leverage SCONE to protect the application data.

As explained in Section 3, COMPSs deploys in every node of the infrastructure an Agent that handles function executions in a Function-as-a-Service manner. Each agent works autonomously, and its memory is isolated from the other agents. Hence, protecting the in-memory data of a service deployed across the Continua is a problem that can be tackled by considering every agent individually. If all the agents composing a deployment protect their local in-memory data, then all the in-memory application data is safe from attackers.

A first approach to secure COMPSs' workflows consisted of sconifying the whole Agent; i.e., protecting with SCONE the process hosting the runtime system (a Java Virtual Machine) and all the executor processes as depicted in Figure 3. The implementation of this prototype following this approach is not trivial as it requires fork support of the agent process. On the one hand, for security reasons, SCONE does not forward the environment from a parent process to its forked children as variables could contain private information. Hence, the executors would not be able to receive the values of those variables that might be used by the application. Modifying the scripts that start the executors to run an environment setting snippet works around the problem.

Besides these challenges that can be solved through additional engineering effort, this approach entails another problem: the overhead due to allocating an encrypted heap space for the Agent process and for each executor and the additional cost of operating over an encrypted memory space. Delving into the details of the COMPSs architecture reveals that a significant part of these overheads is unnecessary. The purpose of this article is to protect the application data, and, as explained in Section 3, application data is loaded onto memory only by the executor processes. The process hosting the agent – and hence, the master part of its Execution Engine – only knows the identifiers of the data values involved in a task, but it is never aware of their content. Thus, to protect the in-memory data of an application, only the memory space of executors needs to be protected as depicted in Figure 4.

With this new approach, the starting of the Agent and the handling of the tasks remain exactly the same as in the original COMPSs version, hence, they do not entail any overhead. Conversely, when the Task Scheduler of the Agent decides, for the first time, to host locally the execution of a task, it spawns all the processes corresponding to the executors for that language. For instance, in the case of requesting the execution of a Python method, the Agent would spawn several Python interpreters running the code of the executor. In order to protect the memory of these new executors, the Python interpreter being executed can not be an out-of-the-box distribution; it must be a modified Python interpreter instrumented by SCONE that allocates an encrypted heap space at launch time and ensures that memory accesses handle data encryption. Thus, when the Agent starts several executors, they all allocate their respective encrypted heap space incurring a significant overhead that depends on their size. Once all the executors are ready, the Execution Engine starts submitting tasks to them. Upon their reception, the executors behave exactly the same as with the original COMPSs version with the only difference being that the modified interpreter handles encryption with every memory access.

6 EVALUATION

With the purpose of validating the described solution and measuring the impact of the overheads, the prototype has undergone several tests. The following sections describe the testbed that hosted the tests, the application being executed, and the results obtained.

6.1 Testbed

For the evaluation, we ran the experiment on a server equipped with two Intel Icelake ES2 XCC CPUs, each with 28 cores, 48MB cache and 64 GB DDR4 2933 RAM. As an operating system, we used Ubuntu 20.04.4 LTS with kernel 5.14.0. Furthermore, we used SCONE v5.7 to run the applications in Intel SGX enclaves.

6.2 Application: Random Forest Classification

RandomForest (RF) is a classification algorithm that constructs a set of individual decision trees, also known as estimators. Each estimator classifies a given input into classes based on decisions taken in random order. The final classification of the model is the aggregate of the result of all the estimators; thus, the accuracy of the model depends on the number of estimators composing it. These tests use the implementation provided in dislib [23], a library that provides application developers with a set of built-in AI algorithms able to run in a distributed manner leveraging PyCOMPSs.

