Enabling Secure and Efficient Data Analytics Pipeline Evolution with Trusted Execution Environment

Haotian Gao
National University of Singapore
NUS Research Institute in Chongqing
gaohaotian@comp.nus.edu.sg

Cong Yue
National University of Singapore
yuecong@comp.nus.edu.sg

Tien Tuan Anh Dinh
Deakin University
anh.dinh@deakin.edu.au

Zhiyong Huang
National University of Singapore
NUS Research Institute in Chongqing
huangzy@comp.nus.edu.sg

Beng Chin Ooi
National University of Singapore
ooibc@comp.nus.edu.sg

ABSTRACT

Modern data analytics pipelines are highly dynamic, as they are constantly monitored and fine-tuned by both data engineers and scientists. Recent systems managing pipelines ease creating, deploying, and tracking their evolution. However, privacy concerns emerge as many of them are deployed on the public cloud with less or no trust. Unfortunately, the unique nature of pipelines prevents the adoption of existing confidential computing techniques with different computational patterns and large performance overhead. Being a potential approach, trusted execution environments (TEEs) are efficient in protecting the confidentiality and integrity of data and computation. However, fast-changing pipelines with latency requirements bring the challenge of reducing the cold start overhead — the main bottleneck in the latest TEE. To support end-to-end private pipeline evolution, we present SecCask, a TEE-based data analytics pipeline management system. SecCask overcomes the problems of a naive design that isolates complete pipeline execution in one enclave by administering enclaves and runtimes. To reduce cold start overheads, our approach consists of reusing trusted runtimes for different pipeline components and caching them to avoid the cost of initialization. We leverage the latest Intel SGX to conduct experiments on representative workloads. The results demonstrate that SecCask reduces the total execution time by 68.4% compared to not reusing, is faster than running all components in one enclave, and incurs a modest average performance overhead of 29.9% over insecure baselines.

PVLDB Reference Format:
Haotian Gao, Cong Yue, Tien Tuan Anh Dinh, Zhiyong Huang, and Beng Chin Ooi. Enabling Secure and Efficient Data Analytics Pipeline Evolution with Trusted Execution Environment. PVLDB, 16(10): 2485 - 2498, 2023.
doi:10.14778/3603581.3603589

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/seccask.

Proceedings of the VLDB Endowment, Vol. 16, No. 10 ISSN 2150-8097.
doi:10.14778/3603581.3603589

1 INTRODUCTION

Modern data analytic pipelines are not static. While traditional data pipelines are created by expert data engineers for specific business (OLAP) applications, many of today's pipelines are created by data scientists and are more exploratory in nature. In particular, they are fine-tuned over many iterations, constantly monitored, and updated even in production [8, 48, 60]. For example, the pipeline is updated when new data sources are used when new features are discovered, when the data distribution changes, or when there are new runtime libraries.

The ease of managing complex, fast-changing analytics pipelines drives work in pipeline management systems [6, 10, 32, 35, 58, 59, 65] that automate the creation and execution of pipeline components, and track change in data and computation using version control. Current pipeline management systems have become more efficient and user-friendly as many of which are being offered as cloud services, e.g., MLflow, Amazon SageMaker, and Azure Machine Learning. However, they are not designed with privacy in mind. In particular, end users must trust the entire system that manages their data and performs computations on it. This assumption is not tenable, especially in the cloud settings, for two reasons. First, the system can be compromised either through software vulnerabilities or insider threats. Second, users may not be able to upload raw data or components, considering privacy regulations such as GDPR, or concerns over intellectual properties.

Hence, protecting data and computation privacy in an untrusted environment becomes an important while challenging problem. Different techniques are proposed to address it, including differential privacy [1, 46], homomorphic encryption [7, 38, 42], multi-party computation [63, 68], etc. However, it is not suitable to adopt these techniques in a broader pipeline management process. First, most of them mainly target machine learning (ML) modeling, which is only one stage of modern pipelines [8]. Second, they are designed with specific communicative and computational patterns, and comprehensive changes in the development workflow are required to work with these patterns. Third, they suffer from either high communication or computational overhead (for multi-party computation and encryption-based techniques, respectively), or loss of accuracy (for differential privacy techniques), which is not practical for either large or small data setups.

One approach to providing privacy without the above limitations is to leverage trusted execution environment (TEE) techniques, e.g., Intel SGX, Arm TrustZone, and Arm Confidential Computing. TEE protects the confidentiality and integrity of both data and computation against powerful software adversaries including the malicious operating system. Existing works demonstrate the viability of using TEE for various applications despite hardware limitations such as memory size [15, 22, 26, 39, 45, 66]. The latest release of Intel SGX significantly loosens the memory limitation, being able to run memory-intensive application services [16, 34]. Although this makes it possible to run pipelines directly inside TEEs, we observe that it does not meet our performance goal. In particular, TEEs suffer from the *cold start* problem, where the cost of initializing a trusted runtime from scratch is high. This type of overhead can become significant in a typical pipeline evolution, making it a major issue for efficiency. First, during the early phases, many procedures are short in execution time and are frequently, repeatedly executed, as data scientists are exploring the parameter space [48, 61, 65]. Second, some pipeline components such as data inference and feature extraction require low latency.

To target the performance limitations of pipeline privacy preservation and provide an end-to-end solution for pipeline evolution in an untrusted environment, we present SecCask, a TEE-based secure and efficient pipeline management system. It allows different user roles to collaborate on securely creating, scheduling, monitoring, and executing fast-changing data pipelines. For secure file storage, we propose *Encrypted Filesystem* (EncFS) for modules in SecCask to overcome the limitations of existing TEE file sealing solutions. To reduce overheads from the cold-starting, SecCask supports runtime reusing by executing components on *compatible* cached workers. We propose a novel caching policy, called *Pipeline-aware Caching* (PAC), which is based on access patterns across pipeline components. Specifically, PAC analyzes user pipeline submission activities and predicts probabilities of subsequent component versions based on two locality properties, *spatial locality* and *temporal locality*. By caching the runtimes that could serve components more likely to execute for the subsequent pipeline, the system reduces the costly cold starts for long-term pipeline evolution.

In summary, we make the following contributions:

- We present SecCask, a secure and efficient data analytics pipeline management system based on TEEs. Unlike existing works using TEEs for secure ML systems, SecCask manages end-to-end, non-linear data analytics pipeline evolution, including components with, e.g., preprocessing, training on small and large datasets, and inference. To overcome the limitations of existing file sealing solutions for TEE, we provide SecCask's modules with a new file sealing middleware called EncFS.
- We provide a formal definition of the cold start problem in the context of TEE-based pipelines. We then propose a runtime reusing strategy and a novel PAC policy to reduce cold start overheads. It exploits two locality properties, i.e., temporal locality and spatial locality, extracted from the pipeline history.
- We evaluate the effectiveness of PAC through extensive benchmarks on worker scalability, cold start overheads, EncFS, and PAC. The results show that PAC is effective in reducing the impact of cold starts, achieving up to 74.28% more cold start elimination over the baselines with generic caching policies.
- We test the efficiency of SecCask through two representative workloads as case studies, where the pipelines come from existing works. The results show that PAC helps reduce the average execution time by 68.4%, enabling SecCask a modest overhead of 29.9% over the insecure baselines, with small EPC overheads.

The remainder of the paper is organized as follows. Section 2 presents the background on pipeline management systems and TEEs. Section 3 discusses the threat model, goals and challenges, and the design of SecCask. Section 4 discusses the cold start problem migrated from serverless computing, provides the problem definition in TEE, and describes PAC. Section 5 details the implementation of SecCask. Section 6 presents the evaluation results. Section 7 reviews the related work before Section 8 concludes.

2 BACKGROUND

2.1 Data Analytics Pipeline

A *pipeline* represents a series of tasks or procedures serving a data analytics objective, consisting of various *components* (or phases), for example, data gathering, cleaning, preprocessing, model training, and deployment. The output of one component can become the input of another. The components are executed in a sequential or pipelined manner, as the intermediate results are transformed from the original form to the final output. A pipeline can be linear, that is, components have at most one input and one output, or the components with multiple inputs or outputs can form a directed acyclic graph (DAG). A pipeline management system automates the creation and management of pipelines.

Most modern pipeline management systems focus on machine learning pipelines. For example, ModelDB [58] allows users to specify parameter-metric records, MLflow [10, 65] supports parameter injection and framework dependency, while ModelOps [20] supports quality control metrics with a flexible system design. ModelHub [35] uses versioning to track model lineage, and MLCask [32] offers non-linear versioning semantics (e.g., branch and merge) and a pipeline search tree for efficient merging. These systems, however, do not protect the security of the data and its computation.

2.2 Trusted Execution Environment

A trusted execution environment (TEE) ensures the confidentiality and integrity of the data and computation inside the environment from outside attackers. A TEE can be provisioned by hardware, software, or a combination thereof. In this work, we focus on utilizing hardware-based TEEs, in particular ones based on Intel SGX. However, other TEEs, such as those based on Arm TrustZone or Arm Confidential Computing, can also be used.

Intel SGX supports TEEs in the form of *enclaves* [11]. A user application can create an enclave, and provision private data and computation to it. The enclave should secure the processing against powerful attackers, including malicious operating systems and physical attackers. The CPU provides a series of instructions to create, enter, and exit the enclave. Enclaves on the CPU share a protected memory region called Enclave Page Cache (EPC). Until recently, the system-wise hard limit on EPC is 128MB. The latest release of SGX, however, extends this limit to 64GB [16, 34].

The security model of SGX makes it challenging to port existing applications to run inside enclaves. To address this problem, there are several *shielding frameworks* [4, 44, 49, 50, 56] that allow running unmodified applications inside enclaves. The framework runs in an enclave, transparently handles requests for operating system (OS) services, and verifies the OS responses before returning them to the application. SecCask uses Gramine [56] as it is the most feature-rich. Other frameworks, such as Occlum, have limited support for commonly used programming languages and packages.

