Chair of Network Architectures and Services
School of Computation, Information, and Technology
Technical University of Munich

TECHNICAL UNIVERSITY OF MUNICH

SCHOOL OF COMPUTATION, INFORMATION, AND TECHNOLOGY

INFORMATICS

BACHELOR'S THESIS IN INFORMATICS

Assessment of TEEs in the AI Model Lifecycle

Mila Brajkovska

Technical University of Munich

School of Computation, Information, and Technology

Informatics

Bachelor's Thesis in Informatics

Assessment of TEEs in the AI Model Lifecycle

Bewertung von TEEs im Lebenszyklus von AI-Modellen

Author:	Mila Brajkovska
Supervisor:	Prof. Dr.-Ing. Georg Carle
Advisor:	Filip Rezabek, M. Sc.
	Kilian Glas, M. Sc.
Date:	April 14, 2025

I confirm that this Bachelor's Thesis is my own work and I have documented all sources and material used.

Garching, April 14, 2025

Location, Date

Signature

Abstract

Large Language Models (LLMs) have become essential across various fields, yet their reliance on sensitive data raises privacy and security concerns. Previous research has explored hardware-based Trusted Execution Environments (TEEs) to isolate computations and protect against untrusted infrastructures, but such approaches often suffer significant performance overhead and require extensive infrastructure modifications. However, the extent to which TEEs can efficiently handle LLMs, like those with hundreds of millions of parameters, remains underexplored—creating a gap in our understanding of the trade-offs between security and speed. In this thesis, we address this gap by investigating the feasibility of running a 110million-parameter BERT model only on AMD SEV-SNP, a VM-based TEE technology designed to encrypt memory and shield computations from adversaries. Through systematic experiments varying batch sizes and epochs, we find that memory encryption imposes only about 10–16% degradation on CPU-based training and inference—significantly lower than overheads reported in process-level enclaves—while acknowledging unresolved challenges like data poisoning and adversarial inputs. These findings highlight TEEs' practicality for confidential LLM training on CPUs, offering critical security guarantees at manageable cost, and they lay the groundwork for future exploration of even larger, GPU-accelerated models.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recent advances in artificial intelligence (AI) have led to its adoption in various industrial and academic settings, yet privacy and security concerns continue to limit its broader deployment. Multiple studies suggest that organizations employing AI-driven technologies can achieve significant gains in productivity, often cited between 15 to 40 percent [1], [2]. However, the risks of data leakage and unauthorized model extraction have forced many companies to develop and train AI models (or tools) in-house using secure protocols such as federated learning, differential privacy, homomorphic encryption, Trusted Execution Environments (TEEs) and many more. These protocols (frameworks) help ensure end-to-end confidentiality across the model lifecycles stages—including data collection, training, validation, deployment, and inference—but can introduce computational overhead, a lot of complexity and even demand change in infrastructure [3]–[6].

Ensuring privacy throughout the entire AI model lifecycle remains a significant challenge. During training, controlling and encrypting model access to data is essential to prevent leakage and theft of sensitive information, such as model weights that could be stolen or tampered with [7]. Each phase requires strict security strategies, which add complexity and time to maintain robust privacy standards. Integrating TEEs into AI workflows enhances security by isolating sensitive data and computations in hardware-based enclaves. During data ingestion, secure transport protocols like TLS encrypt data in transit [8], while TEEs process this data securely to prevent unauthorized access. In high-security ML settings, TEEs (e.g., Intel TDX or AMD SEV-SNP) isolate sensitive computations during preprocessing and training, ensuring that only authorized code runs via attestation protocols. During validation and testing, TEEs protect proprietary

test data and intermediate results, and in deployment, they shield the model against reverse engineering or unauthorized extraction of weights and architecture.

Sensitive data must be protected at all stages to prevent unauthorized access or leakage [6], [7]. Remote attestation is employed to verify that the deployed software (including the model) matches the expected, untampered configuration, thereby providing verifiable evidence that the secure setup is in place [9]. Finally, during inference, TEEs maintain the confidentiality of sensitive input data and output results by processing them within a hardware-isolated enclave—a critical feature in sectors such as healthcare and finance where data privacy is the most important [6], [7].

Hardware companies are beginning to integrate such secure capabilities directly into their products. For example, NVIDIA's H100 Tensor Core [10] offers hardware-based isolation for secure data and code processing, Intel incorporates TEEs through technologies like Software Guard Extensions (SGX) [11] and Trust Domain Extensions (TDX) [12], and AMD employs Secure Encrypted Virtualization (SEV) [13] to provide hardware-based memory encryption for virtual machines.

1.1 Research Questions

This thesis seeks to investigate how TEEs can be effectively utilized to provide security for Large Language Models in centralized environments. By examining both the theoretical and practical aspects of TEE deployment, we aim to identify their benefits, limitations, and performance implications, with a focus on maintaining efficiency while ensuring robust data protection.

- RQ1: What specific components of the AI model lifecycle (e.g. training, fine-tuning, inference) are most vulnerable to security breaches, and how effectively can TEEs protect each phase?
- RQ2: How do the security guarantees provided by TEEs impact the complexity of implementing and maintaining them within AI model deployment systems?
- RQ3: What are the types of attacks that TEEs remain vulnerable to, even with current security mechanisms?
- RQ4: What is an effective methodology for evaluating the security and performance of each component in the AI model lifecycle?
- RQ5: What trade-offs exist between security and performance in TEE-enabled LLM deployments, and how can these be optimized?

1.2 Thesis Structure

Chapter 2 provides the necessary background to understand the topics of this thesis, covering Machine Learning, LLMs, TEEs, and related architectural concepts. Chapter 3 defines the AI model lifecycle and details the types of threats it is exposed to, identifying which ones can be mitigated by a TEE while providing a comprehensive overview of the security landscape. In this chapter, we conclude with an in-depth discussion on which phases are the most vulnerable to threats and how TEEs either help or fall short in defending against these attacks. Chapter 4 describes the experiment design. Chapter 5 covers the implementation of the experiment design. Chapter 6 analyzes and compares the experimental outcomes of the implementation and setup with existing literature, focusing on the overhead and added complexities that TEEs introduce into the AI model lifecycle. Finally, Chapter 7 points to the answers of the research questions and potential directions for future work.

Chapter 2

Background

This chapter offers an overview of the key technologies and concepts behind secure computing systems and large language models(LLM). It covers the evolution of TEEs in CPU and GPU architectures, focusing on their role in protecting sensitive computations and data. It also introduces fundamental machine learning paradigms as well as various Transformer architectures used in language modelling. Together, these topics provide essential context for the analysis and discussions in this thesis, and for understanding the integration of secure computing and AI models.

2.1 Trusted Execution Environments

While traditionally implemented in CPUs, TEEs have also been integrated into GPUs (e.g., NVIDIA's H100 and H200) to expand the role of secure (confidential) computing. TEEs can operate at either the process level (e.g., Intel SGX, ARM TrustZone) or the VM level (e.g., Intel TDX, AMD SEV-SNP). In the process-based approach, only specific application processes or threads are isolated within protected enclaves, whereas VM-based TEEs extend the trust boundary to the entire virtual machine (including the OS and all running applications). VM-based TEEs are increasingly relevant for large-memory, containerized, or virtualized deployments—especially when combined with GPU TEEs for high-performance tasks [14]–[25].

A TEE enforces security guarantees by leveraging hardware-based architectural features such as specialized instructions, memory encryption, and fine-grained access control [14]. With GPU integration, TEEs now enable secure execution of high-performance workloads, ensuring data confidentiality even in untrusted environments. For instance, GPU-based TEEs allow organizations to securely perform computations like AI inference or

model training on cloud-hosted GPUs without exposing raw data to the cloud provider. NVIDIA's H100 Confidential Computing feature, for example, encrypts AI workloads during processing so that even the cloud provider cannot access the data. When an application requests a secure enclave (Intel's terminology) or a secure world (ARM's terminology), the hardware sets up a protected memory region inaccessible to other processes and privileged software, including the operating system and hypervisor [16]. Malicious software running outside the TEE cannot read or modify the data inside it, even with elevated privileges [14].

The main goals of TEEs include:

1. **Confidentiality** – Ensuring data processed within the TEE remains hidden from external entities [18].
2. **Integrity** – Guaranteeing that the code and data inside the TEE have not been tampered with [15].
3. **Attestation** – Allowing a remote party to verify the genuineness of the TEE and the code running within it [19].

Attestation involves generating a measurement (hash) of the loaded code/data, signed with a hardware-provisioned key [20]. The verifier checks this signature against a trusted authority (e.g., the hardware vendor's attestation service) to confirm that the TEE is genuine and unaltered [19]. In cloud or distributed environments, remote attestation is critical for ensuring that sensitive workloads are executed in a legitimate TEE.

2.1.1 Use Cases

Healthcare: TEEs help protect patient data during analytics or when records are shared among facilities, aiding compliance with regulations such as HIPAA. They also prevent privileged insiders or compromised hypervisors from inspecting sensitive data [26], [27].

Intellectual Property (IP) Protection: Proprietary algorithms, advanced analytics scripts, or machine learning models can be kept in a secure enclave, mitigating risks of reverse engineering or unauthorized copying [28].

Digital Rights Management (DRM): TEEs can store decryption keys and usage policies in secure hardware, curbing the risk of unauthorized content reproduction and piracy [29].

Financial Services: Banks and financial institutions handling high-value transactions use TEEs to isolate account data and transactional logic, strengthening fraud prevention [30].

Privacy-Sensitive Applications: TEEs protect personal information (e.g., biometric credentials) by limiting access to specially designated secure zones in the processor [31].

AI Workloads: With the rise of confidential computing in cloud environments, TEEs enable confidentiality-preserving training and inference of machine learning models. This allows organizations to leverage high-performance cloud resources (e.g., GPU-based TEEs) without exposing their raw data to potentially untrusted infrastructure providers [28]–[53].

2.1.2 Process-Based TEEs

Intel SGX

Intel Software Guard Extensions (SGX) is a CPU-based TEE that enables enclaves, protected memory regions isolated from the operating system [14], [54]. Early implementations featured a relatively small Enclave Page Cache (EPC) of about 128 MB of physically protected memory, with 90–94 MB typically usable by applications [14]. Large AI models often exceed these constraints, and while newer architectures support paging mechanisms, frequent memory swaps increase latency under heavy workloads [14], [55]. Because SGX primarily targets application-level isolation, its throughput is bounded by CPU memory bandwidth and enclave overhead can lower effective performance for AI inference or training tasks [56].

ARM TrustZone

ARM TrustZone is a hardware-based TEE commonly used in mobile and edge devices [24], [25]. By partitioning the system into two execution worlds—secure and normal—TrustZone enables secure services (e.g., cryptographic key storage, secure boot) to run in the protected secure world. Memory and other system resources allocated to the secure world remain inaccessible to the normal world, reducing the risk of tampering or data leakage [24]. TrustZone is well-suited for lightweight security tasks on embedded systems or IoT devices [25]. In edge AI scenarios, it can protect sensitive inference parameters or handle secure data streams, though large-scale model training often exceeds its resource capabilities [24], [25].

2.1.3 VM-Based TEEs

Intel TDX

Intel Trust Domain Extensions (TDX) protects entire virtual machines (VMs) rather than individual processes [57]. Unlike SGX's enclave-limited memory, TDX can scale to the host system's full memory capacity, subject to hardware and platform configuration [23], [57]. For instance, modern Intel Xeon servers can support up to multiple terabytes of DDR4/DDR5 memory, which TDX encrypts using Multi-Key Total Memory Encryption (MK-TME) [19]. Memory bandwidth is generally comparable to that of non-TEE environments, often reaching up to 307 GB/s depending on the Xeon generation, although some overhead arises from encryption and attestation procedures [23]. This broader memory coverage and VM-level isolation make TDX more suitable for running large AI models without severe memory-enclave constraints, as found in SGX.

AMD SEV-SNP

AMD Secure Encrypted Virtualization–Secure Nested Paging (SEV-SNP) offers CPU-level encryption and integrity guarantees for entire VMs on AMD EPYC processors [20], [21]. It encrypts guest memory with per-VM keys, and the Secure Nested Paging extension protects the memory layout from manipulation by a compromised hypervisor [21]. AMD EPYC servers can scale to large memory footprints (often up to 4 TB or more per socket, with bandwidth typically exceeding 200 GB/s on third and fourth generation EPYC), allowing memory-intensive AI workloads to run inside protected VMs [20]. Because SEV-SNP secures the full VM, applications generally do not need code changes, making it practical for containerized and virtualized AI deployments [21].

NVIDIA H100/H200

NVIDIA H100 and H200 are GPU-based TEEs that incorporate Confidential Computing features into high-performance GPU accelerators [22]. Designed for large-scale AI and HPC tasks, the H100 provides up to 80 GB of HBM3 memory, offering up to 3 TB/s of memory bandwidth, while the H200 is reported to increase both memory capacity and bandwidth for even more demanding workloads [22], [58]. This hardware-level isolation encrypts data during GPU processing, preventing unauthorized access by the host or hypervisor. The high bandwidth is critical for parallel AI training and inference, allowing substantial data throughput while still preserving confidentiality [22]. Thus, NVIDIA H100/H200 GPUs enable end-to-end security for model parameters, intermediate results, and data, without severely impacting the GPU's massive parallel computing capabilities.

