Auditability for Secure Machine Learning in the Cloud

PIC2 - Master in Computer Science and Engineering
Instituto Superior Técnico, Universidade de Lisboa

Daniel Ferreira — 1112291*
ist1112291@tecnico.ulisboa.pt

Advisors: Professors Luís Rodrigues and Nuno Santos

Abstract

Machine learning systems are increasingly being deployed in cloud environments due to the scalability they provide. When doing so, users place their trust in the cloud service provider and the whole environment where their workload is executed. This opens up a variety of attacks since any privileged actor from the cloud service provider is capable of inspecting and tampering with the data and code provided by the user. In addition to this, the lack of transparency in the cloud makes it difficult to understand what actually ran in each workload that is executed, highlighting the need to collect relevant information about processes that are run so that machine learning pipelines can be audited in the future. Because of that, recent work has explored the implementation of provenance-based systems to later audit the machine learning phases that were executed. Some of these systems have been deployed in trusted execution environments so that users can trust that their workloads were executed without being tampered with and in the conditions that were expected in adversarial cloud environments. However, none of the systems provide a solution to capture verifiable provenance in all phases of a machine learning lifecycle while binding each client to the phases they execute. In this work we propose VeriProv, a provenance-based framework for enforcing and auditing non-repudiable and verifiable ML workflows in adversarial cloud environments.

Keywords — Machine Learning, Data Provenance, Confidential Virtual Machines, Auditability, Accountability

This work was supported by the EU's Horizon Europe research and innovation programme under Grant Agreement No 101189689 (ACHILLES).

*I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa (https://nape.tecnico.ulisboa.pt/en/apoio-ao-estudante/documentos-importantes/regulamentos-da-universidade-de-lisboa/).

Contents

1 Introduction

Currently the AI market is still quickly growing, and many of its applications are being deployed in the cloud. These environments are a great fit for teams deploying their AI applications without having their own infrastructure in place, due to their scalability, ability to handle large workloads and vast computational resources, allowing these applications to 'hyper-scale' [1]. With this increase, it is important that we understand what components are used in each phase of these applications, so that we can trust the artifact that is created and, for instance, debug the process in case an error occurs or make a person accountable for a malicious or erroneous action. Unfortunately, understanding how each phase of an AI application works is often difficult and their deployment in the cloud makes the process even more opaque, which degrades the trustworthiness of these applications.

One of the solutions presented to alleviate this problem are model cards [2] and datasheets for datasets [3], two types of short documents that model and dataset providers can develop in order to give more insight on properties of these artifacts. Datasheets, for example, provide information about how the data was collected, which sources were used, and any transformations that were performed, and model cards describe the model's intended use, the data used to train it and the evaluation of the model.

To ease the extraction of properties or components used in different machine learning phases some authors have built systems that can extract provenance data from phases of an AI application [4–7]. These systems usually extract information such as what dataset and configurations were used to train a model or how well a certain model was evaluated.

Despite these recent advancements in dataset and model documentation, and machine learning provenance capture, existing systems fail to provide verifiable provenance capture of full non-repudiable machine learning pipelines. A central goal of this work is to ensure non-repudiation of ML computation phases, binding each execution step to an accountable actor. A solution to provide these assurances is especially relevant in an adversarial cloud environment, where a malicious cloud provider can silently tamper with both the execution of the workload or the data and provenance records being captured.

Although datasheets and model cards give more context to datasets and models that were produced, they are fully controlled by their producers, who may lie about the properties of their artifacts. Because of this, users have to trust in the entity's reputation to trust in the underlying dataset or model. In addition to this, general machine learning provenance systems such as MLFlow [4] and ProvLake [8] do not provide any cryptographic guarantees of the data that is captured or the environment and runtime used for each machine learning phase. Systems like Laminator [6] and VerifiableFL [7] tackle this problem by running the workloads in trusted execution environments. However, none of these systems bind each machine learning phase to the responsible actor, which allows attackers to safely train a model with a malicious dataset without being accountable in the future. In addition to this, some of these systems focus in specific machine learning phases, such as training or evaluation, instead of covering the full machine learning pipeline.

The objective of this work is to fill the gap of providing the non-repudiation and policy enforcement in end-to-end machine learning pipelines in an adversarial cloud environment by running the workloads in confidential virtual machines and storing the recorded provenance in a provenance graph.

In summary, this work proposes a provenance-based framework for enforcing and auditing non-repudiable and verifiable machine learning workflows in adversarial cloud environments.

The rest of this report is organized as follows. Section 2 introduces the goals and expected results of our work. Section 3 presents all the necessary background to understand the related work described in the same section. Section 4 proposes an architecture for the solution that will be developed and in Section 5 we describe how the proposed solution will be evaluated. Finally, in Section 6 we present the work schedule planned to develop our work and we finish the report with a short conclusion in Section 7.

2 Goals and Expected Results

In this work we address the problems of supporting auditability in the execution of machine learning tasks in the cloud. More precisely:

> *Goals:* We aim to introduce i) a binding mechanism between the identity of actors and the execution of machine learning workloads in the cloud to guarantee the non-repudiation of all actions that are performed, and a policy engine that can check for the compliance of user-defined policies before running a machine learning workload for more flexible application-level guarantees; ii) a functional prototype, called VeriProv, that can safely capture provenance of machine learning workloads executing in confidential virtual machines in the cloud with acceptable overhead, and a way to safely store and query the attestable records that are collected.

This work is expected to produce the following results:

> *Expected Results:* this work aims to produce i) a functional prototype to capture provenance and enforce a collection of examples policies in machine learning workloads that can be deployed in a confidential virtual machine in the cloud; ii) a provenance store that safely stores attestable records in a graph structure; iii) a query model to audit the information that has been previously stored; iv) a set of evaluation metrics to measure the effectiveness of the solution.

The main contribution of this work is a provenance-based framework that ensures non-repudiation of machine learning workflows in adversarial cloud environments while enforcing runtime policies and providing an auditable storage of attestable records.

3 Background and Related Work

3.1 Background

In this section we will present the necessary background for our work. Section 3.1.1 defines trusted execution environments, Section 3.1.2 introduces confidential virtual machines and Section 3.1.3 discusses the main aspects about provenance.

3.1.1 Trusted Execution Environments

A trusted execution environment (TEE) is an environment that is tamper-resistant and can guarantee the confidentiality and integrity of its code, data and runtime state, such as the CPU registers of a host machine. One of its main contributions is to allow users to execute their sensitive code with potentially sensitive data with the assurance that it is not read or modified during execution while providing a way to verify the correctness of these guarantees.

TEEs typically rely on hardware-based mechanisms rather than software-based ones, as hardware operates at a lower abstraction level and reduces the size of the trusted computing base, thus minimizing the number of components that must be trusted. In this case, the trusted computing base is composed solely of the hardware that is used and the application that is executed in the TEE. With this trusted computing base, TEEs assume the existence of adversaries that have privileged access to the machine's whole host operating system, hypervisor and all host software and applications that are executing. This includes the machine's administrator, third-party software that is installed in the machine, a compromised host operating system or hypervisor, or a cloud service provider.

A TEE guarantees three major properties: isolation, secure storage and remote attestation. Isolation guarantees that no outside software or entity, such as a malicious administrator, the operating system, or an hypervisor can access its data or tamper with its execution. TEEs also provide secure storage with persistent storage that cannot be modified by any outside mechanism. Finally, remote attestation allows an external third-party to obtain an authentic and timely report about the software and data state of a TEE, for example, to verify that the

code provided to the TEE was not modified and that it is executing correctly. This last property is what makes a user trust the machine is running as expected.