3 DESIGN

3.1 Threat Model

The goal of the adversary is to learn the raw data or to tamper with the pipeline execution. SecCask assumes that it has full control of the software stack in the cloud, either via insider threats or software vulnerabilities. Also, the adversary can compromise the cloud file storage to perform attacks on any system metadata, owners' uploaded components, and intermediate results of component executions. This cloud software stack is mostly closed-source and represents a significantly larger attack surface. However, the process running in the enclave, including both the shielding framework and the component code, is trusted not to violate security. For example, the runtime does not contain exploitable vulnerabilities and offers secure isolation between different library executions. We note that this is a common assumption in systems that leverage TEE [37, 41, 45, 64]. Furthermore, the attack surface of libraries can be reduced by software verification or state-of-the-art hardening techniques, e.g., software fault isolation, data flow integrity, and memory-safe system languages. In addition, the attacker cannot compromise the hardware protection of the TEEs. The remote attestation procedure by Intel SGX is trusted, and we assume that the owners and the users know the identities of the trusted modules, e.g., by verifying their public source code.

SecCask does not consider denial-of-service attacks, which are possible because the cloud provider can refuse to run the enclaves or trigger random errors during the component execution. We assume that such attacks can be detected by the end users of the system. Side-channel attacks (e.g., those based on enclave memory access patterns [9, 57]), membership inference [51], and model inversion attacks [36] are also out of scope.

3.2 Goals and Challenges

Our goal is to secure data analytic pipelines with less overhead. In particular, the system ensures the privacy of inputs, outputs, intermediate data, and management data, e.g., stages and evolution history, for pipelines. To achieve these goals with performance addressed, we leverage Intel SGX to create TEEs to secure pipelines.

A naive design using SGX is to create one enclave per user per pipeline, where the user attests and provisions the enclave with the keys for decrypting data. Upon submission of the pipeline via a secure channel, all pipeline components are executed within the enclave. However, this design has two limitations. First, there is no opportunity for sharing pipeline components between different users, as each enclave holds the entire pipeline. Second, the available EPC is limited, and the pipeline must fit into EPC to avoid paging that severely affects performance. However, different pipeline components may require different packages, where all packages must load at the execution's start, increasing EPC usage. Additionally, Python, the most prevalent runtime for data analytics and ML tasks, cannot unload packages. Consequently, components using different package versions cannot both be executed.

Our approach to addressing the limitations of this naive design is to break up the pipeline into smaller components and run them in separate enclaves. There are two challenges with this design. First, the data flows between enclaves must not leak data. Second, the cold start overheads, which are needed to prepare the runtimes, can be large. These overheads comprise enclave creation, runtime initialization, and package loading, where the first is proportional to the maximum heap size [16] that is directly related to the memory consumption of the running components. To address the first challenge, SecCask ensures that data flow between enclaves is encrypted, and only the intended enclave can decrypt the data. For the second challenge, we design a novel enclave sharing and caching mechanism that reduces the number of cold starts.

3.3 SecCask Overview

SecCask is a performance-oriented trusted data analytics pipeline management system for untrusted clouds (Figure 1). It involves four roles: dataset owners, library owners, users, and the cloud service provider. Owners prepare *component manifests* summarizing data or the required runtime, encrypt them together with component files, and upload them to the cloud storage. They then attest and establish a secure channel with a trusted *key delivery service* (KDS) and send component keys. Users attest to the *coordinator* before submitting pipeline structures. The workspace handles pipeline evolution, metadata, and performance metrics, while the coordinator schedules pipeline execution over workers. New workers are initialized with the corresponding component manifest's runtime and retrieve keys from the KDS. All entities, including the coordinator, KDS, workspace, and workers, run inside SGX enclaves.

Pipeline management. Pipelines' metadata and performance metrics, stored in the workspace, are tracked and updated by the coordinator's workspace manager. SecCask inherits both workspace and its manager from MLCask [32], which supports collaborative pipeline evolution and efficient non-linear branching and merging.

Coordinator. Being the core module, the coordinator handles the pipeline submission requests from users. Internally, it monitors the running workers, handles requests for new workers, performs worker-component compatibility checks, sends component manifests to workers to trigger component execution, and manages the metrics to the workspace through the workspace manager.

Active worker. In SecCask, each library component contains the code run over some data. It is done by an active worker, managed by the coordinator. Each worker retrieves the dataset or the output of the previous component, executes the component code, securely persists its output to the storage engine, and sends metadata and performance metrics generated from the execution process back to the coordinator. The workers can be created on different nodes to mitigate the EPC limitation.

Cached worker. To reduce component execution overheads due to cold starts, different components can reuse a worker. A cached worker is one that finishes a component execution task. Instead of being terminated, it is marked as inactive and then handed over

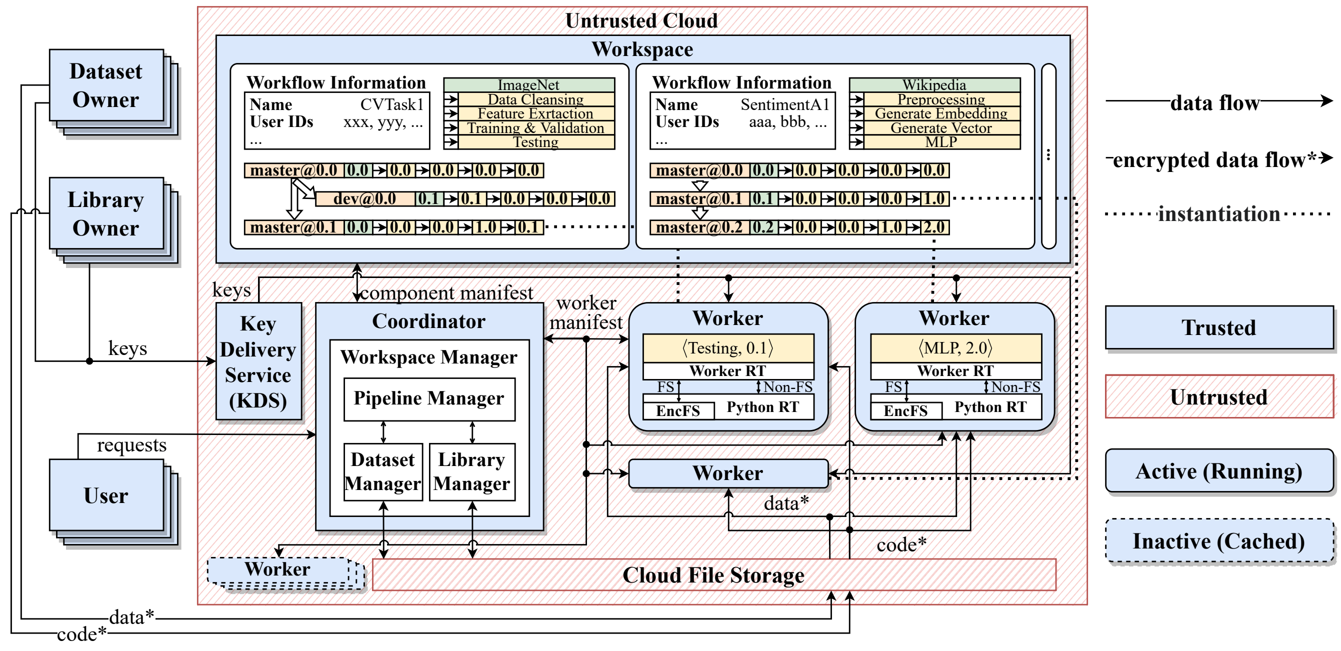

Figure 1: SecCask's architecture.

to the coordinator. When a compatible component is found, the coordinator activates it by sending the component manifest.

Storage service. The system relies on untrusted cloud storage infrastructure to persist pipeline metadata and owners' component files, which is vulnerable to unauthorized access. We implement a utility that encrypts component files with EncFS, discussed in detail in Section 3.5. The owners run this utility locally to encrypt their components and upload them to the untrusted cloud storage, with their keys sent to KDS, dispatched on behalf of the owners.

3.4 Manifest Compatibility

In SecCask, a *manifest* contains the metadata of a specific entity. In particular, it can represent a component or a worker. The system uses information on manifests to determine whether a given component can be executed by a given worker.

A dataset manifest contains the owner-specified dataset name and a version assigned by the workspace manager. Since a dataset is only used as input for another component, we do not consider its compatibility with a worker. A library manifest specifies its required environment with the package dependencies. The compatibility of a library with a worker is determined by the coverage of APIs, represented by the package versions.

A worker can be reused to execute a component when the worker is *compatible* with the component. We define this compatibility as the condition that the worker is able to call all its required APIs without state changes to its manifest by loading packages.

Definition 3.1 (Worker-Component Compatibility). Given a worker w with a set of modules M_w imported, each module $m \in M_w$ belongs to a package $K(m)$ with version $V(K(m))$. For a component f which requires a set of packages K_f with each package $k \in K_f$ having version $V(k)$, worker w is compatible with component f, defined by $\pi(f, w)$, when

$$\pi(f, w) := \forall k \in K_f, \exists m \in M_w, V(K_f) \overset{c}{=} V(K(m)),$$

where $\overset{c}{=}$ means two versions are compatible.

Two types of version compatibility are available. *Exact Version Match* regards two versions compatible only when they are identical. However, this can be too restrictive, especially when upgrading packages does not affect APIs. To tackle this, the more fine-grained *Semantic Version Match* is used to capture API-level compatibility. Packages adopting this matching strategy must be versioned with conventions to explicitly indicate API-level compatibility.

Definition 3.2 (Semantic Version Match). Two versions defined by [43] with the format MAJOR.MINOR.PATCH are semantically compatible, namely $a \overset{c}{=} b$, if

$$(\text{MAJOR}(a) = \text{MAJOR}(b)) \wedge (\text{MINOR}(a) \leq \text{MINOR}(b)).$$

Note that $\overset{c}{=}$ is not commutative for semantically compatible.

The examples in Figure 2 demonstrate the compatibility check results for different scenarios. For Exact Version Match, the runtime must contain packages with the same versions as specified in the manifest. If a package is missing or has a different version, the check fails. For Semantic Version Match, the runtime must contain packages with compatible versions as defined by Definition 3.2. The system extracts a list K_w from the runtime and compares it with the manifest to perform the check. This list is referred to as the *active packages* and is discussed in detail in Section 5.1.