2.1.4 Architectural Differences Across TEE Solutions

Although all TEEs aim to safeguard data and computation from untrusted parts of the system, their underlying approaches vary [14], [19].

Process-Level TEEs (Intel SGX, ARM TrustZone) create enclaves that isolate specific applications or services. They typically rely on a small, dedicated memory region that, if exceeded, forces paging and introduces performance overhead [14], [47], [59]. This fine-grained protection is valuable for lightweight or highly targeted security needs but may not suit large-model AI workloads [14], [59].

VM-Based TEEs (Intel TDX, AMD SEV-SNP) expand isolation to entire virtual machines, providing a larger memory footprint—fully encrypted—to support containerized or virtualized deployments [21], [60]. Performance overhead can occur when switching between encrypted and unencrypted regions, but these TEEs are better suited to large-scale computations [26].

GPU-Based TEEs (NVIDIA Hopper CC) integrate the GPU into the trust boundary, encrypting data in GPU memory and offloading massive parallel computations securely [45], [46], [61]–[63]. While this allows confidentiality-preserving AI training/inference, it also adds cryptographic overhead to data transfers across interfaces like PCIe or NVLink [62], [64].

IoT/Edge TEEs (ARM TrustZone, RISC-V TEEs) enable system-wide partitioning into secure and normal worlds but typically run on resource-constrained devices, limiting their applicability to lightweight tasks rather than large-scale AI [19], [65]–[67].

Memory usage is a key factor for performance. TEEs that use CPU enclaves usually have limited memory, while VM-based TEEs can tap into the host's full memory with encryption [24], [26]. Offloading work to GPUs speeds up AI tasks but also adds extra complexity and encryption work [23], [57], [62]. In short, process-level enclaves work best for smaller security needs, VM-based TEEs are ideal for containerized or virtual setups that require more memory, and GPU-based TEEs are essential for secure machine learning workflows [22], [23], [57].

2.2 Large Language Models

2.2.1 Machine Learning

Machine Learning (ML) is the field that enables computers to learn from data and improve through experience, rather than following fixed rules. It blends ideas from computer science and statistics, and it underpins much of modern AI and data science.

ML's rapid growth is driven by better learning algorithms, expanded datasets, and more powerful computing resources [68], [69].

A key principle in ML is to build a model (e.g., a decision tree or neural network) by adjusting parameters to minimize errors on a training dataset. This is often done using optimization techniques like gradient descent. Generalization—the ability to perform well on new, unseen data—requires methods to prevent overfitting, such as regularization or cross-validation. Another foundational idea is the bias–variance tradeoff, which balances model simplicity and complexity to reduce errors [68], [69].

ML now powers a wide range of applications across science and industry. By learning from example data rather than relying on hand-coded instructions, ML systems handle tasks such as object recognition, language understanding, and robotics. They also fuel recommendation systems, detect fraud, support medical diagnoses, optimize logistics, and find hidden patterns in large datasets. ML's ability to learn from experience enables it to solve data-driven problems at scale [68].

2.2.2 Development

Development of Large Language Models (LLMs) encompasses an end-to-end process that integrates system design, data collection, preprocessing, model architecture selection, training, and performance evaluation. The process begins with organising diverse and representative datasets to capture language intricacies. Subsequently, engineers design optimized architectures and implement scalable algorithms that support efficient learning. Careful evaluation and iterative refinement ensure the model's reliability, robustness, and adaptability to various applications [70]–[72].

2.2.3 Training

Training in the context of LLM development is the iterative process where the model is exposed to large amounts of data and adjusts its internal parameters via optimization algorithms. During training, techniques such as gradient descent and backpropagation refine weights to minimize loss functions. The procedure involves careful batch processing, regularization, and hyperparameter tuning to prevent overfitting and enhance generalization. Extensive computational resources and time are required for training large-scale models. Effective training is crucial for achieving high performance in downstream natural language tasks and ensuring that the model comprehends complex patterns in language data [70]–[72].

2.2.4 Fine-Tuning

Fine-tuning in the context of LLMs refers to the specialized retraining process applied after initial pre-training. In this phase, the model is exposed to a smaller, task-specific dataset, enabling it to adapt its generalized linguistic representations to a specialized domain. By applying a lower learning rate and selective weight adjustment, fine-tuning preserves fundamental knowledge while improving performance on targeted applications. This process is crucial for mitigating overfitting and ensuring effective domain adaptation, leading to more robust and accurate performance on tasks such as sentiment analysis, summarization, or translation. This process optimizes LLMs for variety of real-world challenges [73].

2.2.5 Inference

Inference in the context of LLMs refers to the phase where a pre-trained model processes new, unseen input data to generate predictions or responses. During inference, the LLM applies learned representations and probability distributions over its vocabulary to produce contextually relevant outputs. This phase is computationally optimized to reduce latency while maintaining accuracy and robustness under diverse applications. Inference differs from training as it does not involve weight adjustments, but solely relies on fixed parameters, ensuring scalable deployment. The process is critical in deploying LLMs for real-world tasks such as natural language understanding and generation [70]–[72].

2.2.6 Types of Models

Encoder-Only Models (Transformers as Encoders)

Encoder models use only the Transformer's encoder stack to encode input data into contextual representations. BERT is a prime example: it uses a multi-layer bidirectional Transformer encoder [73]. The encoder processes the entire input sequence simultaneously, enabling each token to attend to both left and right context. During pre-training, models like BERT mask some input tokens and learn to predict them from surrounding context (masked language modeling) capturing the input's meaning holistically [73], [74].

Encoder-only Transformers excel at understanding or analyzing text rather than generating it. After encoding, a classification layer or other task-specific head can be attached to the encoder's output [73]. This approach is effective for tasks like sentiment analysis, entailment, and named entity recognition [73], [74]. BERT's embeddings, for instance, can be fine-tuned to achieve state-of-the-art performance on question answering and language inference benchmarks [73]. Models such as RoBERTa and ALBERT

also follow this encoder-only design, focusing on language understanding. In summary, encoder models produce powerful fixed representations of text, making them ideal for downstream classification and retrieval tasks rather than free-form text generation.

Decoder-Only Models (Transformers as Decoders)

Decoder-only models use only the Transformer decoder stack, which includes masked self-attention (preventing tokens from attending to future positions) and a feed-forward network [75]. This masking enables autoregressive generation—predicting one token at a time while conditioning on previously generated tokens [76]. The architecture is optimized for language modeling, learning the probability distribution of text by predicting the next word given the previous words [75].

These transformers excel at producing fluent, coherent text in tasks such as text completion, story generation, and dialogue. The GPT series exemplifies this approach [76]. Trained on large collections to predict the next token, GPT can generate open-ended text and be adapted for summarization or question answering via prompt-based methods or fine-tuning. For example, GPT-3 (175 billion parameters) demonstrated that a pure decoder architecture can produce human-like paragraphs and even perform few-shot tasks [77]. Decoder-only models are preferred when text generation is the primary goal, leveraging their autoregressive nature to generate outputs of arbitrary length. Models such as LLaMA, GPT-4, and Claude also belong to this category.

Encoder-Decoder Models (Sequence-to-Sequence Transformers)

Encoder-decoder models combine an encoder to read the input with a decoder to produce the output. The encoder is a stack of self-attention and feed-forward layers that processes the entire input sequence into latent representations [75]. The decoder then generates the output sequence one token at a time, attending both to the encoder's output (via cross-attention) and to previously generated tokens (via masked self-attention) [75]. This design is well-suited to tasks like machine translation or summarization, where the output must be grounded in the input [77].

Such architectures handle input-output alignment effectively. For example, T5 converts text-to-text by taking an input sentence or question and producing the corresponding output, be it a translation, summary, or classification label [77]. BART likewise pairs a bidirectional encoder with an autoregressive decoder [75]. Encoder-decoder models are thus ideal for scenarios requiring both accurate input understanding and fluent output generation—such as translating documents, summarizing articles, or context-based question answering [78]. By combining a strong encoder (e.g., BERT) with a generative

decoder (e.g., GPT), they offer the best of both worlds for supervised sequence-to-sequence learning.

TEE	Type of Model	Isolation Mechanism	Memory Capacity	Accelerator Compatibility	Best Suited For
Intel SGX	CPU instructions (process-level enclaves)	Per-process enclaves in CPU	∼128 MB EPC in early SGX	CPU only; does not support accelerators	Smaller datasets, fine-grained security needs, inference tasks
Intel TDX	CPU virtualization extensions for confidential VMs	Entire VM runs in a trusted domain via virtualization extensions	Scales to host system's full memory (multiple TBs)	Supports accelerators (GPUs, FPGAs, TPUs)	Containerized/ virtualized workloads with big memory demands
AMD SEV-SNP	CPU virtualization on AMD architecture	Encrypted guest memory for entire VM with secure nested paging	Supports large memory footprints (up to 4 TB or more per socket)	Supports accelerators (GPUs, FPGAs)	Virtualized environments, cloud/federated setups requiring robust isolation
NVIDIA Hopper CC (GPU-based)	GPU hardware modifications + secure PCIe/NVLink	Encrypted GPU memory providing GPU-level enclaves	80 GB HBM3 memory on H100 (∼141 GB on H200)	Natively supports GPU acceleration; not designed for other accelerator types. Usually paired with VM-based TEEs	Large-scale parallel computations, deep learning training/inference
ARM TrustZone	Hardware-based secure world integrated into ARM CPUs	System-wide isolation by partitioning the processor into secure and normal worlds	The secure world is allocated a fixed, relatively small portion of system memory	Primarily CPU-based; does not support accelerators	Low-power, mobile, IoT, and edge devices requiring secure boot, DRM, key management, etc.

Table 2.1: Comparison of TEE solutions.

Chapter 3

Analysis

This chapter begins by defining the AI model lifecycle in detail, breaking it into the key stages where vulnerabilities are most likely to occur. We then analyze the various threats that commonly take place at each stage, exploring how Trusted Execution Environments (TEEs) can mitigate—or fail to mitigate—specific risks. Finally, we conclude by pinpointing which phase is the most vulnerable overall and discuss the potential reasons behind it.

3.1 Defining the AI Model Lifecycle

Previous research typically divides the machine learning lifecycle into broad phases like development, training, and inference. However, for this analysis we needed a more detailed approach, one that establishes stricter definitions and breaks the lifecycle down into smaller subphases [70]–[72], [79]. This granularity allows to precisely identify where and why specific attacks occur. While those earlier works provided valuable guidance, their broader categorizations left some ambiguity about the exact points of vulnerability. Because of that, we developed the following definition to offer clearer boundaries for analyzing security threats at each stage. The following definition is focused on a centralized model lifecycle, which significantly differs at certain phases from the decentralised one, which is outside of the scope of this paper.

3.1.1 Centralized Model Lifecycle

1. **The Development Phase** consists of two subphases: **model selection** and **data collection and preparation**. This phase is where the LLM is created. Model Selection defines choosing an appropriate base model considering factors

like architecture, scalability, and performance. In centralized settings, the model is typically designed to leverage a complete, aggregated dataset, ensuring consistency across training. However, this requires robust security measures to prevent breaches, as all data is stored in one location. We then go on to data collection and preparation, where we gather, clean, and normalize data in a unified repository. The centralized approach enables comprehensive data preprocessing and precise quality control, but it may increase risks related to privacy and data leakage.

2. **The Training Phase** consists mainly of two subphases: **pre-training** and **fine-tuning**. The rest of the subphases, such as reinforcement learning, are optional. In this phase, the model is provided with data and trained on it. Pre-training is where the model is initially trained on a large-scale dataset using self-supervised learning. In a centralized system, this process benefits from high-performance computing infrastructure, allowing for the efficient processing of extensive data. Fine-tuning, on the other hand, is where the model is refined with domain-specific data to enhance its applicability to particular tasks (e.g., summarization or question answering). Fine-tuning ensures that the model adjusts uniformly to specialized contexts. In some cases, reinforcement learning (often with human feedback) is used to align the model's behavior with ethical and functional goals. Since the training environment is controlled and centralized, it allows for direct implementation of optimization strategies and continuous improvement cycles. Other types of learning can be supervised, unsupervised, etc., depending on the purpose and goal for the model.

3. **The Evaluation Phase** is where we do **performance assessment** and **iterative improvements** on the trained model. Performance Assessment is where the trained model is evaluated using a set of defined metrics (accuracy, coherence, etc.) on a separate validation dataset. Evaluation leverages the full scope of collected data, yielding reliable performance benchmarks. When this is completed, the next step would be to do iterative improvements based on assessment outcomes. Here, developers iterate on the model architecture, training data, and hyperparameters to optimize performance. This setup enables quick re-training and uniform updates across the entire dataset.