To perform a common remote attestation of a TEE the following general steps are normally followed:

1. Create a secure connection between the verifier and the trusted execution environment;
2. The verifier sends a challenge to the TEE, asking for a proof of the code and data that is being used;
3. The TEE gathers evidence for the verifier challenge, which can be, for example, a chain of hashes of all components in the TEE's trusted computing base and the software components loaded in it, that prove the correct initialization of the TEE. In the future, a verifier can check the TEE system integrity by using this measurement;
4. The attestation data is signed by the TEE with an attestation key and sent to the verifier. The verifier challenge is also signed to guarantee freshness;
5. With the signed attestation data provided by the TEE, the verifier starts by ensuring that the signature is correct. After that, it verifies that the signed data contains its challenge, to guarantee freshness. Finally, it inspects the provided measurement and uses a policy to decide if the TEE state can be trusted. To do so, the verifier usually compares the measurement it received with a list of measurements that are known to be trustworthy.

Due to these strong security guarantees, TEEs are being used to provide trust in applications running in the cloud, since that allows users to run their workloads safely without having to trust in the cloud provider. This also opens opportunities to safely run machine leaning applications in the cloud with the guarantees that the training or inference of models are not tampered with. However, despite these guarantees, regular hardware-based TEEs, that provide application-level security, have limitations in terms of memory size, ease of programmability and performance, which hinders the development and deployment of full machine learning pipelines with this technology. There has been however some work done to safely integrate TEEs with untrusted accelerators such as GPUs in order to speed up the execution of these applications. Although it is usually considered an out of scope attack vector, TEEs are also vulnerable to side-channel attacks which can compromise the confidentiality of the data and code that is stored in the TEE.

3.1.2 Confidential Virtual Machines

A confidential virtual machine (CVM) is a type of virtual machine that runs as a trusted execution environment, extending the security guarantees of a regular TEE to full operating system and more complex applications. Like regular TEEs, CVMs are designed to guarantee the confidentiality and integrity of the code and data that reside in it from any component residing outside of the CVM environment, such as the hypervisor or a cloud service provider.

In contrast with traditional TEEs, which execute isolated applications with a small trusted computing base, CVMs protect entire virtual machines, which includes the guest operating system, application stack, and data. This larger scope allows developers to use more familiar tools, libraries, and programming environments while still having access to hardware-based isolation and attestation that come from regular TEEs. Because of that, CVMs ease the secure deployment of complex and full machines learning pipelines spanning data transformations, training, evaluation and inference stages and are becoming more popular to deploy these kind of applications securely.

However, the reliance in these additional abstractions and tools enlarges the trusted computing base of the CVM, since the inclusion of the virtualized guest operating system also makes the attack surface larger than the one observed in regular TEEs.

CVMs are supported by major cloud providers such as Microsoft Azure through Azure Confidential Computing, Google Cloud with Confidential VMs, and AWS through Nitro Enclaves and Nitro-based CVMs.

3.1.3 Provenance

Provenance, also sometimes referred to as lineage, is a term used to refer to the metadata that records all the components and their relationships that contribute to the existence of a piece of data. That is, provenance captures the origin, transformations and dependencies of data as it moves or is used through a system. This information has several applications, such as, providing understandability and transparency of processes by showing how an end product was obtained, or aiding in the reproducibility of processes by keeping the inputs and transformations used to obtain a certain result.

Data provenance is often divided into why-, where- and how-provenance [9], which focuses in different questions a user might ask about the origin or transformations of a data item. Why-provenance tries to answer why an output exists by identifying the subset of input that produced it, which answers questions such as "Which dataset was used to train this model?". How-provenance describes the derivation process of a certain result, identifying the intermediate transformations that were used to compute it. This answers queries such as "How was this prediction computed from the input and which transformations were applied?" Finally, where-provenance tracks information about the origin of specific values of the output by specifying which specific values of the input were used to compute it, which answers queries such as "Where in the dataset did this feature value come from?". In machine learning these types of provenance can, for example, help explain how a model was trained or in what data and transformations a model inference depends on.

When choosing how to collect provenance data from an application, the provenance data's granularity needs to be considered, that is, the level of detail of the information that is collected [10]. When considering coarse-grained granularity we assume the script or workflow to be executed in a black box model. Because of that, we assume that the output that is generated at the end of execution depends solely on all of the inputs used in the script and the execution context that is used. For example, the inputs and outputs used in a model training step would be considered as coarse-grained provenance. Fine-grained provenance includes the provenance of individual data items, for example, of a specific row in a dataset, including the data transformation they go through to generate a certain output.

In addition to this, there are also several different strategies to actually capture the desired provenance level. These different capture strategies can be categorized by workflow-, process-, or operating system-based strategies [11]. Workflow-based strategies are normally coupled to a certain workflow system which enables an easier way to collect provenance through the system's API. Process-based solutions on the other hand have to instrument each script they use automatically or manually to extract provenance. However, they let the user define exactly what information they want to collect although they are limited to capturing provenance of runtime information. Finally, operating system-based solution have the advantages of not requiring the modification of the scripts or workflows they want to execute and being agnostic about how the scripts or workflows are built, since they capture only provenance of runtime information with the inspection of system calls that are executed. This captures richer information at the cost of possibly storing too much data which in addition to using a lot of storage, also hinders querying and visualizing the captured data.

After collecting the necessary provenance, the information can be stored in different ways to later be audited. Previous work has stored provenance in relational databases, XML documents or RDF graph databases. The choice of query language is tightly coupled with the storage solution that is chosen. Most of the times, the collected provenance is represented as a directed cyclic graph where each node represents an entity or a process and each edge represents a relationship between two nodes. This allows why-, how- and where-provenance to be naturally represented [12].

Finally, there should be some security considerations when storing provenance. Firstly, in order for provenance to be useful it is expected that it cannot be tampered with, so that users can trust in the information that was captured. In addition to this, the order of the records that are captured should also not be modified, since the relationships between records can change depending in the representation strategy. Lastly, the necessity of securing confidentiality of the stored data should also be considered when storing sensitive information.

3.2 Related Work

In this section we will present systems that might or not use TEEs to capture provenance of general scripts and machine learning applications with different strategies and provenance granularity. We will end this section

with a discussion to compare the systems that are presented.

3.2.1 NoWorkflow

noWorkflow [13] is a command line tool designed to capture provenance of Python scripts. Its primary objective is to automatically collect the necessary provenance information without requiring the user to instrument the script itself, while also providing tools to analyse the data that is collected.

noWorkflow captures three different types of provenance: definition provenance, which captures information about the script definition such as function definitions, its arguments, and function calls; deployment provenance that collects information about the execution environment such as the operating system used, environment variables that were set and libraries the script depends on; and execution provenance which captures runtime information of a script that is executed, such as how many times a function is called, and the values that are used as arguments in each call.

Since provenance is captured automatically, the authors had to choose the right level of provenance granularity so that the amount of information collected was neither too overwhelming to analyse nor too sparse to draw conclusions. Because of that, noWorkflow only captures provenance of user-defined functions.

To capture provenance, this tool starts by capturing definition provenance by using AST analysis to capture the code structure, followed by using Python libraries to collect information about the environment that will be used to execute the intended script, and finally it uses Python's profiling API to capture execution provenance by tracing calls to user-defined functions and monitoring file access. All data is stored locally in SQLite and can be analysed with graph visualizations, by comparing two different executions, or by querying it using Prolog.

Although noWorkflow is a great tool to capture provenance of general Python scripts, it does not provide any security guarantees such as verifiable attestation records of data that is collected or non-repudiation of the workloads that are executed. Users using this system have to trust the entire environment used to execute the desired Python script since noWorkflow does not use any confidential computing solution. Its focus is on local provenance capture of scripts, and is therefore not suitable for secure auditability in adversarial cloud environments as is.

3.2.2 ReproZip

ReproZip [14, 15] is an open-source packaging tool that leverages the collection of provenance data of arbitrary scripts to enable their reproducibility. Its main objective is to make reproducibility more accessible and simple, allowing researchers to automatically trace the execution of a script, to package all the necessary dependencies and send it to another person who can later run the exact same experiment in its own machine. This allows the researcher to not be worried with the application's reproducibility, instead of having to manually record every input, dependency, environment variable and the execution environment that was used to run a script to later reproduce its execution.