Discussion. The component owner defines the manifest, and therefore is responsible for ensuring worker-component compatibility. In particular, for semantic versioning, the owner needs to check

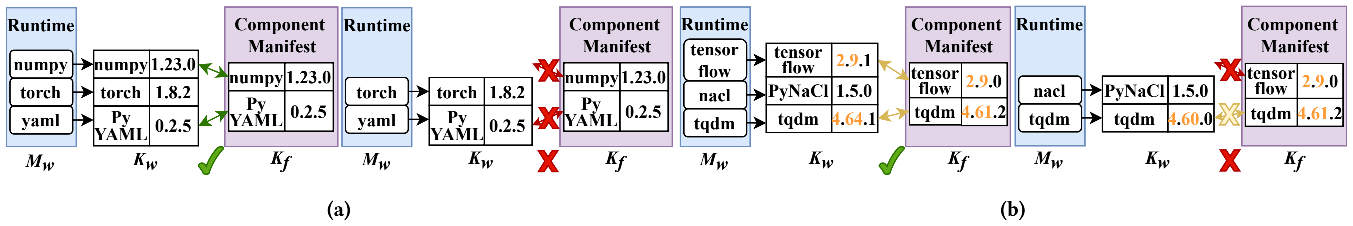

Figure 2: Examples of compatibility with (a) Exact Version Match (b) Semantically Compatible. K_w is the active package list.

that all packages and their dependencies adopt semantic versioning. We note that this versioning is increasingly common in practice [13, 14, 40], therefore we can expect more reusing of workers. For packages that violate compatibility despite using semantic versioning [40, 67], we assume the users can detect incompatibility issues during the execution, and then change the manifest to use exact versioning. The exact version is automatically derived from the last successful execution, as recorded in pipeline history.

3.5 Encrypted File Storage

The main challenge in the storage module of SecCask is to make access to encrypted files transparent to the application. In particular, the system requires no changes to the code that runs on a trusted, non-enclave environment. To achieve this, SecCask introduces a middleware layer called *Encrypted Filesystem* (EncFS). It sits between the interpreter (Python) and the shielding framework, intercepting filesystem APIs and transparently performing authenticated encryption and decryption[1]. EncFS supports random access by using AES-CTR, where the counter is based on the offset of the file divided by the block size. Combined with the Galois message authentication code (MAC) and the Encrypt-then-MAC approach, the authenticated encryption mode becomes AES-GCM. EncFS computes one MAC for each page, respecting the page-level access pattern of OS file maps. Hence, authenticated file accessing is performed on the pages that cover the given bytes.

For each encrypted file, there is a separate *MAC file* for storing the MACs of encrypted pages. For each write, EncFS identifies the involved page IDs, then writes the encrypted pages to the file, and writes each MAC at the offset $ID \times Ms$ where Ms is the MAC size. A read operation first retrieves the corresponding encrypted MACs, then verifies the decrypted against those of the encrypted pages.

3.6 Key Delivery

In SecCask, the owners encrypt their components and upload them to an untrusted storage service. The workers running EncFS are provisioned with the appropriate decryption keys. A special enclave, called Key Delivery Service (KDS), receives the decryption keys and dispatches them to the correct workers.

Encryption. Components are encrypted by AES-GCM. Specifically, the owner generates a random key and IV as follows. The owner's password is hashed, using PBKDF2-HMAC-SHA512, resulting in a 512-bit random number. The first k bits (where $k \in \{128, 256\}$) are used as the AES key, and the subsequent 96 bits are the IV. The encryption is performed in parallel, at the page granularity.

KDS. The trusted workers need to load and decrypt components. The decryption key can be provisioned to the worker directly by the owner. However, this approach requires the owner to be online during the execution, and it does not scale for a large number of workers. The Key Delivery Service (KDS) enclave maintains the decryption keys and only gives them to trusted workers. The owner first sends the keys to KDS, using the attestation process described below. The KDS then uses an SGX feature called sealing to store the decryption keys on untrusted storage. Sealing ensures that the keys are encrypted and can only be decrypted by the KDS enclave.

3.7 Attestation

Remote attestation is an important concept and procedure for building trust with TEE. In short, a hardware-signed *quote* packing a computed *measurement* of an enclave is required to be sent to a remote verifier to convince it that the enclave is trustworthy [11]. To let the system with multiple modules obtain the trust of user entities, SecCask builds a custom attestation model with minimal interference to the management process while addressing all security requirements, illustrated by Figure 3.

There are three main attestation steps in SecCask. The first is between the owners and KDS after encrypted components are uploaded to the cloud. The owner checks that it is communicating with the trusted KDS that faithfully manages and delivers the keys. The measurement of KDS is known to the owner. SecCask uses RA-TLS [27] for this attestation, which establishes a TLS channel through which the decryption keys are sent.

The second attestation is between the users and the coordinator. The user checks that it is communicating with a trusted coordinator that correctly updates the pipeline history, schedules the pipeline execution, and returns the trained models and metrics. The measurement of this trusted coordinator is known to the user. RA-TLS is used to establish a secure channel between the user and the trusted coordinator. The user then uploads the pipeline via this channel.

The third attestation is done by KDS and the coordinator to check that they are communicating with the trusted workers. The measurements of the trusted workers are known to both KDS and the coordinator. The KDS uses RA-TLS to attest the workers, and so does the coordinator. If successful, the execution task and decryption keys are sent to the worker, which loads the components from the untrusted storage, and then starts the execution.

The TLS channels between workers, KDS, and coordinator, are reused for different tasks, reducing remote attestation overheads.

[1]We note that this layer is similar to Intel Protected File System, but the latter is not fully supported by the shielding framework.

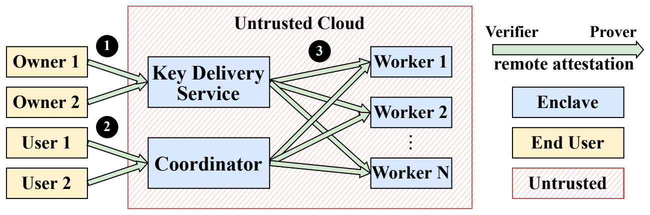

Figure 3: SecCask's attestation model. Three main attestation steps include (1) owners' delivery of component keys (2) users' pipeline submission (3) workers' task and key receiving from KDS and the coordinator.

3.8 Security Analysis

We now analyze how SecCask achieves the privacy goal discussed in Section 3.2, given the threat model in Section 3.1.

Metadata. The metadata in SecCask includes workflow data, namely the name, the pipeline DAG, and versions of the pipeline history. It also contains the outputs of the pipeline execution, including the intermediate results as well as other system parameters. The metadata is maintained by the coordinator enclave, either in its enclave memory or in the encrypted files. The key for decrypting the encrypted metadata is derived from the enclave's sealing key, therefore only the coordinator enclave can decrypt it.

The coordinator attests to the other enclaves, then establishes secure channels with them before transmitting metadata through the channels. This ensures that the data exchanged between enclaves are protected, and only the correct enclaves can decrypt the data.

Worker. A worker loads and decrypts the component with the key provisioned by KDS. It also exchanges encrypted metadata with the coordinator. Both KDS and the coordinator attest to the worker before sending the key or metadata. The security of the attestation protocol ensures that only workers that are allowed by the pipeline can receive the key and metadata. Since we assume that the enclave for a worker is secure, all data and computation in it are protected.

Components. The component owner performs attestation with KDS before sending it the key to decrypt the component on the untrusted storage. The key stored in the enclave is sent only to attested workers about to execute the corresponding component. Since KDS is trusted, the adversary cannot introduce a malicious component to the pipeline, as it will not be accepted by the coordinator. Furthermore, after each execution, the worker runtime rolls back to its initial state. Therefore, it can be reused for other tasks without leaking information from past executions.

Side channels. SecCask starts modules and workers in separate enclaves. Although the data communicated between them are encrypted, we acknowledge that the necessary cross-enclave interactions introduce undesirable side-channel leakages. Compared to the naive design of running the entire pipeline in one enclave (Section 3.2), the adversary, by observing cross-enclave communication, can learn two more things. First, it learns the structure of pipelines by observing how many enclaves are created, and how data flows between them. Second, for each component, it learns the resource consumption profile and the sizes of the output. These can be used, e.g., to infer the type of each component: the training component is CPU intensive, the feature extraction component produces smaller output than its input, etc. Closing this channel, however, is challenging. In particular, to hide output sizes, it is necessary to pad the output to the maximum output size of all components. To hide resource consumption, techniques such as oblivious data access and computation padding are required. To hide the data flow, it may be necessary to support oblivious or delayed data access, which incurs significant overheads [21]. We leave the question of how to use them in SecCask to future work.

4 MITIGATING COLD START

In this section, we describe the cold start problem in TEEs and propose a novel caching policy called Pipeline-aware Caching (PAC) that reduces cold starts while maintaining high resource utilization.

4.1 Cold Start

Starting an execution environment takes time. *Cold start* measures the time taken from when the system starts the execution environment to when it is fully ready to execute, i.e., the latency of system initialization. This depends on the environmental requirements of the user, e.g., the package dependencies.

Our approach to reducing cold starts for TEEs is inspired by works in a slightly different setting, namely serverless computing. A serverless platform starts fresh runtime to serve user functions when the workload increases, which includes starting a new container and loading dependent packages. However, most user requests are short-lived and the cold start overhead dominates the end-to-end latency [47]. Approaches to reducing it include maintaining a pool of warm containers that are ready to handle request spikes [31, 52, 54], predicting future loads and provisioning the runtime in advance based on user history [12, 19, 47], lightweight isolation that reuses dependencies and runtimes [2, 3, 17], etc.

4.2 Worker Reusing Objective

Reuse of workers effectively reduces the cold start overhead when more compatible workers are predicted and cached for incoming components. Hence, the objective is to maintain a worker pool that maximizes the probability of having a compatible worker for a new component, which performance is determined by the caching policy. To this end, we introduce representations of pipelines, their history, and other prerequisite information to predict, before specifying the worker reusing objective.

Pipeline. In a data analytics pipeline, the data is transformed through a sequence of components. More precisely, a *pipeline* is defined as a DAG $G = \{\mathcal{F}, \mathcal{E}\}$, where $f \in \mathcal{F}$ represents a component and $e \in \mathcal{E}$ represents the data flow between consecutive components. Given a component f, its succeeding components (which accept the output of f) and preceding components (which feed their outputs to f) are defined as $suc(f)$ and $pre(f)$, respectively. A pipeline G preserves the following relationship:

$$\forall i, j, \left(f_i, f_j \in \mathcal{F} \wedge e_{ij} \in \mathcal{E}\right) \rightarrow \left(f_j \in suc(f_i) \wedge f_i \in pre(f_j)\right).$$

A component f_i can either be a *dataset* containing the data, or a *library* performing transformation $y = f_i(x \mid \theta_i)$ over the outputs of its preceding component x with the parameter θ_i. Each f_i is assigned a semantic version $v(f_i)$ of the form ⟨`schema.increment`⟩, where `schema` represents the version of output schema which must be increased once changed, and `increment` denotes the algorithm-level changes not affecting the output schema.