4. **The Deployment Phase (Inference)** consists of **integration** and **optimisation**. During integration the model is integrated into a production environment (e.g., cloud services, web APIs) where it begins making real-time predictions. Deployment ensures that all users interact with the same model instance, simplifying maintenance and consistency. This is where the inference of the model officially starts. Most papers refer to this whole phase as just inference. Once the model

is up and running, shortcomings and bottlenecks can be recognized, so the model needs to be optimized. This is where optimization techniques such as quantization, pruning, and hardware acceleration (e.g., GPUs or TPUs) are applied, so that efficient real-world inference—the process of using a trained model to make predictions or generate outputs from new data—can be ensured. These optimizations reduce latency and resource consumption during high-demand usage.

5. **Monitoring and Maintenance** is the last phase, made up of **Continuous Monitoring** and **Updates and Refinements**. During monitoring, the model's performance is continuously tracked using logging and analytics. This allows for the detection of issues like model drift, where the performance degrades over time due to changing data patterns. Regular updates and refinements (including retraining or fine-tuning) are applied based on new data or feedback. Maintenance ensures that improvements are uniformly distributed and any security vulnerabilities are patched promptly. From here, developers and researchers can always go back to the training or deployment phase.

3.2 Security Threats

We gathered these security threats from related work based on their relevance to the AI model lifecycle, adopting the categorization from "SoK: A Systems Perspective on Compound AI Threats and Countermeasures" [80], because it effectively met our requirements. In addition, we extended their framework by describing the stages in the AI model lifecycle where these attacks occur and assessing whether they can be mitigated by TEEs or not. Specifically, we divide the identified threats into three main layers: software-level, hardware-level, and algorithmic/ML-level, based on what layer they initially attack.

We also clarify that "white-box" attacks assume full access to model parameters, while "black-box" attacks rely on only the model's outputs. Unless we specify a particular subphase (e.g., data collection vs. active training), the threat can emerge at any relevant point in that "larger" phase. Finally, these attacks often do not exist in isolation. Additionally, if the same attack can take place at two different phases we provide an example of how it manifests at each specific phase. Adversaries often combine them for greater impact, a topic we address briefly in the last section on combining attacks across layers.

3.3 Software / Application-Layer Threats

To begin addressing software/application-layer threats, we note that the complexity of hypervisors/OS can hide both known and undiscovered zero-day exploits, creating multiple potential entry points for attackers since they contain a large Trusted Computing Base (TCB). Even if many vulnerabilities are publicly reported and fixed, unknown flaws can still leave systems exposed, potentially leading to unauthorized access or other security breaches [63]. These attacks can happen at any stage and are partially mitigated by TEEs, which isolate guest memory from hypervisors/OS. However, unknown firmware or microcode bugs within the TEE itself still pose threats, because trust is moved to hardware/firmware, which can also harbor undiscovered vulnerabilities.

Continuing with threats related to privileged software, untrusted privileged software (e.g., hypervisors, OS processes) can observe or modify memory unless TEE protection is in place. Confidential Computing (CC) blocks hypervisor access to system and GPU memory [81], safeguarding code and data from OS compromises, malicious administrators, or physical attacks [39]. By design, TEEs treat privileged host software as untrusted, enforcing hardware-level isolation to protect data during development, training, and deployment. Without this protection, compromised OS processes could tamper with models or access logs and memory snapshots.

In a similar vein, privileged insiders (e.g., cloud administrators, employees) can leak confidential data—intentionally or accidentally—causing breaches, fraud, or IP theft, while untrusted cloud providers with physical or administrative control pose similar threats. NVIDIA's CC isolates sensitive computations and protects against unauthorized memory access or certain side-channel attacks [26], [63], [79]. Although TEEs prevent untrusted providers from viewing memory via hardware-level encryption, their confidentiality depends on secure provisioning and attestation. If a malicious tenant insider gains legitimate TEE privileges, or if code within the TEE is compromised, sensitive data remains vulnerable throughout the model's lifecycle.

Looking further into software supply chains, supply-chain attacks introduce backdoors or data-exfiltration logic into AI systems (e.g., pip or conda package typosquatting). Attackers may inject trojaned or malicious packages that leak sensitive model data or enable unauthorized code execution [79], [80]. TEEs partially mitigate this threat by ensuring the code running inside the enclave is exactly as delivered (via attestation). However, TEEs cannot verify if that code is inherently malicious; they simply prevent host-level tampering. During development, Trojaned libraries or container images can exfiltrate data or embed backdoors, while in deployment, malicious dependencies in serving frameworks can leak model weights.

Continuing with external service interactions, exploiting insecure APIs, endpoints, or misconfigured services can grant attackers unauthorized access or privileges [26]. TEEs protect memory contents from host-level reads/writes, but an insecure API can still be exploited outside the TEE. This threat typically appears during integration or deployment, where flawed or badly configured APIs let attackers bypass application logic.

Regarding deeper privilege misuse, privilege escalations exploit bugs or misconfigurations in frameworks, drivers, or access control lists to gain higher-level permissions [80]. TEEs partially mitigate these attacks by isolating the guest environment from the host, restricting the damage from external privilege abuse. Still, vulnerabilities within TEE-managed software remain exploitable if the code inside the enclave is itself buggy. During development/training, attackers can escalate privileges and access sensitive training data; in deployment, they can steal or alter the served model.

Next, tampering with data at the software layer involves forging or rewriting in-flight model parameters or user inputs stored in untrusted buffers, which can compromise AI outcomes [80]. TEEs partially mitigate such software-layer tampering by requiring memory integrity for in-flight data but cannot prevent logical or application-level misconfigurations. Attackers can overwrite datasets or weights during training to degrade AI performance, or corrupt logs and intermediate data after deployment if the environment is misconfigured.

Moving on to encryption-based issues, replay attacks exploit reused ciphertext or unrefreshed encryption states—e.g., replaying old encrypted parameters—to corrupt training updates or revert inferences [35]. TEEs alone do not inherently block replays if stale ciphertext is accepted. In practice, many confidential computing frameworks enforce nonces or IV counters (like AES-GCM with incrementing IVs) to prevent these issues [63], [79]. If done properly, TEEs can mitigate replay attacks, especially during training or deployment, by rejecting old or duplicated ciphertext.

Switching to user-driven vulnerabilities, input (prompt) injection targets LLMs or other AI pipelines by crafting malicious prompts that bypass safety checks and potentially achieve code injection or SSRF [79], [80]. TEEs do not mitigate input injection, since malicious inputs are executed exactly as provided. These attacks typically emerge during deployment/inference when users supply prompts to LLMs.

Looking at availability concerns, resource exhaustion attacks (DoS, ReDoS, forced system panics) overwhelm the system to disrupt service availability [80]. TEEs focus on data confidentiality and integrity, not on sustaining system availability under heavy load, so these attacks remain unmitigated. They can appear during training or evaluation by overloading computations or in deployment by flooding inference requests.

Addressing broader advanced threats, targeted cyber-attacks (APTs, malware, etc.) use spear phishing, password compromise, or zero-day exploits to infiltrate cloud or on-premise environments [26]. TEEs help shield data in memory from direct exfiltration but do not block phishing or OS-level hijacking. Once an attacker runs code inside the tenant's TEE session, they can access data legitimately used by the application. Such attacks can happen in any phase.

On a more encouraging note, TEEs fully mitigate unauthorized memory access at the software level by encrypting application buffers in RAM or GPU memory. Even if an attacker gains host privileges, they cannot read or dump that memory. Without a TEE, malicious processes could inspect memory snapshots to steal weights or data during training, or inference inputs at runtime.

However, returning to internal coding pitfalls, memory safety flaws (e.g., buffer overflows, use-after-free) let attackers read or corrupt sensitive data within the TEE [80]. TEEs do not fix vulnerabilities in user code and cannot prevent an attacker from exploiting those bugs to manipulate data inside the enclave. These attacks can occur during development, training, or deployment if the application code or libraries have memory safety issues.

In a similar way, side-channel attacks (software-level) remain partially mitigated. While TEEs encrypt memory structures, attackers can still glean information via timing or cache-based channels. During pre-training or inference, subtle memory and cache usage may leak partial model parameters or user data if an attacker can measure performance counters. Although TEEs limit direct data reads, advanced side channels can still extract valuable information.

3.4 Hardware / Physical Layer Threats

Turning to hardware/physical threats, physical attacks on hardware involve local techniques like attaching probes to DRAM buses or tampering with power lines—generally beyond the TEE's threat model. Although hardware encryption can thwart cold-boot or bus snooping, a fully determined attacker with specialized equipment can still breach the device's physical security. Such attacks can occur during development, training, deployment, or monitoring.

Likewise, bus snooping or dumping memory (including cold-boot attacks) can leak proprietary parameters or private inputs [80]. HETEE and on-package HBM encryption help defend against bus probing, but if attackers obtain encryption keys or breach the

hardware root of trust, the protection may fail. During development/training, attackers can extract partial weights; in deployment, they can reveal final models or user inputs.

Expanding on hardware interface threats, bus hijack or I/O intercept targets untrusted interconnects like PCIe or NVLink [35]. TEEs rely on hardware-based encryption and integrity checks for off-chip traffic, but if an attacker has physical control or compromised firmware, advanced man-in-the-middle attacks are still possible. These issues arise in training and deployment alike, potentially exposing updates or inference data in transit if not properly encrypted.

Addressing microarchitectural leakage, side-channel attacks (hardware-level) observe power usage, electromagnetic signals, or performance counters to deduce sensitive information [80]. TEEs mitigate some of these vectors by encrypting memory and restricting certain counters, but timing-based or EM-based leaks can still occur. During training, side channels can reveal model parameters; in deployment, attackers can glean sensitive user inputs from runtime activity.

Building on memory-based exploits, rowhammer (bit-flip) attacks repeatedly access certain DRAM or HBM rows to induce bit flips in adjacent cells [80]. Memory encryption and integrity checks can detect or prevent silent corruption, but they do not fully eliminate the risk of system instability or crashes. Attackers may degrade model accuracy or sabotage inference by flipping bits in the training or deployment stage.

Similarly, fault injection (with laser or voltage glitches) physically alters hardware behavior, causing transient computation errors in model parameters or logic [80]. TEEs do not defend against such invasive physical modifications. Attackers can sabotage training or cause incorrect inferences by forcing hardware faults at critical steps.

Turning to firmware integrity concerns, firmware tampering (hardware trojans) adds malicious logic into CPU/GPU firmware or board components [80]. If the hardware root of trust is compromised, TEEs cannot detect or block malicious firmware that bypasses the normal security chain. These trojans might become active during training or deployment, silently leaking data or corrupting computations.

Along the same lines, leftover GPU memory exploitation occurs when prior computations leave sensitive data in memory that an attacker can later retrieve [63]. GPU TEEs encrypt memory allocations to reduce leftover-data leaks, but malicious firmware or advanced side-channel methods can still bypass these measures. During training or deployment, memory not fully cleared or re-provisioned can be harvested by attackers in a multi-tenant setting.

Further complicating multi-tenant clouds, insufficient isolation in shared hardware can let one tenant steal or corrupt another tenant's data [26]. TEEs encrypt memory to protect data from the host or co-tenants, but side channels in caches or scheduling remain possible. Training with multiple tenants on the same machine or deploying shared GPU inference can thus create data-leak or data-corruption risks.

Lastly, unverified hardware accelerators such as custom TPUs or NPUs undermine TEE guarantees if the accelerator itself is compromised or tampered with [61], [81]. While CPU or GPU TEE modes can encrypt memory traffic, an untrusted or unverified piece of hardware might leak data or subvert computations. These threats surface especially during training or deployment when external accelerators handle sensitive data without TEE-level attestation.

3.5 Algorithmic / ML-Layer Threats

Shifting focus to algorithmic threats, model/data poisoning occurs when attackers inject malicious samples into training data or tamper with local gradient updates, degrading model accuracy or installing hidden backdoors [37], [46]. TEEs partially mitigate infrastructure-based tampering, but cannot block corrupted inputs from legitimate data providers. These vulnerabilities appear during development, training, or monitoring, where adversaries slip in malicious data over time.

Within federated learning, a related risk is the "lazy" participant, who lies about dataset size or composition [36]. Since TEEs simply process whatever data they receive, they cannot detect dishonest claims. Although TEE memory encryption protects data from the host, it does not verify data authenticity. Attackers exploit this during development/training phases.

Turning to concurrency issues, scheduler/concurrency-based integrity attacks manipulate thread scheduling in multi-threaded or distributed ML pipelines to force incorrect computations [46], [81]. TEEs help isolate guest memory but still rely on host-level scheduling for thread management. Consequently, an untrusted OS could cause concurrency conflicts that degrade the model's integrity during training or inference.

Moving on to inference-time privacy threats, model inversion or data reconstruction extracts private training information from the model's outputs or gradients [37], [46]. TEEs partially mitigate host-based snooping of internal states, but they cannot prevent a remote user (who queries the model) from deducing data via normal outputs. Hence, inversion emerges mainly at deployment/inference.

Similarly, membership inference attacks (MIA) let adversaries determine whether a data record was used in training [37], [46], [80]. TEEs do not mitigate MIA because these attacks hinge on query responses from the model. Once again, the TEE keeps data hidden from the cloud host, but does not alter the model's externally visible predictions.