This tool is divided into two main functionalities: *packing*, which captures everything needed to reproduce a script's execution and creates a package with all of the necessary information that was collected; and *unpacking*, which is able to extract the content stored in a previously built package and reproduce the exact same experiment in a potentially different environment.

To create a package for an experiment, ReproZip captures provenance from a command that is executed by using *ptrace* to collect all system calls that are used during the experiment's execution, storing all information in a local SQLite database. All of the data is analysed to identify all input and output files and software packages the experiment depends on. Packaged application can later be run in any execution environment by either setting up the necessary environment variables for execution, or by using Vagrant or Docker to set up a virtual machine or a container.

Although ReproZip is a great tool to provide reproducibility of experiments, it cannot capture application level information since it only collects what system calls are executed and information about the execution environment that is used to run the experiment. Because of that, machine learning specific data cannot be captured, which is essential for our work.

3.2.3 MLflow

MLflow [4] is an open-source platform that helps manage the machine learning development lifecycle. Its goal is to provide tools that ease the development and deployment of machine learning models in large teams. This platform follows an *open interface* philosophy by providing general interfaces for different abstractions such as training a model, deploying it, or storing a locally trained model, while allowing the users to use their own code, libraries and workflows to implement their desired software. This platform's functionality can be divided in three major modules: a tracking module which tracks the experiments ran through the platform, a projects module that packages code and other necessary parameters so that it can be easily reused, and a models module, which packages models using a generic format to later deploy them in different environments.

In our work, we are mainly interested in the MLflow Tracking module, which records information about each training run that is executed using the platform. Each run can store various metadata defined by the programmer itself such as the code used, data, input, output files and other parameters. The tracking is enabled by a Python API provided by MLflow and that can be used by developers to specify what information they would like to track in the code files they write. All of the logs are collected by default to a local directory of the host machine but can also be redirected to a custom server to make it accessible to the whole team. The collected logs can later be queried through an API or MLflow's web solution.

Although this tracking tool is extremely versatile, it assumes that all users can be trusted, with no associated threat model. The logs that are collected do not give verifiable guarantees about the integrity and authenticity of the data, since any log can be tampered with or deleted without detection by any team member. In addition to this, the tracking model focuses on storing information about the training and evaluation steps, disregarding information about inference queries that are made to deployed models. This does not allow a user to easily make a connection about the evaluation or training components that were used for the construction of a model that was queried. Finally, MLflow logs are not bound to identities so any user can deny having trained a model or claim that the logs were generated by someone else. These properties make MLflow unsuitable to support auditability and accountability in an adversarial cloud environment.

3.2.4 Lima

LIMA [16] is a low-granularity data lineage tracing and reuse framework used inside ML systems. It captures fine-grained data lineage of a machine learning process, keeping track of each variable used in the runtime in the most space-efficient way possible. Its main goal is to be able to use the captured lineage traces in order to avoid duplicate computation of intermediate values (e.g. calling the same method with the same parameters twice), accelerating the whole machine learning algorithm. However it also serves as a space-efficient way to capture low-granularity lineage information of a model's training, such as which part of the dataset is relevant to the model's training.

To do so, the framework maintains a lineage DAG where each node is an operation that has been executed with the associated inputs and outputs and the edges are data dependencies. This graph determines exactly how each intermediate value was produced without including the computation that determined the control flow path to reach each instruction.

Instructions in the script are instrumented so that the DAG can be built during execution, since all necessary information to build the operation's node (e.g. inputs, opcode, output...) is collected before the operation itself is executed. The lineage graph that is built can later be serialized into a lineage log that can be stored and later inspected.

Due to the capture of fine-grained provenance, the generated graph can become large. The authors apply several compression techniques, such as deduplicating recurrent subgraphs, in order to reduce memory usage.

Although LIMA offers space-efficient lineage tracking and supports runtime analysis of the script that is executed, its design is aimed at optimizing runtime reuse rather than auditability or verifiability. It does not include any mechanisms for authenticating records, enforcing policies, or binding accountable actors to the script that is executed, making it unsuitable for secure provenance in adversarial cloud environments.

3.2.5 ProvLake

ProvLake [5, 8] is a provenance management system that supports runtime capture, storage and querying of provenance data from multiple distributed workflows that use data from distinct data stores in large-scale scientific applications. Its main goal is to offer a system that can capture runtime data from these distributed workflows in a way that does not incur in high computational overheads for the actual application while enabling end-to-end traceability between all the captured workflows.

To achieve its goals, ProvLake is divided into five main components: the ProvLake library, which can be imported into the user's script to capture provenance of the execution environment and of information selected by the user; ProvCapturer, a service that is responsible for processing the requests that come from the ProvLake library by generating the relationships between the data that is collected and by converting the data into a specified standard; ProvManager which receives the processed data from ProvCapturer and stores it into a centralized provenance store in the form of a graph; the PolyProvQueryEngine, which is responsible for providing a provenance query API for the clients to use; and the Messaging System, which is used to support the use of asynchronous communication between all the components.

With the use of these components, ProvLake provides a system that is capable of minimizing the provenance capture overhead in multi workflow environments while storing relationships between the workflows that are executed, in contrast with MLflow where these relationships need to be inferred externally. However, this system does not offer confidentiality, integrity or authenticity guarantees for the provenance records that are stored. In addition to this, it also does not ensure the non-repudiation for the code that is executed in each workflow. This system assumes that the code and environment where each workflow and ProvLake component is executed is not changed by a malicious actor, and, because of that, the correct execution of ProvLake's components and of each workflow cannot be verified. Because of this, ProvLake cannot be safely hosted in an adversarial cloud setting as is.

3.2.6 PraaS

PraaS [17] (Proof as a Service) is a cloud solution that leverages the use of the Intel SGX TEE to provide verifiable proofs of dataset's properties, such as the mean or standard deviation of a dataset.

Its aim is to provide a system that dataset owners can use to prove their datasets hold particular properties without having to reveal the dataset itself to the public. In addition to that, model owners that want to use a particular dataset can verify its quality by inspecting the properties that were calculated before using the dataset blindly and jeopardizing their model unnecessarily. This system proves to be useful, for example, in a federated learning environment where the model owner would like to make sure that the datasets it is using are valuable, while making sure that they remain private.

In this work, the authors assume that users have access to an infrastructure that provides up-to-date TEEs and they do not take vulnerabilities in the TEE's hardware into account. Dataset owners can be malicious and try to convince other users that its dataset holds properties that it does not hold.

Since the attestation quote obtained from a TEE is not linked to the computation that is performed in it, it is not possible to convince a third-party user that the properties calculated in an enclave are trustworthy. Because of that PraaS creates an ephemeral key pair in their enclave code and includes a hash of the public key in the attestation report. After attestation, results computed inside the enclave are signed with the corresponding private key. Consumers can then verify that the results (which include dataset hashes, property values, and optionally, function code) were generated by a genuine enclave. The size of these proofs can vary due to the amount of properties that are computed over the dataset. However, the authors determined that the size of the combined attestation quote, and the hashes of the input dataset, optional Python function code and the signature of the result are 11.74KB in total.

While PraaS provides strong guarantees for capturing dataset properties, it is limited to this specific use case, without considering capturing provenance of any machine learning phase and providing end-to-end auditability of machine learning pipelines. In addition to this, it also does not mention the storage strategy used to keep all generated records and lacks a non-repudiation mechanism, making it unsuitable as a general solution for accountable machine learning pipelines executing in the cloud.

3.2.7 Laminator

Laminator [6] is the first framework to build verifiable machine learning property cards with the use of property attestations in TEEs. Its goal is to guarantee that the information in the property cards are correctly measured and not tampered with. In addition to this, they allow users to verify the inputs used and outputs produced during training and inference of an AI application and understand certain properties of a model that is trained, such as accuracy, fairness and robustness, leading to trust in the artifacts that are built. Laminator also introduces the concept of inference property cards, that capture a binding between a query, a model and the query's output.