Pipeline history. In order to achieve the objective, the history of the users' submitted pipelines can be important to realize an effective worker reusing policy. We denote **H** as a sliding window of pipeline submission history. It contains M pipelines with timestamps from $i - M + 1$ to i:

$$\mathbf{H}^{(i)} = \begin{pmatrix} \mathbf{p}^{(i-M+1)} & \mathbf{p}^{(i-M+2)} & \cdots & \mathbf{p}^{(i)} \end{pmatrix},$$

where $\mathbf{p}^{(i)}$ contains the components' version vector at i, i.e.,

$$\mathbf{p}^{(i)} = \begin{pmatrix} v(f_1)^{(i)} & v(f_2)^{(i)} & \cdots & v(f_j)^{(i)} & \cdots \end{pmatrix}^\top.$$

Each component in a pipeline is given an index, and we define a partial order on the indices as follows.

Definition 4.1 (Order of Component History). Given a pipeline **p**, the following holds:

$$\forall f_i, f_j \in \mathcal{F}, (i < j) \rightarrow (f_i \prec f_j),$$

where $\prec$ satisfies

$$\forall i, j, (f_i \prec f_j \land e_{ij} \in \mathcal{E}) \rightarrow (f_j \in suc(f_i) \land f_i \in pre(f_j)).$$

Version score matrices. Components stored in the system have different versions. We capture the probability of a component f_j having a specific version using a *version score matrix* (VSM), denoted by $\mathbf{S}_j$. The matrix covers all possible versions of f_j, where each element s^j_{mn} denotes the probability that the next occurrence of f in a new pipeline has version $\langle m.n \rangle$. More specifically,

$$\mathbf{S}_j = \begin{pmatrix} s^j_{00} & s^j_{01} & \cdots & s^j_{0d} \\ s^j_{10} & s^j_{11} & \cdots & s^j_{1d} \\ \vdots & \vdots & \ddots & \vdots \\ s^j_{c0} & s^j_{c1} & \cdots & s^j_{cd} \end{pmatrix},$$

where $c = \max\{\texttt{schema}(f_j)\}$, $d = \max\{\texttt{increment}(f_j)\}$.

Worker reusing objective. Given the definitions of VSM and pipeline history, we observe the transformation of the VSM $\mathbf{S}_j$ at the timestamp t_i given the pipeline history until t_{i-1}. The probabilities of component versions at the next timestamp are only based on those at the current timestamp and the pipeline history, i.e. the next $\mathbf{S}_j$ is determined by the current $\mathbf{S}_j$ and **H**, hence denoted by $\mathbf{S}^{(i)}_j = A(\mathbf{S}^{(i-1)}_j; \mathbf{H}^{(i-1)})$. Here, we omit timestamp indicators to simplify the formulation. The worker reusing objective can then be realized by choosing the version in the transformed VSM with min (max) probability to be reclaimed (preloaded), i.e.,

$$v^-(f_j) = \arg\min_{v_m}\{(A(\mathbf{S}_j; \mathbf{H}))_m \mid f_m \in \mathcal{F}^*\},$$

$$v^+(f_j) = \arg\max_{v_m}\{(A(\mathbf{S}_j; \mathbf{H}))_m \mid f_m \in \mathcal{F}\},$$

where $\mathcal{F}^* = \{f \mid f \in p, p \in H_i\}$ represents the last executed components (i.e., those components having a compatible worker in the worker pool). This process can be repeated several times to make multiple reclaiming and preloading.

The objective with the notion of VSM is then mapped to the problem of designing the transformation function A that supports the state transition of $\mathbf{S}_j$ across different timestamps.

4.3 Pipeline-aware Caching

We describe a heuristic caching policy, called *Pipeline-aware Caching* (PAC), that aims to realize the worker reusing objective. It is guided by two locality properties: *temporal locality* and *spatial locality*, each of which has a respective A function transforming the VSM. The policy enables predicting the component versions which are more (less) likely to be chosen in the next submitted pipeline. We first introduce a quantitative method to manipulate the VSM.

VSM scaling function. Given a set of versions being predicted as more likely to occur in the future, the *VSM scaling function* decreases the scores of the other versions in favor of the selected one (i.e., the *target*) while preserving the grand sum of the VSM.

When the target is a single version, the *entry-level scaling* takes as input the `schema.increment` pair $\langle m, n \rangle$, and uses a factor α to adjust the intensity. The VSM transformation then becomes

$$A(\mathbf{S}; m, n, \alpha) = (1 - \alpha)\mathbf{S} + \alpha\mathbf{e}(m, n),$$

where

$$\mathbf{e}(m, n) = (e_{ij}) = \begin{cases} 1, & i = m \land j = n \\ 0, & \text{otherwise} \end{cases}, \forall i, j.$$

When the target is a set of versions **V**, the scaling cannot be performed iteratively on each element in **V**, as the entry-level scaling function is not associative. Instead, a *batched scaling* function is used in the VSM transformation as follows:

$$A(\mathbf{S}; \mathbf{V}, \alpha) = (1 - \alpha)\mathbf{S} + \frac{\alpha}{card(\mathbf{V})} \sum_{m,n \in \mathbf{V}} \mathbf{e}(m, n).$$

Using the aforementioned scaling functions, PAC utilizes two localities extracted from the version patterns during pipeline evolution to transform the VSM across different timestamps.

Temporal locality. The *temporal locality* refers to the property that a component in a recently submitted pipeline is likely to be submitted again. In particular, the user is likely to use the same component with a different input in a new pipeline. This is similar to the temporal locality property in the database and CPU caching. To leverage it, the key idea is to favor the versions of recently executed components. More specifically, PAC assigns lower scores to the versions that are further in the pipeline history window.

Algorithm 1 details the function `VSMTransformTL` that transforms VSMs based on temporal locality. Given the last VSMs of all components $\mathbf{S}^{(i-1)}$, the pipeline history **H**, and the intensity factor α, it first makes a copy of VSMs, then goes through all components in the pipeline, and for each component, it collects the versions appeared in the previous pipelines into a set **V** (line 6). Then it iterates over **V** and filters the version in the latest pipeline to the set $\mathbf{V}_c$ (line 7-10). The versions in $\mathbf{V}_c$ are sent to the VSM scaling function A that assigns them lower scores in the VSM (line 11).

Spatial locality. The *spatial locality* refers to the property that if a component changes between two consecutive submitted pipelines, then it is likely to change again. This property captures the exploration phase for data analytics pipeline development, in which

Algorithm 1: Transforming VSMs with temporal locality.

```
Input: $S^{(i-1)} = \{S_j^{(i-1)}\}$: Last VSMs; $H^{(i-1)}$: Last pipeline history;
    $\alpha$: Intensity.
Output: $S^{(i)}$: Next VSMs.
Function VSMTransformTL($S^{(i-1)}, H, \alpha$)
    $S^{(i)} \leftarrow S^{(i-1)}$
    $p \leftarrow$the last pipeline in $H$
    for $k \leftarrow 1$ to $card(p)$ do
        $\langle f_1, m_1.n_1 \rangle \leftarrow k$'s component in $p$
        $V \leftarrow$ versions of $f_1$ in $H \backslash \{p\}$
        $V_c \leftarrow \{\}$
        for $(v = \langle f_2, m_2.n_2 \rangle) \in V$ do
            if $f_1 = f_2$ and $(m_1 \neq m_2$ or $n_1 \neq n_2)$ then
                $V_c \leftarrow V_c \cup \{v\}$
        $S_{f_1}^{(i)} \leftarrow A(S_{f_1}^{(i)}; V \backslash V_c, \alpha)$
```

Algorithm 2: Transforming VSMs with spatial locality.

```
Input: $S^{(i-1)} = \{S_j^{(i-1)}\}$: Last VSMs; $H^{(i-1)}$: Last pipeline history;
    $\alpha$: Intensity.
Output: $S^{(i)}$: Next VSMs.
Function VSMTransformSL($S^{(i-1)}, H, \alpha$)
    $S^{(i)} \leftarrow S^{(i-1)}$
    $p^{(t)}, p^{(t-1)} \leftarrow$the last, second pipeline in $H$
    for $k \leftarrow 1$ to $card(p^{(t)})$ do
        $\langle f_1, m_1.n_1 \rangle, \langle f_2, m_2.n_2 \rangle \leftarrow k$'s component in $p^{(t)}, p^{(t-1)}$
        if $f_1 = f_2$ and
        $((m_1 = m_2$ and $n_1 \neq n_2)$ or $(m_1 \neq m_2$ and $n_1 = n_2))$ then
            for $h \in$ {from first to $k$'s component in $p^{(t-1)}$} do
                $S_h^{(i)} \leftarrow A(S_h^{(i)}; V \backslash \{$versions of $h$ in $p^{(t-1)}\}, \alpha)$
            if $m_1 = m_2$ and $n_1 \neq n_2$ and
            version $\langle m_1, 2n_2 - n_1 \rangle$ exists for $f_1$ then
                $S_{f_1}^{(i)} \leftarrow A(S_{f_1}^{(i)}; m_1, 2n_2 - n_1, \alpha)$
            if $m_1 \neq m_2$ and $n_1 = n_2$ and
            version $\langle 2m_1 - m_2, n_1 \rangle$ exists for $f_1$ then
                $S_{f_1}^{(i)} \leftarrow A(S_{f_1}^{(i)}; 2m_1 - m_2, n_1, \alpha)$
```

users try different versions of the code or data until satisfied with the results. This property is similar to the spatial locality in the database and CPU caching, as the versions are closer in number. To leverage it, the policy assigns higher scores to the next versions of the component that is modified more recently.