The test measures the time to handle the subsequent classification of 20 samples using a pre-trained RF model when setting up a different number of executors (max number of tasks running in parallel). This model, composed of 40 estimators, classifies an ECG as a normal ECG, Atrial Fibrillation (AF), inconclusive or noise. For each classification, COMPSs generates the graph depicted in Figure 5 composed of 81 tasks with dependencies whose execution is orchestrated to run using a different number of executors. For each estimator, dislib may split its decision tree into several sub-branches and compute the prediction for each sub-branch (*predict_branch* tasks depicted in blue) and then merge the results of each branch to compute the prediction of the whole estimator (*merge_branches* tasks depicted in white). In this case, the model is small enough so that a single node hosts the prediction execution; hence, disLib decides to encapsulate the whole prediction in a single predict_branch task for each estimator. Finally, when all the estimators have computed their prediction, a last task (*soft_vote* depicted in red) gathers all the classifications and decides the overall classification for the sample.

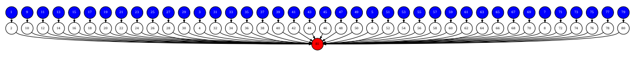

Figure 5: Task dependency graph generated by COMPSs for each Random Forest classification. Blue circles illustrate *predict_branch* tasks; white circles, *merge_branches*; and red circles, *soft_vote*

6.3 Test Results

As already explained in Section 5, the processing of the first request incurs a significant overhead due to the spawning of the executors. For this reason, the execution time for processing the first sample is considered apart from the others. Table 1 contains the average execution time obtained for processing the first request according to the number of executors deployed on the node (from 1 to 32) comparing the performance when SCONE is disabled (Plain) and the performance with SCONE being set up to use different sizes of memory heap (2GB, 8GB and 32GB). The memory size of the host sets a hard limit on the number of executors; with a heap size of 32 GB, the host node can only allocate up to 8 executors.

Table 1: Execution Time (ms) of the first request to perform a Random Forest classification of a 1-sample input according to the number of processes spawned (executors) and the heap size of scone (if secure).

	# executors					
	1	2	4	8	16	32
Plain	6,563	3,335	1,857	1,257	1,024	1,034
2 GB	16,189	11,601	10,919	12,317	19,290	31,825
8 GB	27,330	22,695	25,641	34,066	82,137	188,765
32 GB	75,135	73,265	146,691	219,712	-	-

Figure 6 illustrates the evolution of the execution time for this first request on each security setup when the number of executors increases. With it, it is plain to see the effect of securing the application with scone on the first request. When data is left unprotected (plain line), COMPSs is able to reduce the execution time of the first request as the number of executors increases. As expected, all the protected versions incur some overhead on the spawning of the executors, and the size of the heap has a significant effect on its magnitude. In the 1-executor case, the execution time – 6,563 ms with the unprotected COMPSs – grows along with the size of the heap reaching a 1100% overhead when the executors require 32GB – 75,135 ms. The larger the number of executors increases, the bigger this overhead becomes. The memory allocation of the multiple executors stacks up reaching 219 seconds to spawn 8 executors in the 32GB case (in total 256 GB of memory) – 174 times slower than the 8-executors plain case –, and 188 seconds in the 32 8-GB executors (also 256GB of memory) – 182 times slower than the 32-executors plain case.

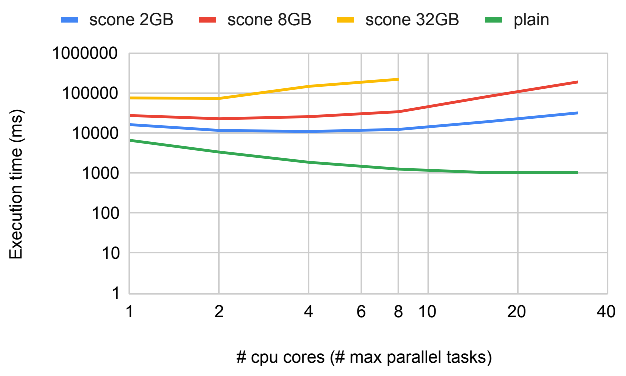

Figure 6: Evolution of the execution time of the first request when COMPSs is unprotected (plain) and protected with scone using a heap space of 2GB, 8 GB and 32 GB

Once an Agent has deployed all the processes for the executors, these overheads no longer apply to the processing of the subsequent requests. Table 2 contains the average execution time obtained for processing a request at operation time – i.e., dismissing the first request – according to the number of executors deployed on the node (from 1 to 32) comparing the performance when SCONE is disabled (Plain) and the performance with SCONE being set up to use different sizes of memory heap (2GB, 8GB and 32GB). Figure 7 illustrates the evolution of this average execution time on each security setup when the number of executors increases.