Laminator considers an adversary model where the actor that trains, evaluates and deploys a model can try to provide a valid attestation to a property that is not verified in it. If he creates this attestation without a verifier noticing, the attacker is successful.

To issue property attestations, Laminator assumes there is an attestation mechanism using a TEE that executes a piece of code, produces a result and signs specific assertions, if and only if, the assertions hold. For example, an inference property attestation is created if and only if the output that was calculated is the same as the result of querying the deployed model with the input provided by the client. A general representation for creating a verifiable attestation for a machine learning phase can be seen in Figure 1. The measurer is the Python script that is executed and that is responsible for measuring the properties of a dataset, model or inference and for creating the attestation with the recorded properties. All phases always receive an input for the script to be executed and produce both an output and an attestation record which binds the inputs, outputs and code that was executed with the environment that was used. The TEE is also loaded with Python and Pytorch in order to execute the measurer script and Gramine, which is a library OS used to run unmodified applications inside Intel SGX enclaves.

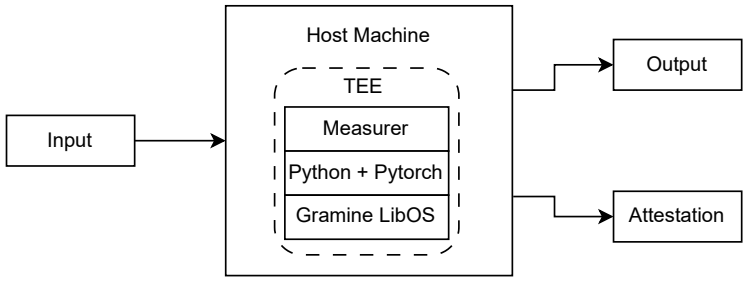

Figure 1: High-level architecture for verifiable execution of a machine learning phase in Laminator

The issued attestations by themselves are not very useful. Because of that, whenever a verifier wants to analyse the relationships between the attestable records that have been collected they can later be chained appropriately to provide meaningful conclusions such as "the query Q was answered by model M which exhibited an accuracy of 87%", or "model M was trained with a dataset D with a distribution property P".

While Laminator provides a strong foundation for verifiable provenance capture of machine learning pipelines, it lacks mechanisms for non-repudiation of each phase that is executed in a TEE, which means actors cannot be held accountable for each machine leaning pipeline step they execute. In addition to this, Laminator does not define a particular storage strategy to keep the records that are generated, which means that all records need to be agglomerated and chained whenever the verifier wants to check the relationships between the collected data.

3.2.8 VerifiableFL

VerifiableFL [7] is a system capable of training federated leaning models and generating verifiable proofs about about how the model was trained. In addition to that, they also introduce a new abstraction called exclaves, which are integrity-based execution environments, based on TEEs, in order to not rely on the confidentiality of TEE's private keys to guarantee the safety of the data and code stored in it, protecting against side-channel attacks.

Exclave data records (EDRs) are attestation proofs that contain a hash of the measurement of the code used in the exclave, and two key/value data structures, one keeping the hashes of inputs used in a federated learning phase (such as a dataset) and the other storing the hashes of outputs generated in that phase (such as a trained model). These records are signed by the exclave's attestation key to guarantee it they were generated in a secure environment.

To create an analysis of a complete pipeline, an attestation graph is built with all of the EDRs that were created in each phase that was computed. Each of the EDRs whose attestation report has been verified is considered a vertex of the final graph and there is an edge connecting two vertices when at least one of the outputs of one EDR is used as an input in another EDR, creating a dependency between both nodes.

In this work, the authors considered that both a dataset owner or a model owner can be malicious by trying to invalidate the claims recorded in the graph above without being detected. It is assumed that an adversary has access to privileged software including the OS and hypervisor. The graph described above can later be traversed by any user to identify a set of attacks performed by either a malicious dataset owner or model owner, such as:

- Firstly, an attacker could try to change the source code that is loaded in the exclave, for example, to skip a certain phase of the whole federated learning algorithm. Since each node contains a measurement of the code that was loaded in the exclave, an auditor can verify if the loaded code is part of a set of accepted code measurements for each of the tasks, and identify an attack if that is not the case;
- An attacker could also try to modify the data sent by one of the nodes in transit before reaching a different exclave. However, the receiving node would have a hash of an input that does not match any hash of outputs from other nodes, which would be a sign of an attack;
- An attacker could also redirect its output directly to another exclave, skipping any phase that the attacker does not wish to execute. However, this leads to missing edges or vertices in the resulting graph and can still be verified by the auditor;

While VerifiableFL is a strong verifiable provenance capture system for federated learning workflows, it does capture provenance of arbitrary machine learning tasks and cannot enforce additional policies during each task's execution. In addition to this, it does not bind actors to the corresponding tasks that they ran, failing to provide non-repudiation of tasks that are executed by a certain actor.

3.3 Discussion and Comparison

In this section, we compare the systems presented in the related work based on a set of key dimensions that are relevant for the secure collection and management of provenance in machine learning workflows. These dimensions include: provenance granularity, trusted computing base (TCB), record verifiability, storage, and machine learning workflow coverage. Table 1 summarizes this comparison.

3.3.1 Provenance Granularity

Provenance granularity defines the level of detail of provenance that is collected from an application. In addition to that, its choice usually reflects the amount of storage necessary, as fine-grained provenance normally leads to higher amounts of provenance data.

Across the surveyed systems, provenance is mostly collected at a coarse granularity, with only a few frameworks offering fine-grained lineage. ReproZip, MLflow, ProvLake, PraaS, Laminator and VerifiableFL record information at the level of scripts, runs, workflow steps, datasets or model-level properties, but do not track individual data items or internal operations in detail. LIMA is an exception, since it maintains fine-grained lineage DAGs at the level of variables and operations inside ML systems, which incurs in more storage overhead, and noWorkflow collects provenance for individual variables or items in per-phase records. Although fine-grained provenance provides more information about a system, the overhead in storage and execution time, when extracting provenance, is usually higher.

Since capturing fine-grained provenance usually incurs in higher storage and runtime overheads, we chose to adopt coarse-grained provenance collection for each machine learning phase that is executed. This provenance

System	Granularity	TCB	Verifiable	Storage	ML Coverage
noWorkflow	Fine	Entire host	✗	Local SQLite database	Generic scripts
ReproZip	Coarse	Entire host	✗	Local SQLite database	Generic scripts
MLflow	Coarse	Entire host	✗	Central artifact store	Training and evaluation
Lima	Fine	Entire host	✗	Local serialized DAG logs	Full pipeline
ProvLake	Coarse	Entire host	✗	Graph database	Full pipeline
PraaS	Coarse	TEE and runtime	✓	Not specified	Dataset properties
Laminator	Coarse	TEE and runtime	✓	Attestation chain	Training, evaluation and inference
VerifiableFL	Coarse	Exclave and runtime	✓	Graph database	Federated learning training
VeriProv	Coarse	CVM and runtime	✓	Graph database	Training, evaluation and inference

Table 1: Positioning of prior systems by provenance granularity, trusted computing base, storage, record verifiability, and ML workflow coverage.

granularity makes the most sense in our solution since our goal is to capture application level provenance, such as the input and output of the overall workload, instead of focusing in operation-level provenance.

3.3.2 Trusted Computing Base

The trusted computing base of each system determines the components that must be assumed trustworthy during its execution. It is important that it is as small as possible in order to provide more trust to the users using the system, as they will have to trust as little components as possible.

noWorkflow, ReproZip, MLflow, LIMA and ProvLake assume a large trusted computing base that includes the entire host operating system of machines that are used, the runtime that is executed, and their local storage solution where records are kept.