Stepping beyond membership, attribute inference targets private features (e.g., demographics) from the model's outputs [37]. TEEs partially mitigate direct host-level scrutiny but cannot stop a malicious user from inferring attributes via normal inference queries. This typically occurs at deployment, where the attacker leverages repeated queries to reveal sensitive information.

Lastly, adversarial example attacks craft subtle perturbations in inputs to cause misclassifications [37], [46], [82]. TEEs do not protect against these input-based exploits, as they focus on safeguarding memory and data in hardware—not on input validation or model-level robustness. Adversaries can thus supply maliciously perturbed data during deployment/inference to induce incorrect predictions.

3.6 Discussion

3.6.1 Most Vulnerable Phases

The training stage brings together massive volumes of data, distributed or cloud-based infrastructures, and complicated software pipelines—greatly expanding the attack surface. As the model ingests and modifies its parameters based on input data, adversaries can corrupt the process either by injecting poisoned samples that implant hidden backdoors or by exploiting weaknesses in multi-tenant environments and third-party frameworks. Because of this complexity, even minor oversights (e.g., "untrusted buffers" or "trojaned libraries") can allow an attacker to compromise or exfiltrate the model before it's fully formed, leaving little trace. Data poisoning stands out as a particularly severe threat here: attackers slip malicious inputs into training sets to skew future model predictions—often undetected—while infrastructure misconfigurations in cloud clusters or orchestration services can expose training data or enable tampering. The sheer width of infrastructure, datasets, and software used, along with reliance on shared accelerators, makes training severely vulnerable and impactful stage: once the model is compromised at its foundation, every subsequent phase inherits the malicious modification.

Although the model's core logic is set once training completes, the deployment (inference) phase subjects it to continuous, real-world interaction—placing the model within an externally accessible environment. Attackers exploit this exposure by launching adversarial queries or manipulative inputs that bypass many lower-level protections.

Tactics such as input (prompt) injection can trigger malicious code execution or leak internal details, while adversarial examples—tiny, often undetectable input perturbations—can cause misclassifications or errant outputs. Inference also invites model extraction attacks, where adversaries repeatedly query the AI system to clone its behavior, undermining intellectual property. Moreover, if the host environment is compromised, memory scraping or snapshotting can reveal user data or model parameters in real-time. The resulting data breaches or logic manipulations frequently go unnoticed because legitimate queries and malicious queries arrive through the same interface. Thus, while training vulnerabilities threaten the model's internal integrity, deployment-time attacks exploit the external-facing nature of inference to produce immediate and potentially large-scale consequences.

From a software perspective, training code depends heavily on third-party components, raising supply-chain risks and vulnerabilities in orchestration frameworks, whereas in deployment, insecure endpoints or flawed APIs let attackers escalate privileges or exfiltrate outputs. Hardware threats also cut across these phases. During training, adversaries can use multi-tenant GPUs or advanced side-channel techniques (e.g., rowhammer) to extract or corrupt the model's parameters. In deployment, the model must run on hardware potentially shared with untrusted code, enabling timing or bus-contention attacks that leak user inputs or partial weights in real time. Algorithmically, poisoning and backdoor insertion are the biggest concerns in training, while at inference the model can be systematically probed via white-box or black-box approaches to reveal private data, craft adversarial inputs, or replicate the model. Altogether, these layers make the training and deployment phases especially exposed to multidimensional attacks that each require specific defenses.

3.6.2 Rest of the Phases

While development (model creation/data prep) faces moderate exposure to malicious libraries or unsafe code, these threats are generally within the project team's control—vulnerabilities here often stem from insecure supply chains or poor dependency management. Evaluation (testing/validation) is less open to external input, so its main danger is failing to detect attacks or backdoors that originated in training, rather than facing major new exploits. Finally, monitoring and maintenance can miss ongoing attacks (e.g., slow data drift or unauthorized updates) if logs and security patches are not diligently managed; although it is not as high-profile a target as training or deployment, attackers can still exploit weak logging or stale infrastructure configurations to remain undetected or reintroduce corrupted data.

3.6.3 TEEs Strengths and Limitations

TEEs, memory encryption, and strong cloud segmentation help but do not fully solve issues in training and inference. They cannot detect malicious data from authorized sources or alter a model's visible decision boundaries—leaving adversarial queries and model extraction unchecked. Logging and monitoring may fail if training sets or APIs are poorly secured. Thus, robust data governance, validated supply chains, thorough testing, and continual monitoring must complement TEE isolation. Although TEEs protect data confidentiality, enforce in-memory integrity, and isolate tenants from the host, they do not address all AI security challenges (e.g., data poisoning, prompt injections, or side-channel leaks) nor ensure trust in the hardware supply chain. Ultimately, traditional software hardening, rigorous data governance, side-channel defenses, and careful model validation remain critical alongside TEEs.

Name of Attack	Phase of AI Model Lifecycle	TEE Mitigation
Zero-day OS/hypervisor vulnerabilities	Any phase	Partially mitigated
Untrusted privileged software	Development, Training, Deployment	Partially mitigated
Privileged insiders	Any stage	Partially mitigated
Supply-chain attacks	Development, Training, Deployment	Partially mitigated
Insecure APIs or misconfigured endpoints	Integration, Deployment	No
Privilege escalations	Development, Training, Deployment	Partially mitigated
Tampering with data at the software layer	Development, Training, Deployment	Partially mitigated
Replay attacks	Training, Deployment	Partially mitigated
Input (prompt) injection	Deployment/Inference	No
Resource exhaustion / DoS	Training, Evaluation, Deployment	No
Targeted cyber-attacks (APTs, malware, etc.)	Any phase	Partially mitigated
Unauthorized memory access	Training, Deployment	Yes
Memory safety flaws (buffer overflows, etc.)	Development, Training, Deployment	No
Side-channel attacks (software-level)	Any phase	Partially mitigated
Physical attacks on hardware	Development, Training, Deployment, Monitoring	No
Bus snooping / dumping memory	Development, Training, Deployment	Partially mitigated
Bus hijack / I/O intercept	Training, Deployment	Partially mitigated
Side-channel attacks (hardware-level)	Training, Deployment	Partially mitigated
Rowhammer (bit-flip) attacks	Training, Deployment	Partially mitigated
Fault injection (laser, voltage glitching)	Training, Deployment	No
Firmware tampering (hardware trojans)	Training, Deployment	No
Leftover GPU memory exploitation	Training, Deployment	Partially mitigated
Insufficient isolation in shared hardware	Training, Deployment	Partially mitigated
Unverified hardware accelerators	Training, Deployment	Partially mitigated
Model/Data poisoning	Development, Training, Monitoring	Partially mitigated
"Lazy" participant in federated learning	Development, Training	No
Scheduler/Concurrency-based integrity attacks	Training, Inference	Partially mitigated
Model inversion / data reconstruction	Deployment/Inference	Partially mitigated
Membership Inference Attacks (MIA)	Deployment/Inference	No
Attribute inference	Deployment	Partially mitigated
Adversarial example attacks	Deployment/Inference	No

Table 3.1: Summary of Attacks, Phases, and TEE Mitigation

Chapter 4

Experiment Design

This chapter explains how to train and run inference on a large language model (LLM) in a Trusted Execution Environment (TEE). We discuss choosing a model suited to the available TEE-enabled hardware and how factors such as model size, GPU usage, and TEE enabled phases affect performance. We then outline the essential hyperparameters (e.g., batch size, number of epochs) to adjust and the performance metrics (e.g., Time To First Token, ITL, TPS, Latency, QPS) to monitor during both training and inference. Finally, we highlight how to measure TEE-specific overhead versus general factors like I/O or network latency, forming a clear methodology for understanding LLM performance under a TEE.

4.1 Hardware, TEE Availability and Model Selection

Selecting a suitable LLM first requires hardware that can accommodate both the model's memory requirements and the available TEE to use. If only CPU-based TEEs (e.g., Intel TDX or AMD SEV-SNP) are available [12], [13], [83], smaller models or CPU-friendly workloads may be necessary, as large-scale GPU acceleration is not protected by the TEE. In contrast, newer GPUs like the NVIDIA H100 include their own hardware-backed GPU TEEs, allowing both training and inference to run confidentially on the GPU [10], [16]. A fully end-to-end secure pipeline can combine CPU TEEs and GPU TEEs so that data remains encrypted throughout, but each added layer increases cryptographic overhead [10], [35].

Because TEE overhead varies greatly with model size, batch size, and which phases (training, inference, or both) run in TEE mode, different hardware setups will result in different performance trade-offs. For instance, bigger models or heavier workloads

on a GPU TEE tend to spread out the cost of encryption more effectively than small or I/O-bound tasks, which may see a higher amount of overhead [62], [81]. Similarly, enabling TEEs across multiple GPUs multiplies encryption and authentication steps for distributed training—especially if gradients must be protected in transit—further impacting performance [35]. In practice, picking an LLM that comfortably fits within the secure memory of the selected CPU or GPU TEE, limiting excessive offloading, and deciding where (CPU,GPU or both) and when (training, inference or both) to enable TEE will all determine the overall overhead [10], [39], [64]. Using high-bandwidth interconnects like NVLink or NVSwitch is recommended if possible to ensure that any observed slowdowns are primarily due to encryption and authentication overhead rather than network limitations [35], [57], [62], [64].

A key consideration is to pick an LLM that fits the available hardware—both in terms of memory and the presence of a TEE on the CPU, GPU, or both. The model must be sized appropriately so that it can be trained and inferred fully within the secure memory available. Otherwise, repeated offloading or swapping outside the enclave undermines security and increases overhead [12], [13], [32], [37]. Both model size and hyperparameters (especially batch size) heavily influence how effectively TEE overhead is divided among individual processing steps—like encrypting inputs, securely transferring data between the CPU and GPU, decrypting outputs, and the associated communication or gradient exchange steps [32], [35], [52].

If only CPU-based TEEs are available, a model up to a few hundred million parameters may be necessary to fit into encrypted memory without excessive overhead [12], [13]. If GPU TEEs (like NVIDIA H100) are available, larger-scale models (up to billions or tens of billions of parameters) become a reasonable choice, provided that GPU-secured memory can accommodate the model weights and activations [10], [16]. The goal is to pick an LLM that uses as much secure memory as possible without pushing memory usage so high that data repeatedly goes outside the TEE boundary. TEE-specific factors also include the model size, batch size, the phase of the model lifecycle when TEE is enabled, the number of available GPUs, and the workload ratio(compute vs I/O) [35], [37], [81]. Large models require more encrypted transfers (weight loading and gradient exchange), but often hide overhead better during long-running computation, whereas smaller models do less processing, making the encryption overhead more noticeable [62].

Additionally, a larger batch size processes more data in each iteration or inference call, which spreads out TEE overhead over more samples, while very small batches repeatedly trigger encryption costs [35], [52]. The model's phase—training or inference—also plays a crucial role. Training in TEE mode may add overhead at every gradient update and communication step, especially in multi-GPU setups [35], [57]. In inference,

the overhead mostly affects time-to-first-token and inter-token latency, though these delays are reduced with large models or long sequences [62]. Moreover, scaling out to multiple GPUs can multiply encryption costs, which can lead to significant slowdowns if each GPU-to-GPU transfer is encrypted [35]. Lastly, if the workload is I/O-bound (many short prompts or small batches), TEE overhead is more pronounced, whereas in compute-bound workloads (large batches or long sequences), the encryption cost is relatively amortized over more computation [32], [37], [52].

4.2 Performance Metrics

Time To First Token (TTFT), Inter-Token Latency (ITL), Tokens Per Second (TPS), Latency (per query), and Queries Per Second (QPS) are typically the five main metrics collected during inference. TTFT measures the delay before the first token appears, thus capturing any startup overhead such as secure memory setup or prompt encryption. ITL tracks the time gap between tokens after the first one, indicating whether additional encryption steps slow down token-by-token generation. TPS reflects the model's overall token production rate, which can drop if TEE-related I/O stalls decoding. Meanwhile, per-query Latency is the total time from request arrival to the final output token, incorporating both TTFT and ITL. QPS shows how many queries can be handled per second while meeting a specific latency target, making it a key indicator of system throughput under load [62].

When a TEE is in place, TTFT and ITL often become the most telling signs of overhead on latency, because they highlight added delays in prompt processing and token generation. Encryption or enclave-switching steps can significantly extend the time it takes to deliver that first token, which is why TTFT is sensitive to TEE configuration. Likewise, if each new token must be encrypted or decrypted before being passed along, ITL rises and TPS can fall accordingly. QPS is affected more broadly, since prolonged processing time per query will inevitably reduce the system's capacity to handle multiple requests at once. These metrics are usually measured by instrumenting the inference pipeline with timers at key points (e.g., when a query arrives, when the first token is emitted, and when the final token is produced), and by aggregating the results across many runs. By comparing TEE-enabled scenarios with TEE-off baselines, the performance impact of confidential computing can be isolated [62].