Table 2: Average Execution Time (ms) of the Random Forest classification of a 1-sample input according to the number of processes spawned (executors) and the heap size of scone (if secure).

	# executors					
	1	2	4	8	16	32
Plain	5,857	3,137	1,740	1,022	659	584
2 GB	7,446	3,942	2,233	1,426	1,013	1,019
8 GB	7,610	4,119	2,258	1,404	1,011	1,040
32 GB	7,190	3,921	2,180	1,370	-	-

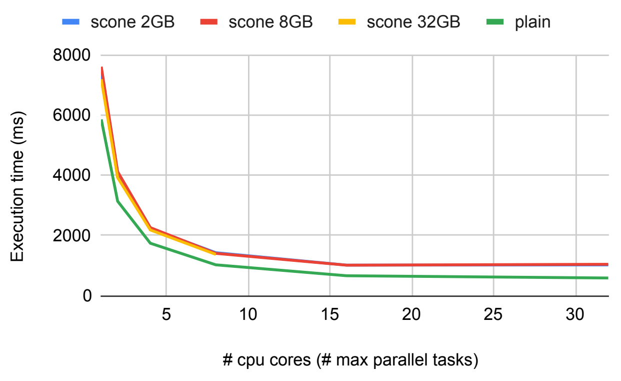

Figure 7: Evolution of the average execution time of a request (ignoring the first one) when COMPSs is unprotected (plain) and protected with scone using a heap space of 2GB, 8 GB and 32 GB

Unlike in the case of dealing with the first request, at operation time, the heap size does not affect the response time. As the chart in Figure 7 shows, all the secured versions of COMPSs behave in a similar manner. Although the average times have some small differences, their statistical significance is minimal and their difference could be explained by other factors like the workload from other applications. When comparing the secured version to the plain, the experiment reveals that the overhead of operating with an encrypted memory slows down the execution by about 30%. However, it is important to notice that this memory encryption overhead does not prevent the COMPSs runtime to improve the performance of the application as the number of executors increases up to 16. Beyond that, the application scalability in the node seems to stall, probably because the encryption accelerator is being shared among several processes.

7 CONCLUSIONS

In this paper we presented a proposal for the design and execution of distributed applications in secure environments. The described system adopts in-memory data protection that allows users to develop parallel applications without the need to deal with the intricacies of the underlying computational infrastructure and with the mechanisms of explicitly encrypting the data.

The framework is based on the COMPSs programming framework that enables sequential applications to be parallelized and executed automatically scaling the computation across the distributed nodes of the computing continuum. The protection of the application data is achieved through the integration of the COMPSs runtime with the SCONE framework that allows to run the user code in trusted environments. The evaluation of the execution of a ML application demonstrate that the scalability of the COMPSs application is kept also taking into account the overhead introduced by the encryption of the memory.

As future work we are investigating how to reduce the overhead and how to increase the performance on a larger number of nodes.

ACKNOWLEDGMENTS

This work has been supported by the Spanish Government (PID2019-107255GB) and by MCIN/AEI /10.13039/501100011033 (CEX2021-001148-S), by Generalitat de Catalunya (contract 2021-SGR-00412), and by the European Commission through the Horizon Europe Research and Innovation program under Grant Agreement No. 101016577 (AI-SPRINT project).