In contrast, PraaS, Laminator and VerifiableFL explicitly reduce the trusted computing base for their provenance extraction due to the use of a hardware-based execution environment, where PraaS and Laminator use TEEs and VerifiableFL uses its exclave implementation, based on TEEs. The rest of the software stack, including the host OS and hypervisor, can be treated as potentially malicious, because integrity and authenticity of records are derived from attestation mechanisms tied to the hardware root of trust. Because of this, users using one of these three systems only need to trust in the execution environment implementation and in the code and data that are used in the application's execution.

One of the main goals of our solution is to provide a way for users to verify that their workload was executed,

without being tampered with, in the intended execution environment in the cloud. Because of that, using a trusted execution environment is essential to provide this guarantee. Although CVMs lead to a larger trusted computing base, since the virtualised operating system also needs to be trusted, they still provide the same security guarantees as regular TEEs and a richer environment to deploy machine learning workloads. For these reasons, we will be using CVMs to deploy our system.

3.3.3 Record Verifiability

After capturing provenance of an application and generating a record, it is important that there are mechanisms to make sure that the captured information was correctly collected without modification. Most systems in this survey do not offer cryptographic verifiability of individual records. noWorkflow, ReproZip, MLflow, LIMA and ProvLake produce logs that can be inspected and queried, but these records are not signed or bound to an attestation service. Because of that, a record can be forged by an attacker to give false information about the execution of an application without being detected.

On the other hand, PraaS, Laminator and VerifiableFL generate records that are directly signed by a TEE or exclave, and bound to an attestation quote, allowing a verifier to check that a record was produced by genuine attested code over specific inputs, in the correct execution environment. This also guarantees that no attacker can create a fake record without detection, giving a strong authenticity assurance of all records that are created to the users.

As discussed in Section 3.3.2 we will be using a CVM to run the user's workload and generate the provenance records. Similarly to PraaS, Laminator and VerifiableFL, these records will be signed by the CVM to generate verifiable attestation records. In addition to this, the actor that performed the execution of the workflow will also be included in the signed record to provide non-repudiation of each machine learning phase that is executed.

3.3.4 Storage

The storage strategies for provenance vary widely and affect how records can be queried and maintained. noWorkflow and ReproZip store provenance in local databases and files on the user's machine, making them suitable for local analysis but less suited for shared, large-scale querying. MLflow relies on a central artifact store that can be queried by users to inspect the experiments they ran. However, establishing relationships for the experiments that are stored has to be performed manually. PraaS does not specify how it stores the verifiable records it produces.

Lima, ProvLake, and VerifiableFL store the provenance information in a directed acyclic graph (DAG), where each node represents an operation or machine learning step, depending on the provenance granularity, and edges represent relationships between nodes. If two nodes are joined by an edge, it signals that an operation or machine learning step uses the output of another node as its input. With this storage representation, it is easier to answer why-provenance and how-provenance queries, since the graph can be efficiently traversed to answer these questions. Although Laminator does not keep a graph provenance store while machine learning steps are executed, it can generate an attestation chain of all records that have been generated. A graph storage solution could be beneficial so that it can be updated as new records are created, without having to rebuild the chain whenever a verification has to be performed.

Motivated by the flexibility of DAG-based representations in systems like ProvLake and VerifiableFL, our design will include a graph-based storage solution to store the verifiable records that are generated. As discussed in Section 3.1.3, this representation allows why-, how-provenance to be naturally presented, which leads to a more natural way to express queries.

3.3.5 ML Workflow Coverage

The systems also differ in how much of the ML lifecycle they natively cover. noWorkflow and ReproZip are domain-agnostic and can be used for ML scripts, but they do not provide ML-specific abstractions or semantics. Since these solutions work without changing the script to be executed, and do not provide a library to instrument it, users would have to manually modify the records to add machine learning specific information. In addition to this, PraaS only concentrates in capturing verifiable dataset properties without covering specific machine learning tasks.

MLflow, LIMA and ProvLake are explicitly ML-focused: MLflow tracks training and evaluation experiments and the models that are used, LIMA captures lineage inside ML systems, capturing fine-grained information about each operation that is performed, and ProvLake supports provenance across all steps of machine learning pipelines. Finally, Laminator spans training, evaluation and inference steps, while VerifiableFL targets federated training workflows without handling other machine learning steps such as the evaluation of a model, or inferences of a deployed model.

Given the importance of supporting the full machine learning pipeline, our solution captures training, evaluation, and inference phases, allowing users to run any of the phases securely while capturing the necessary provenance from each workload that is executed. In addition to these, VeriProv will also be able to capture dataset properties, similarly to PraaS and Laminator.

3.3.6 Policy Enforcement

All of the surveyed systems are capable of either logging what happened in a script that executed (with or without verifiability) or certify properties of specific phases. However, none of these systems are able to let clients declare explicit policies, for example "Only answer the inference query if the model has been evaluated with at least 0.90 accuracy", and to enforce them at runtime when they are not satisfied.

Our system will address this gap by supporting user-defined declarative policies that can be enforced in the CVM before executing the workload itself. This enables the rejection of undesirable machine learning tasks, before they are executed.

In contrast with existing systems, our approach is the first to combine verifiable provenance capture with a binding between actors and the machine learning tasks they execute, and runtime policy enforcement in the context of machine learning workloads deployed in CMVs in adversarial cloud environments.

4 VeriProv

We now propose VeriProv, a system that provides verifiable auditability of non-repudiable machine learning pipelines executed in an adversarial cloud environment. This will be achieved by executing each machine learning phase in a CVM which will enable the generation of attested records containing the inputs, outputs and the identity of the actor that is responsible for executing the corresponding workload. The generated records will be stored as a provenance graph that can later be queried by an auditor and used to enforce user-defined policies before the workload is executed.

Our system extends Laminator, which is already capable of safely verifying dataset properties or train, evaluate and query an AI model in the cloud while collecting verifiable provenance that can be verified in the future, so users can trust the correct software stack and hardware were used for each of the phases.

We extend Laminator in four ways. Firstly, we intend to deploy our solution using a CVM in a public cloud in order to ease the deployment of each machine learning stage. Secondly, we will store the attested records generated by Laminator in a tamper-evident provenance graph, similarly to VerifiableFL, as they are generated, so that auditors can trace dependencies between computations easily and answer why- and how-provenance related queries. We will also add, to the generated records, a binding between an actor and the machine learning phases that it executes, so that it cannot later deny executing each of the phases. Finally, we will add a runtime policy engine that can check for constraints defined by the user before executing the phase, such as “Only answer the inference query if the model has been evaluated with at least 0.90 accuracy”. These policies are checked in the CVM, so that there is a proof that this check was not skipped and that it was performed correctly.

4.1 Threat Model

Our proposed system considers two main adversaries, a malicious cloud service provider whose CVMs are used to deploy our system and the clients’ workloads, and malicious clients that interact with out system by running their workloads in the provisioned CVMs. We considers the following attack vectors:

- **Malicious cloud providers**: a malicious cloud provider can control the whole host operating system and hypervisor and may try to inspect or tamper with the data stored in it. Because of that, we run the

clients' workloads, generate the attestable records and enforce user-defined policies inside CVMs. This allows users to verify that these components executed in the intended secure environment, without being tampered with. In addition to this, a malicious cloud provider could try to generate a fake record. However, it will not be able to do so since it does not have access to the CVM's attestation key.

- **Malicious clients**: firstly, malicious clients can that try to modify or forge provenance records to mislead auditors. Similarly to cloud service providers, they will not be able to successfully do so since they do not have access to the CVM's attestation key. In addition to this, a malicious client could train a machine learning model with a malicious dataset and the action will be recorded in the provenance graph. If later an auditor asks if this user was the one responsible for training the model, he could try to deny it. However, each actor will need to sign the inputs it provides to each phase, which binds their identity to the execution of the phase, making it impossible for them to deny their responsibility. Malicious clients could also try to breach the provenance store used to keep the generated records. However, it is encrypted with a key that is only known by a trusted auditor.