In a training context, related metrics include iteration time (batch-level latency), throughput (tokens or samples per second), and an equivalent notion of QPS that tracks how many training steps can be completed in a given time. TTFT here is less critical unless we specifically measure the overhead of loading the first batch into the enclave.

In general, what matters is whether encryption and secure memory management slow down the forward/backward passes (affecting iteration time) or reduce the frequency of updates (hurting throughput). Evaluating TEE-on vs. TEE-off runs highlights how additional security measures might stall training loops or degrade scalability [26], [63].

4.3 Batch Size, Epochs, and Iterations per Epoch

Alongside performance metrics, batch sizes, epochs, and iterations per epoch, need to be considered, especially in the training process. A batch size refers to the number of data samples (or tokens, depending on the framework) that the model processes before updating the parameters. An epoch is one complete pass over the entire training dataset. Iterations per epoch simply specify how many batches fit into one epoch—if the dataset is split into smaller chunks of size B, and the dataset has N samples in total, then we will have

$$\left\lceil \frac{N}{B} \right\rceil$$

iterations per epoch [36], [69], [84], [85].

Using a TEE can impact these choices, since larger batch sizes might require more secure memory, and frequent enclave transitions can add overhead if you opt for smaller batches. There is usually a balance: bigger batches can reduce the number of enclave entries per epoch, potentially cutting some overhead, but they also demand more secure memory capacity, which might be limited. On the contrary, smaller batches fit more easily into the TEE's memory constraints and can reduce per-batch latency, but they may require more frequent switching and encryption steps, lowering overall throughput [85], [86].

In practice, batch size is typically picked based on hardware memory constraints (including any additional TEE restrictions), while epochs and iterations per epoch are chosen according to the total data size and the desired granularity of updates. Preliminary experiments are often run to find a point that maximizes GPU/CPU utilization without hitting TEE memory limits or causing excessive overhead from repeated secure operations. This process of tuning batch sizes, epochs, and iterations per epoch is critical for keeping training times under control while still benefiting from confidential computing [26], [63].

Chapter 5

Implementation

This chapter describes how the experimental design from Chapter 4 can be implemented. It details the configuration of hardware and software and explains the model training, inference and testing under various batch sizes, epochs, and TEE modes. The primary goal is to validate the methodology and, in Chapter 6, present results that quantify TEE overhead during both training and inference.

5.1 Hardware and TEE Availability

When conducting confidential computing experiments are TEEs are supposed to be ebanled across all relevant layers, beginning at the BIOS or firmware level and extending through the hypervisor, container runtime, and any applicable GPU hardware. On platforms supporting AMD SEV-SNP or Intel TDX, TEE features should be activated in the BIOS and configured at the operating system level [11], [12], [20], [83]. For GPU-based TEEs, such as those in NVIDIA H100 or H200 accelerators, additional hardware-specific settings must be configured to ensure data remains protected throughout GPU computations [10], [16], [22]. This includes updating TEE-capable firmware and BIOS on both the system and GPUs, installing compatible drivers, adjusting CUDA or other libraries, and configuring GPU enclaves for isolation. When AMD SEV-SNP or Intel TDX is also utilized, CPU-level TEEs should be aligned with GPU enclaves to sustain continuous protection of data transfers between CPU and GPU memory.

At the software layer, the virtual machine required by TEEs such as AMD SEV-SNP or Intel TDX must be configured by the researcher. One way to achieve this is through Kata Containers, which encapsulates workloads in lightweight virtual machines (VMs) rather than standard namespaces [11], [12], [20], [83], [87]. By toggling TEE modes on

or off (e.g., unencrypted vs. SEV-SNP/TDX-encrypted containers), it becomes possible to collect performance metrics for tasks such as training and inference on LLM models. This approach allows researchers to quantify TEE overhead in an end-to-end encrypted environment while preserving the convenience of familiar container workflows [26], [65].

In addition, ensuring robust data transfer capabilities is critical to minimize any confounding network or I/O slowdowns. High-bandwidth interconnects—such as NVLink, PCIe, or InfiniBand—should be properly configured and employed wherever possible, preventing peripheral throughput constraints from overshadowing TEE-related overhead. By maximizing transfer efficiency, researchers can better isolate performance impacts arising from confidential computing features [10], [16], [22], [81].

5.2 Model and Dataset Selection

When evaluating transformer-based architectures, selecting a widely recognized model from the literature is recommended, so alignment with established benchmarks can be assured, allowing for direct comparisons with prior work. This process involves verifying that the chosen model—whether encoder-only, decoder-only, or encoder-decoder—is compatible with the available hardware resources in terms of computational power and memory capacity. If no accelerators are employed, the model's size should be adjusted accordingly so that CPU-based training and inference remain feasible.

After finalizing the model, a dataset must be identified based on the model's architecture and intended task—encoder-only variants, for instance, might pair well with paraphrase detection, whereas encoder-decoder versions are suited for translation or summarization. Once the dataset is chosen, tokenization should be applied (e.g., via libraries such as Hugging Face Transformers) and stored for streamlined retrieval [88]. During this stage, factors such as input truncation, padding, and special tokens should be accounted for, to ensure compatibility with the selected architecture's requirements. This approach minimizes redundant preprocessing steps and helps with consistency in evaluating performance across diverse experimental setups.

Selecting an appropriate deep learning framework is crucial for streamlining both the development and execution of a chosen model. Each framework has unique strengths. PyTorch offers a dynamic computation graph, making it highly flexible for rapid prototyping and debugging. TensorFlow provides enterprise-level deployment capabilities and an extensive toolset suitable for production environments. Regardless of the chosen framework, it is vital to ensure compatibility with underlying hardware, drivers, and libraries (e.g., CUDA or ROCm) to avoid environment conflicts and preserve accurate benchmark results [89], [90].

5.3 Testing and Evaluation Process

When conducting training experiments to evaluate TEE overhead, different batch sizes (e.g., 8, 16, 24, 32) and number of epochs (e.g., 1, 3, 6) should be used to capture performance under different workload scenarios. Key metrics such as Time to First Training Step (TTFT), per-iteration latency, overall training time, and model convergence should be collected in both TEE-enabled and TEE-disabled modes [11], [69]. Storing these data points in dedicated output files (e.g., CSV or JSON) simplifies post-processing and allows for direct comparisons.

Python scripts or similar tools can be used to parse these logs, generate plots, and highlight areas where TEE functionality imposes notable overhead. By systematically gathering and analyzing these metrics, researchers gain insight into both raw performance and the specific stages where confidential computing incurs overhead.

A similar approach applies to inference: by testing different batch sizes (e.g., 8, 16, 24, 32) to capture a range of concurrency levels. Crucial metrics include Time to First Token (TTFT), Inter-Token Latency (ITL), Tokens per Second (TPS), and Queries per Second (QPS) [62]. As with training, these metrics should be recorded in an automated fashion (e.g., output files) to make aggregation and visualization using Python scripts easier. By contrasting metrics from TEE-enabled and TEE-disabled runs, the impact of confidential computing on inference responsiveness and throughput can be isolated, quantifying the trade-offs involved in securing model outputs [26], [62], [63].

Lastly, to ensure the successful execution of these experiments, researchers must verify that all required software libraries (e.g., TEE-enabled drivers, Python packages, container runtimes) are compatible with the available hardware. Proper installation and configuration of these components help avoid bottlenecks or incompatibilities that could affect performance measurements.

Chapter 6

Evaluation

In this chapter, we describe how we evaluated the performance impact of running a BERT model inside a Trusted Execution Environment (TEE). Our main goal was to measure the overhead introduced when SEV-SNP encryption is enabled, compared to a baseline that runs without TEE. We explain the experimental setup, the hyperparameters tested, and the performance metrics tracked for both training and inference. Finally we conclude with a discussion on our findings and related work.

6.1 Setup

Our experiments ran on an AMD EPYC 9364 server node equipped with 32 cores at 3.8GHz, 768GB of RAM, and up to 100Gbit/s network interfaces. This hardware supports AMD SEV-SNP [12], [83], providing transparent memory encryption inside virtual machines for confidential execution. We enabled SEV-SNP at the BIOS and operating system levels and deployed Kata Containers to encapsulate our workloads within secure, lightweight VMs [13], [83]. The TEE mode was determined by selecting the appropriate configuration file: `kata_config: 'kata-configuration-snp'` for TEE on, and `kata_config: 'kata-configuration-default'` for TEE off [87]. Apart from this configuration switch, the rest of the software stack (Linux kernel, QEMU, containerd, etc.) remained identical, enabling a controlled comparison of encryption, attestation, and enclave initialization overhead.

6.2 Model Selection

We selected a BERT encoder-only model containing roughly 110million parameters [73]. Despite its considerable size, it remained manageable for CPU-based training while still being representative of modern transformer-based architectures. Specifically, we used a 12-layer BERT variant with a hidden size of 768, taking advantage of PyTorch and Hugging Face Transformers libraries for straightforward loading and fine-tuning [73], [88], [89].

For our dataset, we turned to the GLUE benchmark's MRPC (Microsoft Research Paraphrase Corpus). MRPC consists of sentence pairs manually annotated as to whether they convey the same meaning (paraphrase) or not [91]. We tokenized each sentence pair with a BERT tokenizer at a maximum sequence length of 128 tokens, then stored these tokenized samples on disk [73], [92]. This setup allowed a seamless, immediate load of training and inference batches without excessive preprocessing overhead. The approach and code were adapted entirely from open-source libraries (e.g., Hugging Face's datasets, transformers, and PyTorch) [88], [89].

In practice, BERT uses self-attention layers to capture contextual relations between tokens, effectively handling tasks such as paraphrase detection. For instance, if the two sentences "The weather is nice today." and "It's a beautiful day outside." are fed into the model, it will label them as semantically equivalent—i.e., true for paraphrase.

6.3 Running Training

In training, we varied the batch size among 8, 16, 24, and 32 examples and examined runs with either 1, 3 or 6 epochs. Each epoch was further segmented into

$$\left\lceil \frac{N}{B} \right\rceil$$

iterations [36], [69], [84], [85], with each iteration performing forward and backward passes on one mini-batch, as already explained in Chapter 4. We tracked metrics such as time to first training step(TTFTS), tokens per second, per-iteration latency, overall training time, and model convergence. By performing these runs in both TEE and non-TEE modes, we isolated the additional latency and throughput impact from confidential computation features [93].

Batch size	training time TEE on	training time TEE off	TPS TEE on	TPS TEE off
8	4746.15s	4117.57s	296.77	342.07
16	4529.78s	3906.92s	310.95	360.52
24	4462.21s	3850.10s	315.65	365.84
32	4441.98s	3833.60s	317.09	367.41

Table 6.1: Training time and throughput (TPS) for different batch sizes over 3 epochs comparing TEE on and TEE off.

6.4 Training results

Time To First Training Step(TTFTS): Across all batch sizes (8, 16, 24, 32), the TEE-on configuration consistently increases TTFTS by roughly 15–20% (17.71% on average). For instance, at a batch size of 8, TTFTS is 4738ms with TEE on vs. 4012ms without TEE—an overhead of about 17%. This additional delay comes from initializing the confidential computing environment. Enabling SEV-SNP means all memory pages used by the virtual machine must be encrypted and decrypted transparently. even though the encryption process benefits from specialized hardware features (which help make encryption and decryption faster), there is still a setup cost for establishing the secure pages prior to the first training step. Before execution, the VM may also need to perform attestation, which proves to a remote or local authority that the environment is secure and unmodified. This handshake can add extra latency to the startup phase. SEV-SNP requires additional coordination between the hypervisor (QEMU) and the AMD Secure Processor. This adds overhead before any actual workload can begin. All these factors compound at the very beginning, so TTFTS is visibly higher in TEE mode. Once the system has been initialized most of the extra delay observed during training or inference arises from the continuous processes—like encrypting and decrypting data in memory and the associated system interactions—rather than from that initial setup [36], [37], [41].

Total Training Time: Training overhead averages in the 15% range when comparing TEE on vs. TEE off, regardless of the number of epochs (1, 3, or 6). For example, with one epoch and a batch size of 8, total training time is 1589s (TEE on) vs. 1382s (TEE off), about a 15% difference. Even at larger batch sizes (e.g., 32), the overhead stabilizes around 13–15%. As epochs increase (from 1 to 6), the absolute training time naturally grows, but the percentage overhead due to TEE remains relatively consistent. During each forward/backward pass, the model's parameters and intermediate activations are read from and written to encrypted memory. Although modern processors accelerate these operations, the extra encryption still slows memory throughput slightly. Running inside a TEE can introduce extra context-switch overhead between the guest VM and

the hypervisor, especially if system calls or I/O operations are frequent [62]. In our exepriment, the tokenized training and validation datasets are loaded at the beginning using. This operation is performed only once and its cost is amortized over the entire training run. Once the datasets are loaded, the DataLoader iterates over the in-memory data. There are no repeated, heavy disk reads within the training loop—each batch is prepared in memory and then transferred to the device [73], [92]. This minimizes any ongoing I/O delays. Also, when the environment is set up, each additional epoch faces roughly the same level of overhead, leading to a steady percentage difference across 1, 3, or 6 epochs [26], [36], [37], [41].