Algorithm 2 details the function `VSMTransformSL` that transforms VSM based on temporal locality. Given the same inputs as `VSMTransformTL`, after performing the same VSM copying, the algorithm fetches two consecutive pipelines in the submission history (line 5). Then, for each component f in the two pipelines, it checks for a pattern of continuous version increment, by checking if the consecutive versions change only in either `schema` or `increment` (line 6-7). If so, the algorithm uses this pattern to update VSMs to better predict the next submission. Specifically, it predicts the next iteration of this pattern, namely $\langle m_1, 2n_2 - n_1 \rangle$ for the `schema` change and $\langle 2m_1 - m_2, n_1 \rangle$ for the `increment` change, as more likely to occur in future pipeline submissions. It then increases the corresponding values in f's VSM, if the component with that version has already been submitted to the system (line 10-15).

5 IMPLEMENTATION

In this section, we describe some implementation details of SecCask. We explain how a worker runtime tracks packages based on the imported modules, how manifests are extended to accelerate the compatibility check, and how EncFS is implemented for Python.

5.1 Active Packages

In programming languages like Python, a *package* is a software library that implements a set of functionalities. It can contain one or more *modules* that are loaded statically or dynamically at runtime. Developers organize the modules without standard rules. For instance, modules may have different names from their package, adding difficulty as the version numbers are only applicable to packages. To tackle them in compatibility check, SecCask introduces the notion of *active packages*, which have at least one module imported in the runtime. The checks are then performed by comparing active packages with package requirements in the component manifest.

Deriving the active packages of a worker comprises two parts: building a module-package mapping and retrieving packages for the active modules. The former gets all the module names related to a specific package. For Python, this is done by reading the file "top_level.txt" inside the "egg_info" metadata directory of the package. Being *trusted files* in the shielding framework, the hash of these files is stored in the enclave to protect them against tampering.

5.2 Compatibility Check

For data analytics tasks, checking the compatibility for individual packages can be expensive, as many packages are used to simplify the development. SecCask accelerates it by extending the library component manifest (see Listing 1 for an example) to perform a three-phase check. The first phase compares the semantic version of the new component to that of the last executed one, and checks if the names and versions match. Note that matching the major versions alone is not sufficient, as the output schema equivalence does not mean the code generates outputs using the same packages. If phase 1 fails, the second phase uses a hash function to summarize the package information and compares the hash with the pre-computed hash of the required packages. Here we use SHA-256 over the dumped JSON string of the sorted dictionary with all packages and their versions. If phase 2 fails, the third phase checks package compatibility by comparing the active packages against the requirements in the manifest, using either of the version compatibility types introduced in Section 3.4. The package names and their versions in the component manifest are used for this check.

5.3 Encrypted Filesystem

We modify the file object operational functions of the Python interpreter to implement EncFS. The middleware also adds stubs of global variables to store, e.g., encryption keys, which the modified functions read. The worker links `libpython.so` and sets up EncFS dynamically by initializing and configuring these global variables.

```
type: library
name: imagenet-alexnet-cnn-training
# Phase 1 - Library Version Compatibility
version: master@1.4
# Phase 2 - Package Hash
hash: c41447ee05f4ccd6...088082dec5c92b30
# Phase 3 - Package Version Compatibility
packages: # Exact Version Match
    numpy: 1.22.4
    matplotlib: 3.6.2
    ...
packages_semver: # Semantic Version Match
    tensorflow: 2.9.1
    tqdm: 4.64.1
    ...
```

Listing 1: Example of a library manifest.

We use OpenSSL 1.1.1 for AES. However, it does not provide dedicated APIs for GMAC. As a workaround, we compute it by leveraging GCM's authentication-only mode. Specifically, an AES-CTR context is first used to encrypt pages. The ciphertext is then sent as additional authenticated data into an AES-GCM context. Running encryption in this context computes GMAC for the ciphertext.

6 EVALUATION

In this section, we evaluate the performance of SecCask.

6.1 Workloads

We use four sets of workloads in the experiments. PAC and MLPerf Training workloads are for microbenchmarks, while AutoLearn and Sentiment Analysis workloads are for case studies.

PAC workload. PAC depends on a number of hyperparameters, namely the size of the worker pool P, the size of pipeline history M, and the factor α for the VSM scaling function. To evaluate their impact, we create a *PAC workload* with only version information. It consists of 800 pipelines without branching, each of which has 5 components executing in linear order. This synthetic workload captures real-world activities in which a user performs exploration in the component search space of a final pipeline.

MLPerf Training workload. We use MLPerf Training [33] to evaluate the storage overhead. This workload contains file access patterns of real-world data analytics and machine learning pipelines. We use a recent version of MLPerf Training benchmark[2] that contains eight tasks. Among them, six preprocessing tasks are used for evaluating the end-to-end performance of the storage component[3].

AutoLearn workload. AutoLearn [25] is an algorithm that automates feature engineering by mining, generating, and selecting correlated features to improve model training performance. Figure 4 shows the pipeline evolution of this workload, which uses UCI Sonar dataset as the `DS` component and includes four library components: `IG` for preprocessing based on Information Gain, `DR` for dependent regression on processed data, and `Stable` for selecting

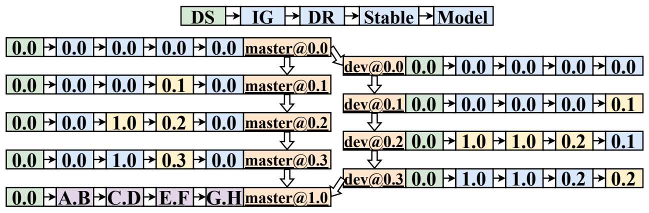

Figure 4: Pipeline evolution of AutoLearn workload.

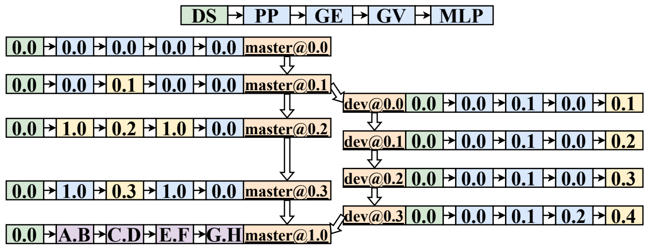

Figure 5: Pipeline evolution of Sentiment Analysis workload.

stable features fed into `Model` component with different models, such as SVM, AdaBoost, and Random Forest. The workflow captures a common pipeline evolution having two branches, `master` and `dev`, which are developed independently and later merged using the metric-based mechanism from MLCask. For simplicity, we ignore the merge operation in our experiments.

Sentiment Analysis workload. This workload is derived from LDA-Reg's experimental study, which tunes neural network parameters based on the knowledge extracted from external corpora [62]. Figure 5 shows the pipeline evolution of this workload, which uses Sentence Polarity dataset with Large Movie Review dataset as the external corpora, packed into the `DS` component. It includes four library components: `PP` for preprocessing, `GE` for generating embeddings, and `GV` for generating feature vectors fed into `MLP` component with a multilayer perceptron. The workflow represents another common pipeline evolution, in which the pipeline development is incremental, with components updated iteratively as hyperparameters are tuned. The best component combination is then merged into the master branch using metric-based merging.

6.2 Baselines

To evaluate the PAC's effectiveness, we implement different caching policies in SecCask. In particular, we implement least-recently-used (LRU), least-frequently-used (LFU), and first-in-first-out (FIFO), and compare their performance with PAC.

For EncFS microbenchmark, Intel Protected Filesystem is compared as a baseline solution. Besides in enclaves, tests also include executing outside to analyze trusted overheads for storage.

To evaluate the overhead of end-to-end trustworthy pipeline execution and PAC in production-ready pipelines, we implement four variants of SecCask. *Untrusted* is a baseline that runs the system outside SGX enclaves. *Trusted* represents the system running modules inside enclaves, with remote attestation enabled. *Trusted, Singleton*

[2]https://github.com/mlcommons/training/commit/e6f45a4dbc7e85d2
[3]The reinforcement learning task does not have a dataset and its corresponding preprocessing phase, while the single stage detector task stores processed data in a standalone MongoDB database, which violates our system model.

Table 1: Breakdown of cold start overhead excluding enclave creation and framework loading.

Setup	Worker	Check Worker Pool	Establish TLS Connection	Establish RA-TLS Connection	Generate Worker Manifest	Dispatch Execution Task
Untrusted	New	15.8ms ± 3.7ms	21.9ms ± 4.5ms	-	1669.9ms ± 568.6ms	-
	Cached	14.3ms ± 3.9ms	-	-	-	12.5ms ± 12.2ms
Trusted	New	25.1ms ± 22.8ms	-	741.3ms ± 270.4ms	2674.6ms ± 963.9ms	-
	Cached	16.1ms ± 5.1ms	-	-	-	11.8ms ± 8.3ms

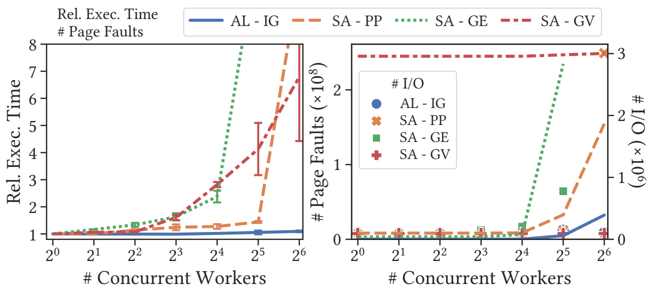

Figure 6: Relative execution time together with numbers of page faults and I/Os of concurrent workers.

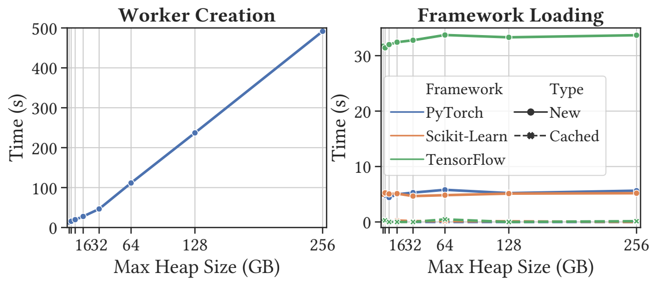

Figure 7: The two major overheads of cold starts.

is a special setup running all components in a single worker. This has a strict assumption that all packages are compatible across a workflow, as discussed in the challenges. We adapt the workloads to this requirement. *Trusted, PAC* adds compatibility checks and PAC to *Trusted* to reduce cold start overheads.

We omit the comparison against non-private pipeline management systems, e.g., ModelDB [58], MLflow [10, 65], and MLCask [32]. All our experiments on the pipeline evolution of these systems show negligible differences from the untrusted mode of SecCask.