4.2 Assumptions

In this solution we assume that physical and side-channel attacks to confidential virtual machines are out of scope of our work so that we can focus on solving problems regarding auditability and non-repudiation of machine learning tasks running in an adversarial cloud environment. In addition to this, we assume there is a public key infrastructure in place, where each client has a unique private key that is not leaked or lost, giving a way for each client to identify themselves, so that we can safely introduce non-repudiation of the machine learning tasks. Finally, we assume that a secret key can be shared securely with a trusted auditor that is able to perform queries in our provenance store. We assume that the shared secret key is safely stored and is not shared with any other entity.

4.3 Architecture

Our proposed system is built upon Laminator that was described previously. We consider this framework as the base of our solution since it provides a good foundation that is capable of performing model construction and inference while generating verifiable records of these stages. In addition to this, it can also generate records containing verifiable information about the distributional properties of a dataset, such as the ratio of male-to-female rows, and records with properties regarding the evaluation of a trained model, such as its accuracy, fairness and robustness. In Laminator all these properties are verifiable since they are executed in a TEE. However, instead of running this framework with a local machine equipped with a TEE, our goal is to provide the same guarantees by using this framework in the context of confidential virtual machines running in the cloud.

At a high level, our architecture is composed of the following components: CVMs, which act as our trusted execution environment that host the machine learning workload to be executed, ensuring isolation and attestation; a Record Generation component, which is responsible for generating an attestable record for each machine learning phase that is run; a Provenance Store, which is an encrypted graph database that stores the generated records and that can be accessed and queried by a trusted auditor and by the Policy Engine module; and a Policy Engine, a runtime module that checks user-defined constraints against the provenance store before executing the machine learning workload. A representation of architecture can be observed in Figure 2.

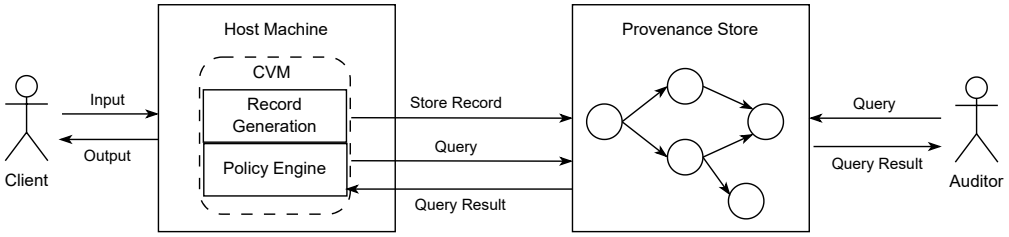

Figure 2: Architecture of our proposed system VeriProv

This initial solution supports the following phases hosted in a confidential virtual machine:

- Calculate distributional properties of a training dataset by providing the training dataset, and the code used to calculate the desired property. This computation will generate a record that contains the hash of the training dataset that was used, the hash of the code that was used and the property that was calculated;
- Create a machine learning model by providing a training dataset and the code necessary to construct the model. This execution returns the trained model to the client and creates a record with the hash of the training dataset that was used, the hash of the code used to train the model and the hash of the model that was produced;
- Calculate evaluation metrics of a previously trained model by providing a test dataset, the model to be evaluated and the code necessary to calculate the desired metric. Example metrics such as the accuracy, fairness or robustness of a model are already provided in Laminator. This computation creates a record with the hash of the test dataset that was used, a hash of the model that was evaluated, the hash of the code that was used to compute the evaluation metric and the metric itself;
- Query a previously deployed model in a confidential virtual machine by providing it with the desired prompt. The computation will return the model's response to the client and generate a record containing the hash of the model that was used to answer the query, the hash of the input provided by the client and the hash of the output generated by the model.

In contrast to Laminator, our system additionaly provides non-repudiation of each computation. The client performing the request must sign the hashes of the inputs that are provided in the phase they are executing with their asymmetric private key. Before performing the desired computation the confidential virtual machine verifies that the signatures are valid and that the inputs provided by the client correspond to the signed hashes. These signed hashes are also included in the final record of any computation, so that each record is bound to the client that executed it. The final record is also accompanied with an attestation quote so that an auditor can verify that the record was generated in the intended environment. This mechanism is not implemented by any of the referenced systems and is what allows VeriProv to bind an actor to all the phases it was responsible for executing, making it responsible for any malicious intent when executing in any of the phases.

Although Laminator by itself can generate the records above, an auditor still needs these records to be stored in a way that enables efficient querying of the records that are collected. As an extension to Laminator, we will store the collected records in a graph-like structure, similarly to ProvLake and VerifiableFL. Each record generated by the phases described above is stored as a vertex in the provenance graph, annotated with the properties needed to describe the corresponding computation. The graph can be implemented either on top of an RDF triplestore, queried with SPARQL, or using a property graph system such as Neo4j, queried with Cypher. To link the different vertices that are created, an edge is added between a pair of vertices if one of the records uses an output of another record as one of its inputs. For example, a vertex containing the description of a query made to a deployed model will have an edge connecting to the vertex that holds a record with information about the model's training, since the model output hash of the latter vertex is also recorded as an input hash in the former. With this approach, the auditor can traverse the generated graph in order to establish links between different computations.

Finally, each vertex of the graph that is built needs to be securely encrypted with a key that is only shared with authorized auditors. With this approach, unauthorized users cannot access the information that is stored in the provenance store. Moreover, if there is the need to remove the permissions of an auditor to access the store, a new key can be generated and provided to the new auditor, as long as the provenance store is decrypted with the revoked key and encrypted again with the new key.

4.4 Policy Enforcement

Policies are declarative constraints that can be specified by a user to obtain additional guarantees about phases that have already been executed. When the responsible confidential virtual machine receives a request, it firstly parses the policies that were defined by the user. To verify each policy, the CVM checks the provenance

graph for the necessary records to audit if the constraints are satisfied. If they are, the execution of the intended phase continues as expected. If not, the execution is halted and the attestation record contains the policies that were violated if further inspection is needed.

We focus on simple stateless policies over existing records, such as existence checks or threshold comparisons. These policies are expressive enough to enforce many useful guarantees, including dataset validation or model evaluation. More complex policies such as temporal constraints are out of scope for this work.

If the user wants to add a set of policies to the request it can send an additional JSON file containing the policies it wants to enforce with the structure defined in Listing 1, where *phase* defines a previous phase in the pipeline that will be checked for the policy enforcement, *key* defines the name of the property to be checked in the policy, *op* defines the operator that will be used to check for the value of the property and *value* defines the threshold for the property that will be checked, which can either be a single value or an interval. The *key*, *op* and *value* are optional parameters so that the user can define a policy that simply checks if a previous phase has already been executed for the model that is going to be used in the current phase. The Policy Engine module is responsible for checking the validity of the specified policies and for parsing and translating the JSON file into valid queries in the query language used to interact with the provenance store.

With this template, users can enforce policies such as the ones seen in Listing 1 and described below:

- "Only answer the inference query if the queried model has been evaluated with at least an accuracy of 0.90";
- "Only train a machine learning model with a dataset D, if the ratio of men to women in the dataset is close to 1";
- "Only answer the inference query if the queried model has been evaluated";

The enforcement of these policies in the confidential virtual machine itself allows the users to trust that the policies were not skipped, and that its computation has not been tampered with.

```
{
    "policies": [
        {
            "phase": "evaluation",
            "key": "accuracy",
            "op": ">",
            "value": 0.90
        },
        {
            "phase": "dataset_property",
            "key": "men_women_ratio",
            "op": "in",
            "value": [0.95, 1.05]
        },
        {
            "phase": "evaluation"
        }
    ]
}
```

Listing 1: Example of a JSON file that defines three different policies

4.5 Provenance Queries

The provenance graph that is built enables auditors to perform a variety of representative queries that cover the main auditability scenarios the system is designed to support and that are not supported by isolated attes-

tations alone. All of the queries can be performed by traversing the graph, recording the relationships between the vertices and verifying the signatures of the records that are stored.