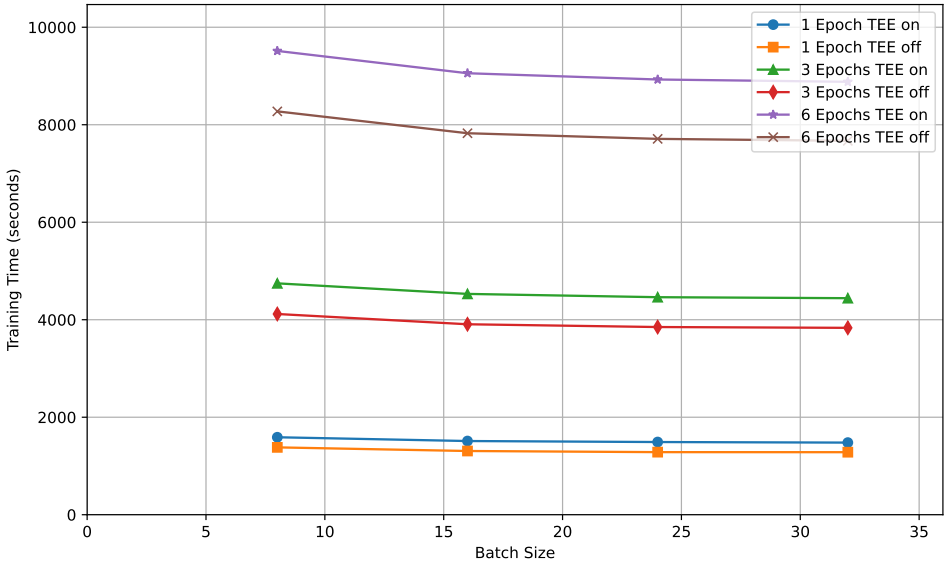

Figure 6.1: Training Time Epochs Comparison.

Tokens per Second(TPS): TEE-enabled runs achieve 10–15% lower TPS (i.e., throughput) than TEE-disabled runs. The gap is obvious at all epoch counts and batch sizes. For a single epoch with a batch size of 32, TPS reaches 317 tokens/s in TEE vs. 366 tokens/s without TEE(13% drop). Throughout each training step, repeatedly reading/writing large parameter tensors from encrypted memory slightly delays raw data bandwidth. Under SEV-SNP, certain hardware optimizations (caching, prefetching) may be slightly less effective when every memory page is encrypted, resulting in lower sustained throughput. BERT training often moves large amounts of data between CPU and memory. Since memory encryption is a dominant factor here, we see an overall 10–15% throughput loss [26], [36], [37], [41].

Per-iterarion Latency: Our training logs also showed that TEE-on runs often record slightly longer step times per mini-batch compared to TEE-off. For example, with batch size 8, TEE-on step times are frequently around 3.45–3.50s, whereas TEE-off is closer to 3.00s—about a 15% overhead. Similarly, for batch size 16, the TEE-on step times of roughly 6.55–6.65s exceed the TEE-off baseline of around 5.70s by a similar margin. This direct measurement of per-batch iteration time confirms that encryption and other TEE-related processes introduce a steady, ongoing overhead on each forward/backward pass, aligning with the 10–15% throughput gap observed in our other metrics [47].

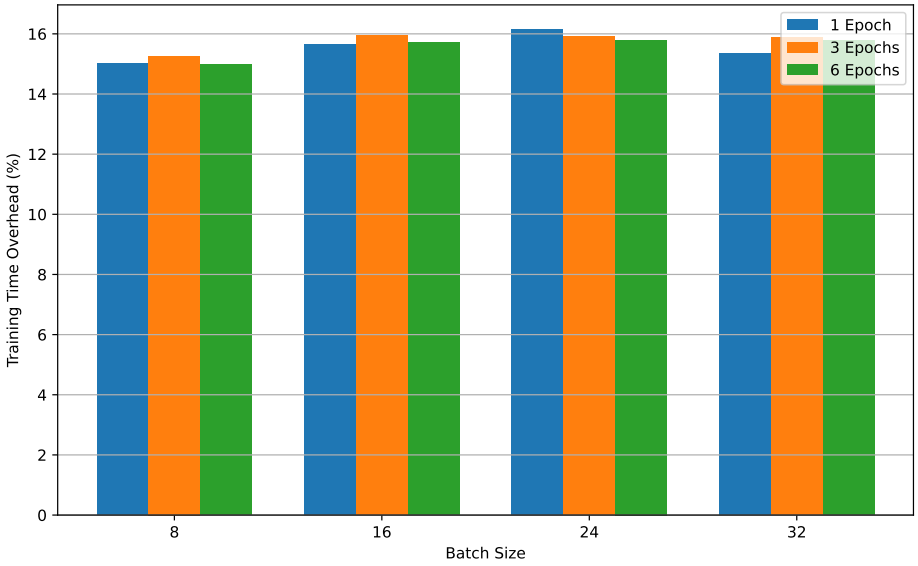

Figure 6.2: Training Time Performance Penalty

6.5 Running Inference

During inference, we maintained the same batch size range (8, 16, 24, 32) to investigate overhead under different query-concurrency conditions. We measured Time to First Token (TTFT), Inter-Token Latency (ITL), Tokens per Second (TPS), and Queries per Second (QPS), providing a detailed view of response times and throughput. Comparing these metrics between TEE-enabled and TEE-disabled runs enabled us to quantify the performance trade-offs of secure inference [93].

Table 6.2: Inference with 3 Epochs

Batch size	TTFT		ITL		TPS		QPS		Latency	
	TEE on	TEE off	TEE on	TEE off	TEE on	TEE off	TEE on	TEE off	TEE on	TEE off
8	232.94 ms	184.77 ms	0.01 ms	0.01 ms	469.22 tok/s	592.04 tok/s	30.55 q/s	38.54 q/s	233.82 ms	185.32 ms
16	293.15 ms	262.08 ms	0.01 ms	0.01 ms	653.58 tok/s	731.08 tok/s	42.55 q/s	47.60 q/s	293.76 ms	262.62 ms
24	427.77 ms	371.65 ms	0.01 ms	0.01 ms	597.53 tok/s	687.78 tok/s	38.90 q/s	44.78 q/s	428.43 ms	372.21 ms
32	571.88 ms	487.85 ms	0.00 ms	0.00 ms	670.68 tok/s	786.15 tok/s	43.66 q/s	51.18 q/s	572.55 ms	488.46 ms

6.6 Inference Results

Time to First Token (TTFT): The TEE-on configuration consistently increases TTFT by 15–30% depending on batch size. Smaller batches can exhibit a slightly higher percentage overhead, because the absolute cost of the TEE setup is amortized over fewer tokens. For example, at batch size 8, TTFT can jump from 188ms (TEE off) to 274ms (TEE on), roughly a 46% difference in that specific case. However, for larger batches (e.g., 32), the gap narrows to around 15–20%. Just as in TTFTS for training, the environment must decrypt and load all necessary model weights into secure memory [47]. For inference, that means there's a short but noticeable overhead before generating the first token. Smaller batches have fewer tokens to distribute that overhead across. The overhead looks larger as a percentage with batch size 8, but for batch size 32, the same overhead gets "spread out," causing a lower relative impact [62].

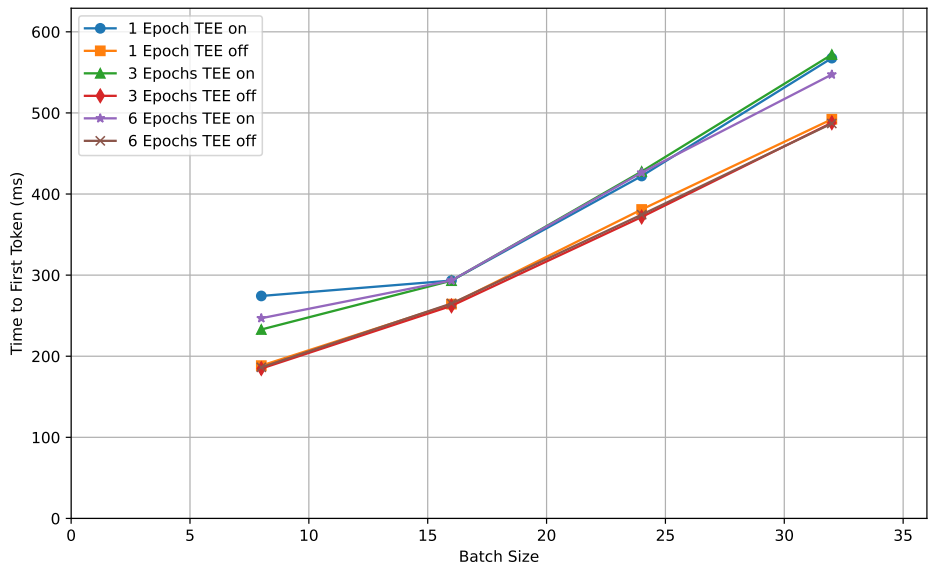

Figure 6.3: TTFT Epochs Comparison.

Tokens per Second (TPS) and Queries per Second (QPS): As with training throughput, inference throughput (TPS/QPS) with TEE shows about a 10–15% decrease across various batch sizes. For batch size 32, TPS is 675–700tokens/s in TEE vs. 780tokens/s without TEE, translating to roughly 11–13% overhead in steady-state generation speed. QPS follows a similar pattern, indicating that when running many inference queries in parallel, TEE-on throughput drops by around 10–15% compared to TEE-off. Inference repeatedly runs forward passes on the model—the process in which input data is fed through the neural network (layer by layer) to compute the output predictions [62], [73]. Each pass deals with large embeddings and attention matrices that must be encrypted/decrypted. Since BERT is relatively large and we're running on CPUs, memory bandwidth—especially encrypted bandwidth—becomes a bottleneck. Even if the queries were parallelized, each thread or process would still operate on encrypted data, so the total overhead would scale with the number of concurrent requests [73].

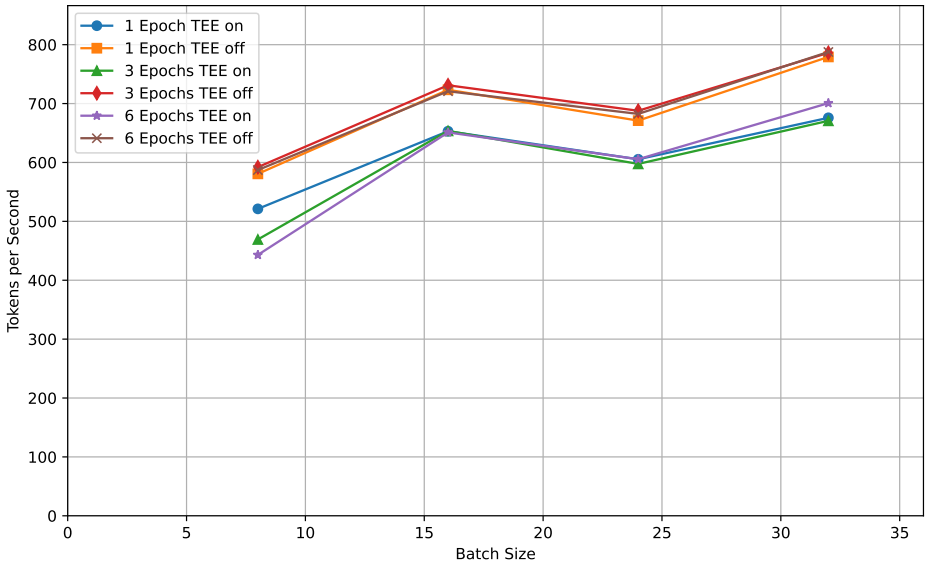

Figure 6.4: TPS Epochs Comparison.

Inter-Token Latency (ITL): The measured inter-token latency shows only minor differences—often just hundredths of a millisecond. Once the first token is generated, much of the model context and memory pages are "already active" in encrypted memory. The additional encryption overhead from token to token is therefore incremental. If the encrypted pages remain in cache, the penalty for reading/writing them each time is lower, reducing the visible ITL. Our encoder-only BERT model processes all tokens at

once, so after the initial setup, the extra encryption cost per token is minimal and barely affects ITL [62], [73].