6.3 Experimental Setup

Hardware and OS. We run our experiments on a server with 2 Intel Xeon Gold 6342 CPUs @ 3.5GHz and 512GB of memory. This server supports the second generation of SGX (SGXv2). The EPC size is set to 32GB. The OS is Ubuntu 20.04 with Linux kernel 5.15.0-43.

Shielding framework. SecCask uses Gramine [56] for shielding applications inside enclaves. We observe that giving fewer resources to Gramine will improve the performance of a single-user application, but are more likely to crash during the execution. Unless explicitly indicated, we experimented with the configurable parameters (stack size, max heap size, thread number, etc.), and set them to the lowest values that do not crash the workers.

6.4 Microbenchmark

Worker scalability. We analyze the scalability of workers by measuring the execution time with increasing numbers of concurrently executing components in one node, each occupying an enclave. We use 8 library components from the AutoLearn and Sentiment Analysis workloads, which have different execution patterns.

Figure 6 shows the relative execution time, the number of page faults, and the number of I/O with concurrently running workers over that of a single worker. We also measure the number of page faults and I/Os, to understand whether the execution is memory or I/O intensive. We omit the results for `DR`, `Stable`, `Model`, and `MLP`, as they are similar to that of `IG`. It can be seen that the impact of concurrent workers on `IG` is small. For `PP` and `GV` from the Sentiment Analysis workload, there are knee points after which the overheads increase significantly, even though they remain reasonable with fewer than 16 concurrent workers. This can be explained by the sharp increases in page faults and I/O after the knee points, indicating that the workers are contending for memory and I/O resources. `GV` has a more stable increase for average execution time, but the standard deviation becomes larger with more concurrent workers. This is because this component writes a large amount of data, which causes the hard disk to be the bottleneck.

Cold start overheads. To understand how SecCask reduces the impact of cold starts, we measure the overhead for initializing a pipeline component, both in trusted and untrusted settings (with and without enclaves, respectively). We break down the duration from when a component execution is requested, to when the component begins its execution, into six stages.

Table 1 shows the latency of different stages, except for the most time-consuming stage of worker creation and framework loading, which we discuss later. Generating a worker manifest incurs a large overhead, as it requires a number of reading from the storage. However, this is a one-time cost, incurred only when new packages are imported during component execution. The total cost of performing remote attestation and establishing the secure channel is less than one second. The other stages have similar costs in the untrusted settings as in the trusted ones.

We examine the overhead of creating a worker and loading the framework in more detail. Figure 7 shows this cost with increasing heap sizes. The left figure shows the worker creation time, which includes creating an enclave, loading the shielding framework, and loading the runtime. The cost is linear with the heap size until the size exceeds the EPC memory limit (32GB in our experiments).

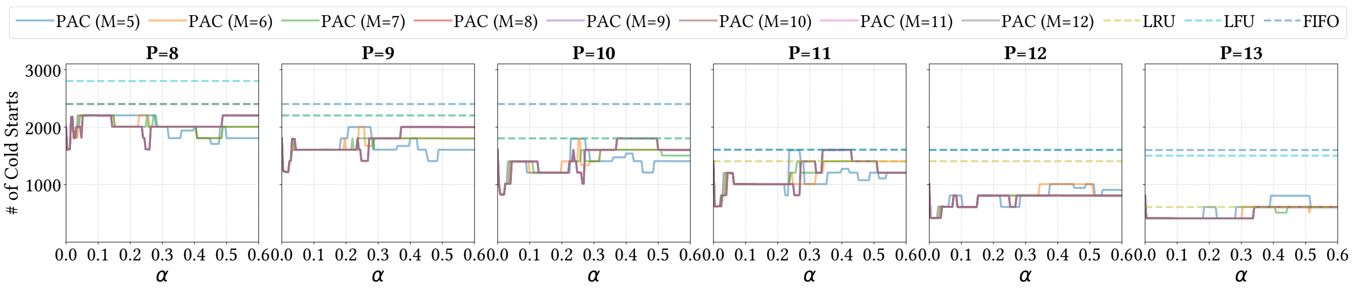

Figure 8: Number of cold starts on PAC workload with PAC-TS, LRU, LFU, and FIFO.

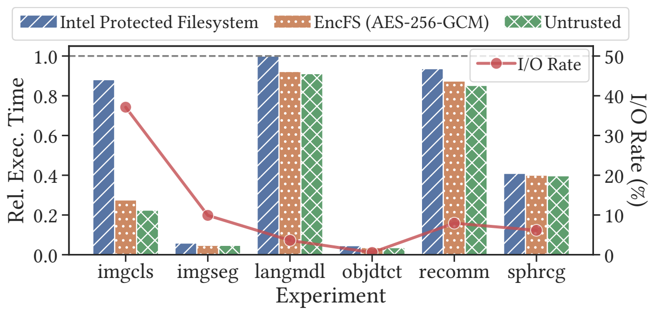

Figure 9: Relative execution time and I/O rates of components in MLPerf Training workload.

After that, the slope becomes steeper. For framework loading, we measure the cost for PyTorch, Scikit-Learn, and TensorFlow. This cost is lower than that of worker creation and is independent of the heap size. In summary, the cold start latency can be in the order of minutes, which is a significant overhead for small pipelines.

EncFS. We execute and record execution time for the preprocessing tasks in MLPerf Training workload, namely image classification (`imgcls`), image segmentation (`imgseg`), language modeling (`langmdl`), object detection (`objdtct`), recommendation (`recomm`) and speech recognition (`sphrcg`).

Figure 9 shows the latencies relative to the longest task. The figure depicts the I/O rates, demonstrating that some tasks are more I/O intensive than others. The result shows that EncFS introduces small overheads across different I/O access patterns. It significantly outperforms Intel PFS for I/O intensive tasks, since EncFS is carefully designed to align with the OS file access patterns.

Impact of PAC. To show the effectiveness of PAC, we perform a comprehensive grid search over the parameter space of the three hyperparameters. The values of P and M are chosen from 8 to 13 and from 4 to 13, respectively, while α is in the range $[0, 0.6]$.

Table 2 summarizes the benefits of PAC over the other caching policies. It shows the impact of individual localities (TO for temporal locality only, SO for spatial locality only) and of PAC that combines both localities (TS). It can be seen that combining both spatial and temporal locality helps achieve the best reduction over all the parameter values. In particular, PAC helps decrease the cold start time, and it is more effective than other policies, by up to 74.28%. Figure 8 shows the number of cold starts when M is set between 5 and 12. The result confirms that tuning α and M is not necessary, as the PAC curves are below the baselines in most cases.

Table 2: Cold start reduction of PAC (in percentage).

	LRU			LFU			FIFO		
	TO	SO	TS	TO	SO	TS	TO	SO	TS
Min	19.29	6.66	6.86	19.29	6.66	6.86	19.29	6.66	6.86
Max	42.36	70.29	70.22	71.08	73.96	73.89	72.85	74.28	74.22
Avg	30.42	33.97	**38.12**	38.14	40.98	**44.92**	37.60	39.84	**43.78**

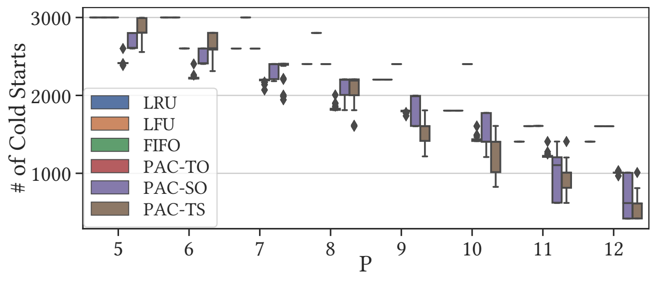

Figure 10: Distribution of cold starts with each and both localities of PAC compared with LRU, LFU, and FIFO.

To further study the effect of individual localities, we average the results over α and M and compute the distribution of cold starts with P from 5 to 12. The results, shown in Figure 10, demonstrate two findings. First, the temporal locality has the potential to achieve better performance at the expense of stability, as the distribution contains more outliers. The spatial locality, conversely, is more stable but is not always better than the baselines. Second, combining both localities results in the lowest number of cold starts.

6.5 Case Study

End-to-end lifecycle. We evaluate the system's end-to-end performance using two realistic pipelines [25, 62]. We measure the total execution time and their cache miss rates, shown in Figure 11.

The differences in end-to-end pipeline execution time are stable as the workflows evolve. PAC achieves a 57.3% overhead reduction for AutoLearn workflow, and a 79.4% of reduction for Sentiment Analysis. We measure the latency of the naive design (discussed in Section 3.2) and denote it as *Trusted, Singleton* in the figure. For AutoLearn, SecCask consistently outperforms the naive design,

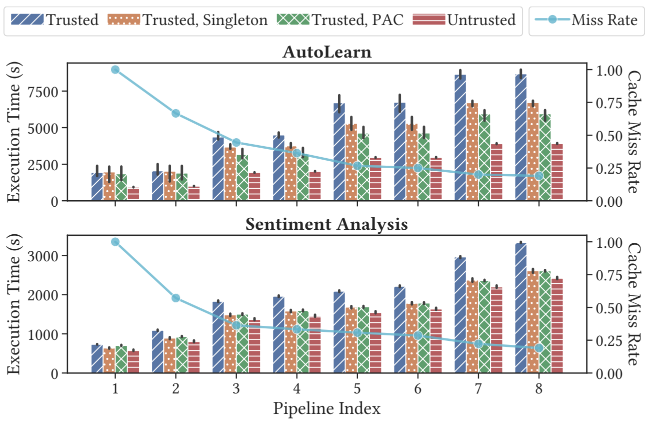

Figure 11: Cumulative pipeline time and PAC's cache miss rate for AutoLearn and Sentiment Analysis workflows.

Table 3: EPC costs for modules and functionalities (in MB).

Base		Metadata		Functionality		
LibOS	Python RT	Worker	Others	RA Gen	RA Ver	EncFS
11.105	36.858	+7.130	+5.979	+0.873	+18.269	+0.045

because of the lower memory requirement per enclave and enclave component sharing. For Sentiment Analysis, the discrepancy between the two setups is smaller. The main reason is that the execution time is more evenly distributed across components. In other words, more components have a large execution time in the workload, decreasing the impact of cold starts. However, we note that *Trusted, Singleton* is only available when all packages used across the whole lifecycle are fixed and do not conflict, which is unlikely in practice, because dependencies evolve and get updated constantly over time [14]. In summary, PAC achieves a small overhead over untrusted settings, and the use of multiple enclaves per pipeline can be more efficient than the single-enclave design.