Our system provides a tool that lets an auditor use pre-defined templates that are then translated into queries in the query language used to directly interact with the provenance store. The planned templates are the following:

- *CheckRecord(phase, constraints)*: the CheckRecord query can be used to check if a certain record describing the execution of a machine learning phase exists in the graph. *contraints* is a list of triples, where each triple defines the name of a property, an operator and a value. With this template, an auditor can check if the provenance graph contains a record with specific values. For example, by using CheckRecord("training", [("model", "=", H)]) an auditor can check if there is a record describing the training of a model with hash H;
- *FetchAll(phase, constraints)*: the FetchAll query can be used to fetch all records that describe the execution of a machine learning phase and follow a list of constraints, similarly to the previous query. For example, an auditor can use this query to collect all records describing the evaluation of models with accuracy lower than 0.90;
- *FetchPipeline(record_id)*: the FetchPipeline query can be used to fetch all records that belong to the pipeline of the record with *record_id*. For example, if an auditor runs this query with the *record_id* of an inference record it will return the record describing the training of the model used for the inference, the evaluation records of that model and the dataset property records of the dataset used to train the model.

These templates can be used to answer queries such as:

- Verify that training a model M used a specific dataset D;
- Given a deployed model or an inference that was performed, identify the dataset, training code, and evaluation results that led to its creation;
- Identify all models that were trained using a specific dataset;
- Verify that a client executed a particular training or evaluation phase in case a model is later found to be faulty or biased;
- Verify that a model was evaluated according to a required set of metrics;
- Given a prediction returned to a user, trace back to the training record and verify that the model satisfied certain evaluation properties at training time.

4.6 Use Case

As a particular example, this system can be used to help a team of medical professionals diagnose the patients they are treating. To do so, a diagnose prediction model can be trained using for example the MedDialog dataset [18] [19] which contains conversations between patients and doctors that can be used to predict diseases according to the symptoms shared by the patients. Doing this with our system allows the medical professionals to firstly be assured that the model was trained with the desired dataset and that some properties such as the training dataset having a gender ratio close to 1.0 or the trained model achieving a good accuracy hold. This system also assures that all computations are ran in confidential virtual machines which can prove that the correct code was ran and that the data and code were not inspected or modified during execution. In addition to this, the storage of all queries in the graph allows a medical professional to support the diagnosis provided to a patient by showing that it was supported by a response from a correctly trained model.

5 Evaluation

In this section we present our evaluation plan. Our main goals are to show that our system can be deployed in practice using confidential virtual machines, to characterize the performance and storage overhead introduced by record generation, policy enforcement and show that queries performed to the provenance graph are feasible.

We use Laminator as our baseline since it is a good base system that captures verifiable machine learning properties in a pipeline. However, Laminator does not bind records to actors' identities during the execution of the system, does not store attestation in an explicit provenance graph that can be queried, and does not enforce declarative constraints in runtime. VeriProv is able to guarantee these three shortcomings of Laminator. However, by doing so, we are sacrificing additional performance and storage overhead in our system.

5.1 Record Generation Overhead

We will first measure the overhead introduced by record generation in each machine learning phase supported by VeriProv which are dataset properties analysis, training, evaluation and inference. For each phase, we will run the following configurations: firstly, to establish a baseline, the workload will be executed in the CVM without generating any Laminator or VeriProv records; after that, the workload will be executed with Laminator, generating an attestation record, without non-repudiation guarantees or policy enforcement. Finally, the workload will be executed in VeriProv's full system. For each configuration and phase, we will measure the latency and report the relative overhead of Laminator and VeriProv compared to the baseline.

The record generation overhead should remain acceptable in order to not majorly slow down the important computation of the workload, such as the training of a model. In addition to this, a high overhead can dominate the execution time of quicker phases such as inferences to trained models. Since in real world applications inferences tend to have a high throughput it is of utmost importance that the overhead introduced by the record generation is not noticeable in this case. We aim to generate attestations in a few milliseconds in order to achieve a similar efficiency as Laminator, since this overhead is negligible when compared to the running time of the workload.

5.2 Storage Overhead

With the introduction of the provenance graph to store the records that are created in each phase, we will also measure how much space we are actually spending by storing the information we believe is necessary in each stage. We will be analyzing the amount of storage needed to store all records generated with different values of throughput for each phase. Once again, phases with high throughput such as inferences need to be taken into account, since they can quickly increase the amount of storage that is needed for the whole graph.

We expect the records that are saved to increase linearly with the amount of requests that are executed. Because of that, we expect the memory needed by the provenance store to also scale linearly with the amount of phases that are executed. We might need to use techniques to minimize the memory used in the provenance store without losing any information. An option is to use deduplication, where instead of storing the same data value in a lot of different nodes (e.g. a dataset that is used to train multiple different models), we can store the data value only once, and each node that requires the value contains a pointer to the stored data value.

5.3 Provenance Query Performance

Finally we will also test how efficiently an auditor can receive the response of a query performed against our provenance store. To do so we will test queries similar to the ones presented in 4.5 and measure the amount of time it takes for the auditor to receive the response. The queries to be tested can be divided into two categories:

- *Local queries*: which only need the information from a single pipeline to look up for the answer of the query, resulting in few vertices being traversed. For example, a client might say that he trained a model M with a dataset D and an auditor might want to verify the statement by asking the query "Did model M train with dataset D?". In addition to this, a client might say that he performed an inference to a model M and that M was certainly trained with a dataset that does not respect property P. To verify that statement,

the auditor can ask the query "Does the dataset used to train model M respect property P?". This query needs to firstly find the node related to training model M to find out what dataset was used to train the model, and secondly it needs to find the node related to the correspondent dataset properties to check if property P is respected;

- *Complex queries*: which need information from different pipelines to answer the query. These queries might need complex graph traversals in order to obtain the final answers. For example, queries such as "Find all models that were evaluated with less than 0.90 accuracy" need to traverse the whole graph to give a satisfiable answer.

Since our Policy Engine needs to query the provenance store to verify the policies that are provided, we need to make sure that answering to these queries does not take a lot of time. However, since the Policy Engine considers constraints that check for properties of a single pipeline, they are limited to the use of local queries, which are, in theory, more efficient than complex queries. In order to not worsen the overhead of record generation even further, our goal is for these queries to not take more than some milliseconds. On the other hand, it is reasonable to assume that complex queries submitted by an auditor can take up to a few seconds to be answered.

We will test each class of query with provenance graphs of different sizes, to understand how scalable the queries are with the increase of data in the provenance store. To perform this study we will measure the average time it takes to answer a selected set of queries for each of the classes.

5.4 Policy Enforcement Evaluation

To evaluate our Policy Engine, we will design experiments where policies are intentionally satisfied or violated. Examples include:

- Training a model on a dataset that fails a required property, and verifying that the corresponding training policy is detected as violated and recorded in the verifiable record;
- Issuing inference requests with specific constraints and verifying, via the provenance graph, that the policies applied to each prediction were those declared by the client.

We will measure the additional latency introduced by policy evaluation compared to running the same phases without policies, and we will verify that the policy outcomes encoded in the records match the result that is expected. This will show if our policy engine is both efficient and practical.

6 Work Schedule

The Gantt chart in Figure 3 illustrates the planned schedule for various tasks related to the MSc dissertation.