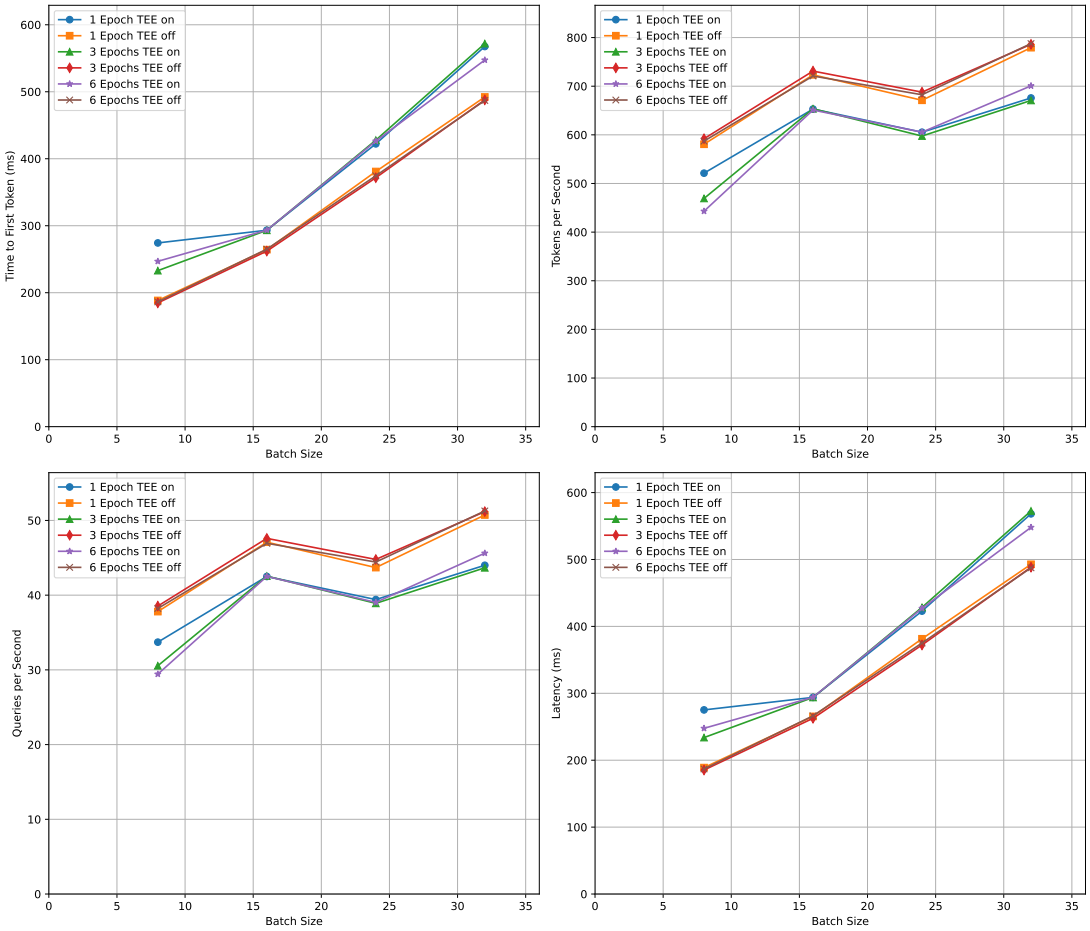

Figure 6.5: Epochs Comparison Summary

6.7 Discussion

Interestingly, the TEE-based run at batch size 24 shows slightly lower throughput than batch size 16. This non-monotonic behavior can come from how encryption overhead, memory alignment, and kernel utilization do not always scale smoothly with batch size. Certain intermediate sizes may suffer from extra setup or alignment costs that outweigh the benefits of larger batches in a TEE environment [62].

Most prior work on TEE-based machine learning either leverages Intel SGX-like enclaves (with strict memory limits) or GPU-based TEEs such as NVIDIA H100/H200 with Intel TDX or AMD SEV SNP for accelerated deep learning [26], [62], [63]. Our experiment

is distinct in that it exclusively uses AMD SEV-SNP on CPUs, without GPU offloading [26]. Consequently, the overhead patterns we observe—around 10–16% for training and inference—differ from the often higher or more complex overheads reported when GPU data transfers, multi-node communication, or very limited enclave memory come into play. From the related studies, we can identify three main bottleneck categories for TEE-based machine learning:

1. **Limited Secure Memory Encryption Overheads:** Many Intel SGX-based works [47], [52] report severe slowdowns (up to 26×) once enclaves must page data in and out of their limited protected region. Our setup bypasses this bottleneck by leveraging SEV-SNP at the VM level, backed by 768GB of RAM. Hence, we do not observe drastic paging or memory-thrashing overhead. Still, encryption of all memory is the fundamental driver of our measured 10–15% overhead, aligning with other CPU TEE findings [37], [41]. Unlike pipeline-optimized GPU TEEs [61], [63], we do not mask encryption costs with asynchronous GPU computation. Our overhead thus emerges in each forward/backward pass, as large parameter tensors are read and written through encrypted memory.

2. **I/O and Communication Bottlenecks (CPU–GPU, Network, Storage):** Studies focusing on GPU TEEs [26], [62] show that encrypting high-bandwidth PCIe traffic can impose overheads of 7–85%, depending on the volume and frequency of transfers. Similarly, multi-node or federated setups must handle repeated encryption of gradients, parameters, and remote attestation steps [29], [34], [39]. In contrast, our single-VM CPU-only workload involves minimal external I/O beyond initial data loading, so we do not see major communication overhead. After the tokenized MRPC dataset is loaded from disk, each batch is prepared and processed in memory, limiting I/O-bound slowdowns. This explains why overhead remains consistently around 10–15%, rather than spiking higher under heavy data transfers.

3. **Scalability Challenges for Multi-Threaded or Multi-Tenant Environments:** Many TEE studies emphasize overhead from context switches, enclave transitions (ecalls/ocalls), or multi-tenant resource contention [33], [37], [39]. For instance, Intel TDX VMs can encounter frequent VM exits, while AMD SEV-SNP uses a Reverse Map Table (RMP) that must validate every page fault [62], [63], [83]. In our experiments, we run a single-tenant workload in a dedicated SEV-SNP VM. We neither stress the system with multiple enclaves nor saturate it with heavy multi-thread concurrency. Consequently, the overhead from TEE-specific context switching is visible but not amplified by multi-tenant competition.

Overall, our analysis shows that CPU-based SEV-SNP overhead is primarily driven by memory encryption, with a stable 10–16% performance penalty for training and inference in a single-VM, CPU-only TEE environment. This is considerably lower than the slowdowns seen in smaller enclaves (such as Intel SGX). Our work, therefore, fills a gap by empirically measuring AMD SEV-SNP overhead without GPU offloading, confirming that modern CPU TEEs can support small-scale transformer training with manageable slowdowns when enough memory is available. It further confirms that memory encryption is the main overhead source—as noted in previous findings [36], [47], [52]—while I/O and multi-tenant overheads found in other studies [62], [63], [83] are not major factors in our CPU-centric scenario.

Chapter 7

Conclusion

This chapter briefly indicates where the research questions posed in Chapter 1 are addressed in this thesis, followed by concise remarks on potential future research directions.

7.1 Answers to Research Questions

RQ1: ("Which phases of the AI model lifecycle are most vulnerable, and how effectively can TEEs protect each?") Answered in **Chapter 3**, where we identify threats at each stage of the AI lifecycle and discuss how TEEs either mitigate or fail to mitigate these threats.

RQ2: ("How do TEE security guarantees impact the complexity of implementing and maintaining AI model deployments?") Covered in **Chapter 3** and **Chapter 6**, with a focus on how memory encryption, enclave initialization, and potential changes in infrastructure increase operational complexity.

RQ3: ("Which attacks remain viable against TEEs even with current security mechanisms?") Explored in **Chapter 3**, which clarifies various software-, hardware-, and algorithmic-level attacks that TEEs cannot fully block, such as adversarial examples or malicious data poisoning.

RQ4: ("What methodology can be used to evaluate security and performance of TEE-enabled AI workflows?") Outlined in **Chapter 3** (analysis), **Chapter 4** (experiment design) and detailed through the **Chapter 5** (implementation), where metrics (latency, throughput, overhead) are defined and measured under TEE vs. non-TEE modes.

RQ5: ("What performance-security trade-offs emerge in TEE-protected LLM deployments, and how can they be optimized?") Quantified in **Chapter 6**, which presents empirical results showing overhead (10–16% in our setup) and the factors (memory encryption, batch size, etc.) that can be tuned for better performance-security balance.

7.2 Future Work

Extending TEE protection to additional accelerators (e.g., NPUs, FPGAs, TPUs) and refining side-channel defenses remain key research goals to further secure large-scale Al deployments. Equally important is advancing adaptive encryption strategies that reduce overhead in multi-tenant or distributed systems, ensuring robust data protection without sacrificing efficiency.

Bibliography

[1] E. Brynjolfsson, D. Li, and L. Raymond, *Generative ai at work*, 2023. DOI: 10.48550/arXiv.2304.11771. arXiv: 2304.11771 [econ.GN]. [Online]. Available: https://doi.org/10.48550/arXiv.2304.11771.

[2] T. Eloundou, S. Manning, P. Mishkin, and D. Rock, *Gpts are gpts: An early look at the labor market impact potential of large language models*, 2023. DOI: 10.48550/arXiv.2303.10130. arXiv: 2303.10130 [econ.GN]. [Online]. Available: https://doi.org/10.48550/arXiv.2303.10130.

[3] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, "Federated learning: Challenges, methods, and future directions", *arXiv preprint arXiv:1908.07873*, 2019. DOI: 10.48550/arXiv.1908.07873. [Online]. Available: https://doi.org/10.48550/arXiv.1908.07873.

[4] K. Pan, Y.-S. Ong, M. Gong, H. Li, A. Qin, and Y. Gao, "Differential privacy in deep learning: A literature survey", *Neurocomputing*, 2024. DOI: 10.1016/j.neucom.2024.127663. [Online]. Available: https://doi.org/10.1016/j.neucom.2024.127663.

[5] L. B. Pulido-Gaytan, A. Tchernykh, J. M. Cortés-Mendoza, and G. Radchenko, "A survey on privacy-preserving machine learning with fully homomorphic encryption", in *Latin America High Performance Computing Conference 2020*, 2020. DOI: 10.1007/978-3-030-68035-0_9. [Online]. Available: https://doi.org/10.1007/978-3-030-68035-0_9.

[6] M. Turtiainen, "Trusted execution environments in protecting machine learning models", Master of Science Thesis, M.S. thesis, University of Turku, Department of Computing, Computer Science, 2023.

[7] K. Zhao, L. Li, K. Ding, N. Z. Gong, Y. Zhao, and Y. Dong, "A survey of model extraction attacks and defenses in distributed computing environments", *arXiv preprint arXiv:2502.16065*, 2025. DOI: 10.48550/arXiv.2502.16065. [Online]. Available: https://doi.org/10.48550/arXiv.2502.16065.

[8] E. Rescorla, *The transport layer security (tls) protocol version 1.3*, RFC 8446, Obsoletes RFCs 5077, 5246, and 6961; Updates RFCs 5705 and 6066, 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8446.

[9] J. Ménétrey, C. Göttel, A. Khurshid, *et al.*, "Attestation mechanisms for trusted execution environments demystified", in *Proceedings of the IEEE/ACM International Symposium on Trusted Computing*, University of Neuchâtel, Switzerland; RISE Research Institutes of Sweden, 2020.

[10] NVIDIA, *Nvidia h100 tensor core gpu architecture: Exceptional performance, scalability, and security for the data center*, NVIDIA Whitepaper, 2023. [Online]. Available: https://www.nvidia.com/en-us/data-center/h100/.

[11] V. Costan and S. Devadas, "Intel sgx explained", IACR Cryptology ePrint Archive, Tech. Rep. 2016/86, 2016. [Online]. Available: https://eprint.iacr.org/2016/86.pdf.

[12] P.-C. Cheng, W. Ozga, E. Valdez, *et al.*, *Intel tdx demystified: A top-down approach*, 2023. DOI: 10.48550/arXiv.2303.15540. arXiv: 2303.15540 [cs.CR]. [Online]. Available: https://doi.org/10.48550/arXiv.2303.15540.

[13] AMD, *Amd sev-snp: Strengthening vm isolation with integrity protection and more*, AMD Tech Docs, 2020. [Online]. Available: https://www.amd.com/en/technologies/secure-encrypted-virtualization.

[14] V. Costan and S. Devadas, *Intel sgx explained*, IACR Cryptology ePrint Archive, 2016(86), 2016.

[15] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: What it is, and what it is not", in *2015 IEEE Trustcom/BigDataSE/ISPA*, 2015, pp. 57–64.

[16] NVIDIA, *Confidential computing on h100 gpus for secure and trustworthy ai*, NVIDIA Developer Blog, 2023.

[17] G. Chen, G. Lin, and B. Zang, "T-sgx: Eradicating controlled-channel attacks against enclave programs", in *NDSS 2017*, 2017.

[18] G. Arfaoui, W. Matallah, and S. Bettayeb, *Confidentiality in trusted execution environments: A survey on secure computing with tees*, ArXiv, 2022. [Online]. Available: https://arxiv.org/pdf/2206.03780.pdf.

[19] P. Pires, M. Pasin, A. Shoker, and P. Esteves-Veríssimo, "Remote attestation in trusted execution environments: Taxonomy, challenges, and future directions", *IEEE Security & Privacy*, 2021, Available at: https://www.researchgate.net/publication/363106383_A_Comprehensive_Analysis_of_Trusted_Execution_Environments.

[20] AMD, *Amd sev-snp: Strengthening vm isolation with integrity protection and more*, AMD Tech Docs, 2020.

[21] M. Werner, "Analysis of amd sev-snp: Features and security implications", in *IEEE Trustcom*, 2021.