EPC overhead. The number of EPC pages consumed by each enclave in SecCask includes the pages used for the data, the model, etc., and the pages introduced by SecCask to enable trusted pipeline execution. We consider the latter as the system's EPC overhead. Measuring this overhead is challenging because the runtime environment inside the enclave is highly coupled. We break down this into the base, consisting of LibOS and the Python runtime (RT), the metadata needed for core functionalities such as EncFS and the coordinator, and the functionality themselves. We obtain this breakdown by subtracting the enclave max EPC usage under the minimal runtime environment with functionalities on and off.

Table 3 lists the EPC costs of SecCask. The metadata costs are different between the workers and other types, i.e., coordinator, workspace, and KDS. The worker consumes more resources because the retrieval of package metadata requires storing more hashes for verification (Section 5.1). PAC does not introduce significant EPC overhead compared to non-worker enclaves. The cost of EncFS is small, as it operates at the page level. For RA, quote verification (Ver) is more expensive than quote generation (Gen), because the verification logic happens within the enclave, as opposed to simply invoking operating to generate quotes by the hardware (Gen).

7 RELATED WORK

SecCask shares a similar goal of securing data analytics applications with other works using TEEs. It addresses a similar problem, i.e. cold start, to other works in the context of serverless computing. We have discussed the cold start problem in depth in Section 4. In this section, we discussed the related work on TEE-based systems.

Mitigating EPC limitation. Prior work on TEE-based secure ML has focused on addressing the high overheads caused by limited EPC, which can significantly impact application performance up to 1000x [39, 53]. To mitigate this issue, one approach is *on-demand weights loading*, where model weights are loaded on-demand, typically layer-by-layer [26, 29, 30]. *Delegation* is another approach that moves some computation outside of enclaves while preserving privacy, mostly matrix multiplication. Some of the works only target model inference [53, 55], while others cover training as well [5, 18]. Other approaches include adaptive convolution layers partitioning [29] and using the compiler to reduce the tensor framework [24]. These works are orthogonal to ours. Moreover, although SGXv2 increases the EPC size, the cold start problem addressed by SecCask remains challenging for applications that require low latency. The techniques in the works discussed above can be implemented in SecCask to further reduce cold start overheads, as it has been shown that the enclave creation latency is proportional to its heap usage.

General TEE-based ML. Chiron [22] and Citadel [66] support both data and model privacy. Each training enclave in Chiron creates a sandbox [23] to confine malicious activities. However, these systems focus on specific components instead of entire, dynamic pipelines. TensorSCONE [28], secureTF [45] and Perun [41] are general in-enclave ML frameworks. Our work supports them as components in the pipelines. Furthermore, it is flexible and can support other non-ML components, for example, ones that perform data cleaning.

8 CONCLUSION

In this paper, we presented SecCask, a secure and efficient data analytics pipeline management system. It leverages trusted execution environments backed by hardware to achieve both security and efficiency on end-to-end data analytics pipelines. It addresses the cold start problem in the above context by runtime reusing and caching. The novel caching policy, called PAC, exploits the pipeline history and uses its access patterns on component versions to guide the manipulation of the worker pool, resulting in a significant increase in hit rates. We implemented the system and evaluated its performance through case studies on real workflows. The results show that SecCask ensures security with practical overheads.

ACKNOWLEDGMENTS

This research is supported by the National Research Foundation, Singapore under its Emerging Areas Research Projects (EARP) Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

REFERENCES

[1] Martín Abadi, Andy Chu, Ian J. Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. 2016. Deep Learning with Differential Privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Vienna, Austria, 308–318.

[2] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018*. USENIX Association, Boston, MA, USA, 923–935.

[3] Mohamed Alzayat, Jonathan Mace, Peter Druschel, and Deepak Garg. 2022. Groundhog: Efficient Request Isolation in FaaS. arXiv:2205.11458

[4] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, André Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Peter R. Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, Savannah, GA, USA, 689–703.

[5] Aref Asvadishirehjini, Murat Kantarcioglu, and Bradley A. Malin. 2020. GOAT: GPU Outsourcing of Deep Learning Training With Asynchronous Probabilistic Integrity Verification Inside Trusted Execution Environment. arXiv:2010.08855

[6] Tian Bai, Shanshan Zhang, Brian L. Egleston, and Slobodan Vucetic. 2018. Interpretable Representation Learning for Healthcare via Capturing Disease Progression through Time. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018*. ACM, London, UK, 43–51.

[7] Sebastian P. Bayerl, Tommaso Frassetto, Patrick Jauernig, Korbinian Riedhammer, Ahmad-Reza Sadeghi, Thomas Schneider, Emmanuel Stapf, and Christian Weinert. 2020. Offline Model Guard: Secure and Private ML on Mobile Devices. In *2020 Design, Automation & Test in Europe Conference & Exhibition, DATE 2020*. IEEE, Grenoble, France, 460–465.

[8] Sumon Biswas, Mohammad Wardat, and Hridesh Rajan. 2022. The Art and Practice of Data Science Pipelines: A Comprehensive Study of Data Science Pipelines In Theory, In-The-Small, and In-The-Large. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022*. ACM, Pittsburgh, PA, USA, 2091–2103.

[9] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium, USENIX Security 2018*. USENIX Association, Baltimore, MD, USA, 991–1008.

[10] Andrew Chen, Andy Chow, Aaron Davidson, Arjun DCunha, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Clemens Mewald, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Avesh Singh, Fen Xie, Matei Zaharia, Richard Zang, Juntai Zheng, and Corey Zumar. 2020. Developments in MLflow: A System to Accelerate the Machine Learning Lifecycle. In *Proceedings of the Fourth Workshop on Data Management for End-To-End Machine Learning, In conjunction with the 2020 ACM SIGMOD/PODS Conference, DEEM@SIGMOD 2020*. ACM, Portland, OR, USA, 5:1–5:4.

[11] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained.

[12] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. 2020. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st International Middleware Conference*. ACM, Delft, The Netherlands, 356–370.

[13] Alexandre Decan and Tom Mens. 2021. What Do Package Dependencies Tell Us About Semantic Versioning? *IEEE Trans. Software Eng.* 47, 6 (2021), 1226–1240.

[14] Jens Dietrich, David J. Pearce, Jacob Stringer, Amjed Tahir, and Kelly Blincoe. 2019. Dependency versioning in the wild. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019*, Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc (Eds.). IEEE / ACM, Montreal, Canada, 349–359.

[15] Kha Dinh Duy, Taehyun Noh, Siwon Huh, and Hojoon Lee. 2021. Confidential Machine Learning Computation in Untrusted Environments: A Systems Security Perspective. *IEEE Access* 9 (2021), 168656–168677.

[16] Muhammad El-Hindi, Tobias Ziegler, Matthias Heinrich, Adrian Lutsch, Zheguang Zhao, and Carsten Binnig. 2022. Benchmarking the Second Generation of Intel SGX Hardware. In *International Conference on Management of Data, DaMoN 2022*. ACM, Philadelphia, PA, USA, 5:1–5:8.

[17] Bishakh Chandra Ghosh, Sourav Kanti Addya, Nishant Baranwal Somy, Shubha Brata Nath, Sandip Chakraborty, and Soumya K. Ghosh. 2020. Caching Techniques to Improve Latency in Serverless Architectures. In *2020 International Conference on COMmunication Systems & NETworkS, COMSNETS 2020*. IEEE, Bengaluru, India, 666–669.

[18] Hanieh Hashemi, Yongqin Wang, and Murali Annavaram. 2021. DarKnight: An Accelerated Framework for Privacy and Integrity Preserving Deep Learning Using Trusted Hardware. In *MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, Virtual Event, Greece, 212–224.

[19] Zhiting Hu, Xuezhe Ma, Zhengzhong Liu, Eduard H. Hovy, and Eric P. Xing. 2016. Harnessing Deep Neural Networks with Logic Rules. arXiv:1603.06318

[20] Waldemar Hummer, Vinod Muthusamy, Thomas Rausch, Parijat Dube, Kaoutar El Maghraoui, Anupama Murthi, and Punleuk Oum. 2019. ModelOps: Cloud-Based Lifecycle Management for Reliable and Trusted AI. In *IEEE International Conference on Cloud Engineering, IC2E 2019*. IEEE, Prague, Czech Republic, 113–120.

[21] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J. Rossbach, and Emmett Witchel. 2020. Telekine: Secure Computing with Cloud GPUs. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020*, Ranjita Bhagwan and George Porter (Eds.). USENIX Association, Santa Clara, CA, USA, 817–833.

[22] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. 2018. Chiron: Privacy-preserving Machine Learning as a Service. arXiv:1803.05961

[23] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. 2018. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. *ACM Trans. Comput. Syst.* 35, 4 (2018), 13:1–13:32.

[24] Nick Hynes, Raymond Cheng, and Dawn Song. 2018. Efficient Deep Learning on Multi-Source Private Data. arXiv:1807.06689

[25] Ambika Kaul, Saket Maheshwary, and Vikram Pudi. 2017. AutoLearn - Automated Feature Generation and Selection. In *2017 IEEE International Conference on Data Mining, ICDM 2017*, Vijay Raghavan, Srinivas Aluru, George Karypis, Lucio Miele, and Xindong Wu (Eds.). IEEE Computer Society, New Orleans, LA, USA, 217–226.

[26] Kyungtae Kim, Chung Hwan Kim, Junghwan John Rhee, Xiao Yu, Haifeng Chen, Dave (Jing) Tian, and Byoungyoung Lee. 2020. Vessels: efficient and scalable deep learning prediction on trusted processors. In *SoCC '20: ACM Symposium on Cloud Computing*. ACM, Virtual Event, USA, 462–476.