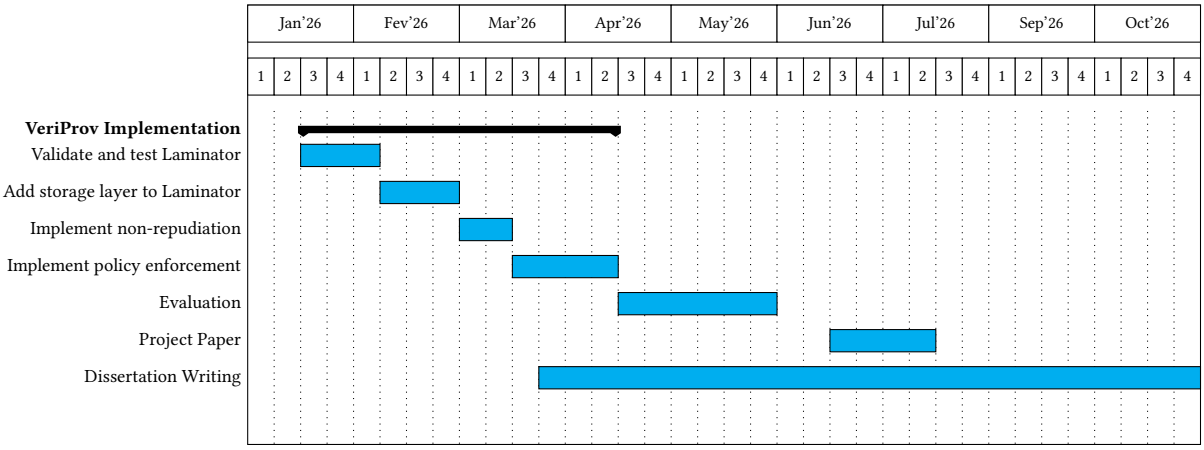

Figure 3: Planned Schedule

7 Conclusions

The growing adoption of AI applications in the cloud, which can be considered an adversarial environment due to the implicit trust placed in cloud providers, makes it essential to ensure the secure execution of machine learning workloads. Because of this, it is also necessary to collect reliable information about the workloads that are executed, to enable a future audit of the relationships between all workloads.

Several existing solutions have explored verifiable provenance collection for machine learning workloads using trusted execution environments, which let users trust that both execution and provenance captured were run securely and without tampering. However, current systems do not guarantee non-repudiation of the phases executed by clients, and they lack mechanisms to enforce user-defined policies policies that give additional guarantees regarding components already computed but that are being used in the current phase.

In this report, we began by introducing the foundations of confidential computing and provenance. We then surveyed the state of the art, analysing systems that capture provenance from general-purpose scripts to full machine learning pipelines. Building on this, we proposed a solution that enables verifiable provenance capture of machine learning pipelines with strong non-repudiation guarantees and policy enforcement using confidential virtual machines. Finally, we outlined an evaluation plan to assess the system's practicality and effectiveness, and planned the schedule for future work.

Bibliography

[1] F. van der Vlist, A. Helmond, and F. Ferrari, "Big AI: Cloud infrastructure dependence and the industrialisation of artificial intelligence," *Big Data & Society*, vol. 11, no. 1, p. 20539517241232630, 2024.

[2] M. Mitchell, S. Wu, A. Zaldivar, P. Barnes, L. Vasserman, B. Hutchinson, E. Spitzer, I. D. Raji, and T. Gebru, "Model Cards for Model Reporting," in *Proceedings of the Second Conference on Fairness, Accountability, and Transparency (FAT*).* Atlanta (GA), USA: ACM, Jun. 2019, p. 220–229.

[3] T. Gebru, J. Morgenstern, B. Vecchione, J. W. Vaughan, H. Wallach, H. D. III, and K. Crawford, "Datasheets for datasets," *Commun. ACM*, vol. 64, no. 12, p. 86–92, Nov. 2021.

[4] M. A. Zaharia, A. Chen, A. Davidson, A. Ghodsi, S. A. Hong, A. Konwinski, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe, F. Xie, and C. Zumar, "Accelerating the Machine Learning Lifecycle with MLflow," *IEEE Data Eng. Bull.*, vol. 41, pp. 39–45, 2018.

[5] R. Souza, L. Azevedo, R. Thiago, E. Soares, M. Nery, M. A. S. Netto, E. Vital, R. Cerqueira, P. Valduriez, and M. Mattoso, "Efficient Runtime Capture of Multiworkflow Data Using Provenance," in *2019 15th International Conference on eScience (eScience)*, San Diego (CA), USA, Sep. 2019, pp. 359–368.

[6] V. Duddu, L. J. Gunn, and N. Asokan, "Laminator: Verifiable ML Property Cards using Hardware-assisted Attestations," in *Proceedings of the Fifteenth ACM Conference on Data and Application Security and Privacy (CODASPY).* Pittsburgh (PA), USA: ACM, Jun. 2025, p. 317–328.

[7] J. Guo, K. Vaswani, A. Paverd, and P. Pietzuch, "VerifiableFL: Verifiable Claims for Federated Learning using Exclaves," *arXiv preprint arXiv:2412.10537*, 2025.

[8] R. Souza, L. Azevedo, V. Lourenço, E. Soares, R. Thiago, R. Brandão, D. Civitarese, E. Brazil, M. Moreno, P. Valduriez *et al.*, "Provenance data in the machine learning lifecycle in computational science and engineering," in *2019 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS).* Denver (CO), USA: IEEE, Nov. 2019, pp. 1–10.

[9] P. Buneman, S. Khanna, and T. Wang-Chiew, "Why and where: A characterization of data provenance," in *Proceedings of the 8th International Conference on Database Theory (ICDT)*, J. Van den Bussche and V. Vianu, Eds. Berlin, Heidelberg: Springer-Verlag, Jan. 2001, pp. 316–330.

[10] M. Herschel, R. Diestelkämper, and H. Ben Lahmar, "A survey on provenance: What for? What form? What from?" *The VLDB Journal*, vol. 26, no. 6, p. 881–906, Dec. 2017.

[11] J. Freire, D. Koop, E. Santos, and C. T. Silva, "Provenance for Computational Tasks: A Survey," *Computing in Science and Engg.*, vol. 10, no. 3, p. 11–21, May 2008.

[12] B. Pan, N. Stakhanova, and S. Ray, "Data Provenance in Security and Privacy," *ACM Comput. Surv.*, vol. 55, no. 14s, Jul. 2023.

[13] L. Murta, V. Braganholo, F. Chirigati, D. Koop, and J. Freire, "noWorkflow: Capturing and Analyzing Provenance of Scripts," in *Revised Selected Papers of the 5th International Provenance and Annotation Workshop on Provenance and Annotation of Data and Processes - Volume 8628 (IPAW).* Cologne, Germany: Springer-Verlag, Jun. 2014, p. 71–83.

[14] F. Chirigati, D. Shasha, and J. Freire, "Packing experiments for sharing and publication," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD).* New York (NY), USA: ACM, Jun. 2013, p. 977–980.

[15] F. Chirigati, R. Rampin, D. Shasha, and J. Freire, "ReproZip: Computational Reproducibility With Ease," in *Proceedings of the 2016 International Conference on Management of Data (SIGMOD).* San Francisco (CA), USA: ACM, Jun. 2016, p. 2085–2088.

[16] A. Phani, B. Rath, and M. Boehm, "LIMA: Fine-grained Lineage Tracing and Reuse in Machine Learning Systems," in *Proceedings of the 2021 International Conference on Management of Data (SIGMOD/PODS)*. Virtual Event, China: ACM, Jun. 2021, p. 1426–1439.

[17] I. E. Akkus, I. Rimac, and R. Chen, "PraaS: Verifiable Proofs of Property as-a-Service with Intel SGX," in *2024 IEEE European Symposium on Security and Privacy Workshops (EuroS&P)*, Vienna, Austria, July 2024, pp. 199–207.

[18] X. He, S. Chen, Z. Ju, X. Dong, H. Fang, S. Wang, Y. Yang, J. Zeng, R. Zhang, R. Zhang *et al.*, "MedDialog: Two Large-scale Medical Dialogue Datasets," *arXiv preprint arXiv:2004.03329*, 2020.

[19] ——, "MedDialog: Medical Dialogue Dataset," Kaggle, https://www.kaggle.com/datasets/xuehaihe/medical-dialogue-dataset/data, 2020.