REFERENCES

[1] 2014. *Amazon Lambda.* https://aws.amazon.com/lambda/
[2] 2014. *Apache airflow.* https://airflow.apache.org/
[3] 2014. *Intel Software Guard Extensions Developer Guide.* https://software.intel.com/en-us/sgx-sdk/documentation
[4] 2016. *Apache Beam.* https://beam.apache.org/
[5] 2017. *Amazon Greengrass.* https://aws.amazon.com/greengrass/
[6] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 689–703. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov
[7] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S Katz, Ben Clifford, Rohan Kumar, Lukasz Lacinski, Ryan Chard, Justin M Wozniak, Ian Foster, et al. 2019. Parsl: Pervasive parallel programming in python. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*. 25–36.
[8] Gabriele Baldoni, Julien Loudet, Luca Cominardi, Angelo Corsaro, and Yong He. 2021. Facilitating distributed data-flow programming with Eclipse Zenoh: The ERDOS case. In *Proceedings of the 1st Workshop on Serverless mobile networking for 6G Communications*. 13–18.
[9] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding Applications from an Untrusted Cloud with Haven. In *33rd ACM Transactions on Computer Systems (TOCS)*. ACM. https://doi.org/10.1145/2799647
[10] Raphael Bost, Raluca Popa, Stephen Tu, and Shafi Goldwasser. 2015. Machine Learning Classification over Encrypted Data. https://doi.org/10.14722/ndss.2015.23241
[11] Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. 2020. Funcx: A federated function serving fabric for science. In *Proceedings of the 29th International symposium on high-performance parallel and distributed computing*. 65–76.
[12] Dask Development Team. 2016. *Dask: Library for dynamic task scheduling.* https://dask.org
[13] Wenliang Du and Zhijun Zhan. 2003. Using Randomized Response Techniques for Privacy-Preserving Data Mining. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Washington,

D.C.) *(KDD '03)*. Association for Computing Machinery, New York, NY, USA, 505–510. https://doi.org/10.1145/956750.956810

[14] Thore Graepel, Kristin Lauter, and Michael Naehrig. 2013. ML Confidential: Machine Learning on Encrypted Data. In *Information Security and Cryptology – ICISC 2012*, Taekyoung Kwon, Mun-Kyu Lee, and Daesung Kwon (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–21.

[15] Jay Kreps, Neha Narkhede, and Jun Rao. 2011. Kafka: a Distributed Messaging System for Log Processing. *ACM SIGMOD Workshop on Networking Meets Databases* (2011).

[16] Francesc Lordan, Daniele Lezzi, and Rosa M. Badia. 2021. Colony: Parallel Functions as a Service on the Cloud-Edge Continuum. In *Euro-Par 2021: Parallel Processing*, Leonel Sousa, Nuno Roma, and Pedro Tomás (Eds.). Springer International Publishing, Cham, 269–284. https://dx.doi.org/10.1007/978-3-030-85665-6_17

[17] Francesc Lordan, Enric Tejedor, Jorge Ejarque, Roger Rafanell, Javier Álvarez, Fabrizio Marozzo, Daniele Lezzi, Raül Sirvent, Domenico Talia, and Rosa M Badia. 2014. ServiceSs: An Interoperable Programming Framework for the Cloud. *Journal of Grid Computing* 12, 1 (2014), 67–91. https://doi.org/10.1007/s10723-013-9272-5

[18] Do Le Quoc, Martin Beck, Pramod Bhatotia, Ruichuan Chen, Christof Fetzer, and Thorsten Strufe. 2017. PrivApprox: Privacy-Preserving Stream Analytics. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 659–672. https://www.usenix.org/conference/atc17/technical-sessions/presentation/quoc

[19] Cristian Ramon-Cortes, Francesc Lordan, Jorge Ejarque, and Rosa M. Badia. 2020. A programming model for Hybrid Workflows: Combining task-based workflows and dataflows all-in-one. *Future Generation Computer Systems* (2020). https://doi.org/10.1016/j.future.2020.07.007 arXiv:2007.04939

[20] Chia-Che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA, USA) *(USENIX ATC '17)*. USENIX Association, USA, 645–658.

[21] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark : Cluster Computing with Working Sets. *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing* (2010). https://doi.org/10.1007/s00256-009-0861-0 arXiv:arXiv:1011.1669v3

[22] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. 2012. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. (2012).

[23] Javier Álvarez Cid-Fuentes, Salvi Solà, Pol Álvarez, Alfred Castro-Ginard, and Rosa M. Badia. 2019. dislib: Large Scale High Performance Machine Learning in Python. In *Proceedings of the 15th International Conference on eScience*. 96–105.