[27] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. 2018. Integrating Remote Attestation with Transport Layer Security. arXiv:1801.05863

[28] Roland Kunkel, Do Le Quoc, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. 2019. TensorSCONE: A Secure TensorFlow Framework using Intel SGX. arXiv:1902.04413

[29] Taegyeong Lee, Zhiqi Lin, Saumay Pushp, Caihua Li, Yunxin Liu, Youngki Lee, Fengyuan Xu, Chenren Xu, Lintao Zhang, and Junehwa Song. 2019. Occlumency: Privacy-preserving Remote Deep-learning Inference Using SGX. In *The 25th Annual International Conference on Mobile Computing and Networking, MobiCom 2019*, Stephen A. Brewster, Geraldine Fitzpatrick, Anna L. Cox, and Vassilis Kostakos (Eds.). ACM, Los Cabos, Mexico, 46:1–46:17.

[30] Yuepeng Li, Deze Zeng, Lin Gu, Quan Chen, Song Guo, Albert Y. Zomaya, and Minyi Guo. 2021. Lasagna: Accelerating Secure Deep Learning Inference in SGX-enabled Edge Cloud. In *SoCC '21: ACM Symposium on Cloud Computing*, Carlo Curino, Georgia Koutrika, and Ravi Netravali (Eds.). ACM, Seattle, WA, USA, 533–545.

[31] Ping-Min Lin and Alex Glikson. 2019. Mitigating Cold Starts in Serverless Platforms: A Pool-Based Approach. arXiv:1903.12221

[32] Zhaojing Luo, Sai Ho Yeung, Meihui Zhang, Kaiping Zheng, Lei Zhu, Gang Chen, Feiyi Fan, Qian Lin, Kee Yuan Ngiam, and Beng Chin Ooi. 2021. MLCask: Efficient Management of Component Evolution in Collaborative Data Analytics Pipelines. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, Chania, Greece, 1655–1666.

[33] Peter Mattson, Christine Cheng, Gregory F. Diamos, Cody Coleman, Paulius Micikevicius, David A. Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debo Dutta, Udit Gupta, Kim M. Hazelwood, Andy Hock, Xinyuan Huang, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Carole-Jean Wu, Lingjie Xu, Cliff Young, and Matei Zaharia. 2020. MLPerf Training Benchmark. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020*, Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze (Eds.). mlsys.org, Austin, TX, USA, 336–349.

[34] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos V. Rozas. 2016. Intel® Software Guard Extensions (Intel® SGX) Support for Dynamic Memory Management Inside an Enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016, HASP@ICSA 2016*. ACM, Seoul, Republic of Korea, 10:1–10:9.

[35] Hui Miao, Ang Li, Larry S. Davis, and Amol Deshpande. 2017. ModelHub: Deep Learning Lifecycle Management. In *33rd IEEE International Conference on Data Engineering, ICDE 2017*. IEEE Computer Society, San Diego, CA, USA, 1393–1394.

[36] Fatemehsadat Mireshghallah, Mohammadkazem Taram, Praneeth Vepakomma, Abhishek Singh, Ramesh Raskar, and Hadi Esmaeilzadeh. 2020. Privacy in Deep Learning: A Survey. arXiv:2004.12254

[37] Fan Mo, Zahra Tarkhani, and Hamed Haddadi. 2022. SoK: Machine Learning with Confidential Computing. *CoRR* abs/2208.10134 (2022), 18 pages. arXiv:2208.10134

[38] Karthik Nandakumar, Nalini K. Ratha, Sharath Pankanti, and Shai Halevi. 2019. Towards Deep Neural Network Training on Encrypted Data. In *IEEE Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops 2019*. Computer Vision Foundation / IEEE, Long Beach, CA, USA, 40–48.

[39] Tu Dinh Ngoc, Bao Bui, Stella Bitchebe, Alain Tchana, Valerio Schiavoni, Pascal Felber, and Daniel Hagimont. 2019. Everything You Should Know about Intel SGX Performance on Virtualized Systems. In *Abstracts of the 2019 SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems*. ACM, Phoenix, AZ, USA, 77–78.

[40] Lina Ochoa, Thomas Degueule, Jean-Rémy Falleri, and Jurgen J. Vinju. 2022. Breaking bad? Semantic versioning and impact of breaking changes in Maven Central. *Empir. Softw. Eng.* 27, 3 (2022), 61.

[41] Wojciech Ozga, Do Le Quoc, and Christof Fetzer. 2021. Perun: Confidential Multi-stakeholder Machine Learning Framework with Hardware Acceleration Support. In *Data and Applications Security and Privacy XXXV - 35th Annual IFIP WG 11.3 Conference, DBSec 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12840)*, Ken Barker and Kambiz Ghazinour (Eds.). Springer, Calgary, Canada, 189–208.

[42] Manas A. Pathak, Bhiksha Raj, Shantanu Rane, and Paris Smaragdis. 2013. Privacy-Preserving Speech Processing: Cryptographic and String-Matching Frameworks Show Promise. *IEEE Signal Process. Mag.* 30, 2 (2013), 62–74.

[43] Tom Preston-Werner. 2013. *Semantic Versioning 2.0.0.* Retrieved 2023-06-05 from https://semver.org/spec/v2.0.0.html

[44] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A. Sartakov, and Peter R. Pietzuch. 2019. SGX-LKL: Securing the Host OS Interface for Trusted Execution. arXiv:1908.11143

[45] Do Le Quoc, Franz Gregor, Sergei Arnautov, Roland Kunkel, Pramod Bhatotia, and Christof Fetzer. 2020. secureTF: A Secure TensorFlow Framework. In *Middleware '20: 21st International Middleware Conference*. ACM, Delft, The Netherlands, 44–59.

[46] César Sabater, Aurélien Bellet, and Jan Ramon. 2020. Distributed Differentially Private Averaging with Improved Utility and Robustness to Malicious Parties. arXiv:2006.07218

[47] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020*. USENIX Association, Virtual Event, 205–218.

[48] Shreya Shankar, Rolando Garcia, Joseph M. Hellerstein, and Aditya G. Parameswaran. 2022. Operationalizing Machine Learning: An Interview Study. *CoRR* abs/2209.09125 (2022), 20 pages. arXiv:2209.09125

[49] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems*. ACM, Lausanne, Switzerland, 955–970.

[50] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. Panoply: Low-TCB Linux Applications With SGX Enclaves. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017*. The Internet Society, San Diego, CA, USA, 15 pages.

[51] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership Inference Attacks Against Machine Learning Models. In *2017 IEEE Symposium on Security and Privacy, SP 2017*. IEEE Computer Society, San Jose, CA, USA, 3–18.

[52] Khondokar Solaiman and Muhammad Abdullah Adnan. 2020. WLEC: A Not So Cold Architecture to Mitigate Cold Start Problem in Serverless Computing. In *2020 IEEE International Conference on Cloud Engineering, IC2E 2020*. IEEE, Sydney, Australia, 144–153.

[53] Zhichuang Sun, Ruimin Sun, Long Lu, and Somesh Jha. 2020. ShadowNet: A Secure and Efficient System for On-device Model Inference. arXiv:2011.05905

[54] Kun Suo, Yong Shi, Xiaohua Xu, Dazhao Cheng, and Wei Chen. 2020. Tackling Cold Start in Serverless Computing with Container Runtime Reusing. In *Proceedings of the 2020 Workshop on Network Application Integration/CoDesign, NAI@SIGCOMM 2020*. ACM, Virtual Event, USA, 54–55.

[55] Florian Tramèr and Dan Boneh. 2019. Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware. In *7th International Conference on Learning Representations, ICLR 2019*. OpenReview.net, New Orleans, LA, USA, 19 pages.

[56] Chia-che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017*. USENIX Association, Santa Clara, CA, USA, 645–658.

[57] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2021. *SGAxe: How SGX Fails in Practice.* Retrieved 2023-06-05 from https://cacheoutattack.com/files/SGAxe.pdf

[58] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. 2016. ModelDB: a system for machine learning model management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD 2016*. ACM, San Francisco, CA, USA, 14.

[59] Wei Wang, Meihui Zhang, Gang Chen, H. V. Jagadish, Beng Chin Ooi, and Kian-Lee Tan. 2016. Database Meets Deep Learning: Challenges and Opportunities. *SIGMOD Rec.* 45, 2 (2016), 17–22.

[60] Doris Xin, Hui Miao, Aditya G. Parameswaran, and Neoklis Polyzotis. 2021. Production Machine Learning Pipelines: Empirical Analysis and Optimization Opportunities. In *SIGMOD '21: International Conference on Management of Data*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, Virtual Event, China, 2639–2652.

[61] Cong Yang, Wenfeng Wang, Yunhui Zhang, Zhikai Zhang, Lina Shen, Yipeng Li, and John See. 2021. MLife: A Lite Framework for Machine Learning Lifecycle Initialization. In *8th IEEE International Conference on Data Science and Advanced Analytics, DSAA 2021*. IEEE, Porto, Portugal, 1–2.

[62] Kai Yang, Zhaojing Luo, Jinyang Gao, Junfeng Zhao, Beng Chin Ooi, and Bing Xie. 2022. LDA-Reg: Knowledge Driven Regularization Using External Corpora. *IEEE Trans. Knowl. Data Eng.* 34, 12 (2022), 5840–5853.

[63] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets (Extended Abstract). In *27th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, Toronto, Canada, 162–167.

[64] Peterson Yuhala, Pascal Felber, Valerio Schiavoni, and Alain Tchana. 2021. Plinius: Secure and Persistent Machine Learning Model Training. In *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2021*. IEEE, Taipei, Taiwan, 52–62.

[65] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Fen Xie, and Corey Zumar. 2018. Accelerating the Machine Learning Lifecycle with MLflow. *IEEE Data Eng. Bull.* 41, 4 (2018), 39–45.

[66] Chengliang Zhang, Junzhe Xia, Baichen Yang, Huancheng Puyang, Wei Wang, Ruichuan Chen, Istemi Ekin Akkus, Paarijaat Aditya, and Feng Yan. 2021. Citadel: Protecting Data Privacy and Model Confidentiality for Collaborative Learning. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC*. ACM, Seattle, WA, USA, 546–561.

[67] Lyuye Zhang, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Bihuan Chen, and Yang Liu. 2022. Has My Release Disobeyed Semantic Versioning? Static Detection Based on Semantic Differencing. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022*. ACM, Rochester, MI, USA, 51:1–51:12.

[68] Chuan Zhao, Shengnan Zhao, Minghao Zhao, Zhenxiang Chen, Chong-Zhi Gao, Hongwei Li, and Yu-an Tan. 2019. Secure Multi-Party Computation: Theory, practice and applications. *Inf. Sci.* 476 (2019), 357–372.