[22] NVIDIA, *Confidential computing on h100 gpus for secure and trustworthy ai*, NVIDIA Developer Blog, 2023.

[23] Intel, *Intel tdx: Architectural overview*, Intel Developer Zone, 2023.

[24] ARM, "Arm security technology: Building a secure system using trustzone technology", ARM, White Paper, 2009.

[25] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey", *ACM Computing Surveys*, vol. 51, no. 6, 2019.

[26] A. Mohan, M. Ye, H. Franke, M. Srivatsa, Z. Liu, and N. M. Gonzalez, "Securing ai inference in the cloud: Is cpu-gpu confidential computing ready?", in *Proceedings of the 2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*, 2024, pp. 164–173. DOI: 10.1109/CLOUD62652.2024.00028.

[27] *Health insurance portability and accountability act*, https://en.wikipedia.org/wiki/Health_Insurance_Portability_and_Accountability_Act, Wikipedia.

[28] T. Hunt, C. Song, R. Shokri, V. Shmatikov, and E. Witchel, *Chiron: Privacy-preserving machine learning as a service*, arXiv preprint arXiv:1803.05961, 2018.

[29] A. P. Kalapaaking, I. Khalil, M. S. Rahman, M. Atiquzzaman, X. Yi, and M. Almashor, "Blockchain-based federated learning with secure aggregation in trusted execution environment for internet-of-things", *IEEE Transactions on Industrial Informatics*, vol. 19, no. 2, pp. 1703–1713, 2023. DOI: 10.1109/TII.2022.3170348.

[30] Z. Gu, H. Huang, J. Zhang, *et al.*, *Confidential inference via ternary model partitioning*, arXiv preprint arXiv:1807.00969, 2020.

[31] T. Xu, K. Zhu, A. Andrzejak, and L. Zhang, "Distributed learning in trusted execution environment: A case study of federated learning in SGX", in *Proceedings of the 7th IEEE International Conference on Network Intelligence and Digital Content (IC-NIDC)*, 2021, pp. 450–454.

[32] K. D. Duy, T. Noh, S. Huh, and H. Lee, "Confidential machine learning computation in untrusted environments: A systems security perspective", *IEEE Access*, vol. 9, pp. 168 656–168 677, 2021. DOI: 10.1109/ACCESS.2021.3136889.

[33] R. Kumar, A. A. Khan, S. Zhang, *et al.*, *Blockchain-federated-learning and deep learning models for covid-19 detection using ct imaging*, arXiv preprint, arXiv link: https://arxiv.org/search/eess?searchtype=author&query=Khan,+A+A, n.d.

[34] S. Queyrut, V. Schiavoni, and P. Felber, *Mitigating adversarial attacks in federated learning with trusted execution environments*, arXiv preprint, n.d.

[35] J. Lee, Y. Wang, R. Rajat, and M. Annavaram, *Characterization of gpu tee overheads in distributed data parallel ml training*, arXiv preprint, n.d.

[36] F. Mo, H. Haddadi, K. Katevas, E. Marin, D. Perino, and N. Kourtellis, *PPFL: Privacy-preserving federated learning with trusted execution environments*, arXiv preprint, n.d.

[37] F. Mo, Z. Tarkhani, and H. Haddadi, *Machine learning with confidential computing: A systematization of knowledge*, arXiv preprint, n.d.

[38] M. A. Ağca, S. Faye, and D. Khadraoui, "Trusted distributed artificial intelligence (tdai)", *IEEE Access*, vol. 11, pp. 113 307–113 323, 2023. DOI: 10.1109/ACCESS.2023.3322568.

[39] J. Zhu *et al.*, "Enabling rack-scale confidential computing using heterogeneous trusted execution environment", in *2020 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2020.

[40] A. Gangal, M. Ye, and S. Wei, "HybridTEE: Secure mobile DNN execution using hybrid trusted execution environment", in *2020 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, Kolkata, India, 2020, pp. 1–6. DOI: 10.1109/AsianHOST51057.2020.9358260.

[41] F. Mo, A. S. Shamsabadi, K. Katevas, A. Cavallaro, and H. Haddadi, *Towards characterizing and limiting information exposure in DNN layers*, arXiv preprint, n.d.

[42] Q. Li, Z. Shen, Z. Qin, *et al.*, "TransLinkGuard: Safeguarding transformer models against model stealing in edge deployment", in *Proceedings of the 32nd ACM International Conference on Multimedia (MM '24)*, 2024, pp. 3479–3488. DOI: 10.1145/3664647.3680786.

[43] M. F. Babar and M. Hasan, "Trusted deep neural execution—a survey", *IEEE Access*, vol. 11, pp. 45 736–45 748, 2023. DOI: 10.1109/ACCESS.2023.3274190.

[44] F. Mo, A. S. Shamsabadi, K. Katevas, *et al.*, "DarkneTZ: Towards model privacy at the edge using trusted execution environments", in *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services (MobiSys '20)*, 2020, pp. 161–174. DOI: 10.1145/3386901.3388946.

[45] F. Tramèr and D. Boneh, *Slalom: Fast, verifiable and private execution of neural networks in trusted hardware*, arXiv preprint, n.d.

[46] S. Volos, K. Vaswani, and R. Bruno, "Graviton: Trusted execution environments on GPUs", in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*, USENIX Association, 2018, pp. 681–696.

[47] O. Ohrimenko, F. Schuster, C. Fournet, *et al.*, "Oblivious multi-party machine learning on trusted processors", in *Proceedings of the 25th USENIX Conference on Security Symposium (SEC'16)*, USENIX Association, 2016, pp. 619–636.

[48] K. Kim, C. H. Kim, J. Rhee, *et al.*, "Vessels: Efficient and scalable deep learning prediction on trusted processors", in *Proceedings of the 11th ACM Symposium*

on Cloud Computing (SoCC '20), 2020, pp. 462–476. DOI: 10.1145/3419111.3421282.

[49] Y. Chen, F. Luo, T. Li, T. Xiang, Z. Liu, and J. Li, "A training-integrity privacy-preserving federated learning scheme with trusted execution environment", *Information Sciences*, vol. 522, pp. 69–79, 2020. DOI: 10.1016/j.ins.2020.02.037.

[50] L. Zhang, B. Duan, J. Li, Z. Ma, and X. Cao, "A tee-based federated privacy protection method: Proposal and implementation", *Applied Sciences*, vol. 14, no. 8, p. 3533, 2024. DOI: 10.3390/app14083533.

[51] N. Hynes, R. Cheng, and D. Song, *Efficient deep learning on multi-source private data*, arXiv preprint arXiv:1807.06689, 2018.

[52] J.-B. Truong, W. Gallagher, T. Guo, and R. J. Walls, "Memory-efficient deep learning inference in trusted execution environments", in *Proceedings of the 2021 IEEE International Conference on Cloud Engineering (IC2E)*, 2021, pp. 161–168. DOI: 10.1109/IC2E52221.2021.00031.

[53] A. Mondal, Y. More, R. H. Rooparaghunath, and D. Gupta, "Poster: FLATEE: Federated learning across trusted execution environments", in *Proceedings of the 2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2021, pp. 707–709. DOI: 10.1109/EuroSP51992.2021.00054.

[54] I. Anati, S. Gueron, S. P. Johnson, and V. Scarlata, "Innovative technology for cpu based attestation and sealing", in *HASP '13*, 2013.

[55] J.-B. Truong, W. Gallagher, T. Guo, and R. J. Walls, *Memory-efficient deep learning inference in trusted execution environments*, arXiv:2104.15109, 2021. [Online]. Available: https://arxiv.org/abs/2104.15109.

[56] Intel, *3rd gen intel xeon scalable processors specifications*, Intel ARK Database, 2022.

[57] A. Mohan, M. Ye, H. Franke, M. Srivatsa, Z. Liu, and N. Gonzalez, "Securing ai inference in the cloud: Is cpu-gpu confidential computing ready?", in *IEEE International Conference on Cloud Computing (CLOUD 2024)*, 2024.

[58] R. Dannenberg, F. Kerschbaum, and S. Rane, "Gpu-based secure computing: A tee approach", in *Proceedings of the 15th ACM Workshop on Scalable Trusted Computing*, 2021, pp. 3–14.

[59] ARM, *ARM Security Technology: Building a Secure System using TrustZone® Technology*, White paper, https://documentation-service.arm.com/static/5f212796500e883ab8e74531, n.d.

[60] R. Kumar *et al.*, "Blockchain-federated-learninganddeeplearningmodels for covid-19 detection using ct imaging", *IEEE Sensors Journal*, vol. 21, no. 14, pp. 16 301–16 314, Jul. 2021.

[61] W. Ren, W. Kozlowski, S. Koteshwara, M. Ye, H. Franke, and D. Chen, "AccShield: A new trusted execution environment with machine-learning accelerators", in *Proceedings of the 60th ACM/IEEE Design Automation Conference (DAC)*, 2023.

[62] J. Zhu, H. Yin, P. Deng, A. Almeida, and S. Zhou, *Confidential computing on nvidia hopper gpus: A performance benchmark study*, arXiv preprint arXiv:2409.03992, 2024.

[63] Y. Tan, C. Tan, Z. Mi, and H. Chen, *PipeLLM: Fast and confidential large language model services with speculative pipelined encryption*, arXiv preprint arXiv:2411.03357, to appear in ASPLOS 2025, 2024.

[64] Y. Wang, R. Rajat, J. Lee, T. Tang, and M. Annavaram, *Fastrack: Fast io for secure ml using gpu TEEs*, arXiv preprint arXiv:2410.15240, 2024.

[65] M. Li, Y. Yang, G. Chen, M. Yan, and Y. Zhang, "Sok: Understanding design choices and pitfalls of trusted execution environments", in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security (ASIA CCS '24)*, 2024, pp. 1600–1616. DOI: 10.1145/3634737.3644993.

[66] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey", *ACM Computing Surveys*, vol. 51, no. 6, 2019.

[67] X. Ren, "An enclave-based tee for se-in-soc in risc-v industry", *arXiv preprint arXiv:2208.03631*, 2022, Accessed: 2025-04-12. [Online]. Available: https://arxiv.org/abs/2208.03631.

[68] I. Sarker, "Machine learning: Algorithms, real-world applications and research directions", *SN Computer Science*, vol. 2, 2021. DOI: 10.1007/s42979-021-00592-x.

[69] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, A comprehensive resource covering deep architectures, neural network training, and applications in vision, speech, and language.

[70] G. Edegbe and S. Acheme, "A systematic review of centralized and decentralized machine learning models: Security concerns, defenses and future directions", *NIPES Journal of Science and Technology Research*, vol. 6, pp. 161–175, 2024. DOI: 10.5281/zenodo.14681449.

[71] K. D. Duy, T. Noh, S. Huh, and H. Lee, "Confidential machine learning computation in untrusted environments: A systems security perspective", *IEEE Access*, vol. 9, pp. 168 656–168 677, 2021. DOI: 10.1109/ACCESS.2021.3136889.

[72] F. Mo, Z. Tarkhani, and H. Haddadi, "Machine learning with confidential computing: A systematization of knowledge", *ACM Computing Surveys*, vol. 56, no. 11, Article 281, 2024. DOI: 10.1145/3670007.

[73] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, *BERT: Pre-training of deep bidirectional transformers for language understanding*, arXiv preprint arXiv:1810.04805, 2018. [Online]. Available: https://doi.org/10.48550/arXiv.1810.04805.

[74] Y. Liu, M. Ott, N. Goyal, *et al.*, *RoBERTa: A robustly optimized BERT pretraining approach*, arXiv preprint arXiv:1907.11692, 2019. [Online]. Available: https://doi.org/10.48550/arXiv.1907.11692.

[75] Wikipedia contributors, *Transformer (deep learning architecture)*, Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/wiki/Transformer_(deep_learning_architecture) (accessed March 16, 2025), n.d.

[76] T. B. Brown, B. Mann, N. Ryder, *et al.*, *Language models are few-shot learners*, arXiv preprint arXiv:2005.14165, 2020. [Online]. Available: https://doi.org/10.48550/arXiv.2005.14165.

[77] Amazon Web Services, *Transformers in artificial intelligence*, https://aws.amazon.com/what-is/transformers-in-artificial-intelligence/ (accessed March 16, 2025), n.d.

[78] C. Raffel, N. Shazeer, A. Roberts, *et al.*, *Exploring the limits of transfer learning with a unified text-to-text transformer*, arXiv preprint arXiv:1910.10683, 2019. [Online]. Available: https://doi.org/10.48550/arXiv.1910.10683.

[79] A. T. Vassilev, A. T. Vassilev, A. Oprea, A. Fordyce, and H. Anderson, "Adversarial machine learning: A taxonomy and terminology of attacks and mitigations", National Institute of Standards and Technology, Tech. Rep. NIST AI 100-2e2023, Jan. 2024, Report number: NIST AI 100-2e2023. DOI: 10.6028/NIST.AI.100-2e2023.

[80] S. Banerjee, P. Sahu, M. Luo, A. Vahldiek-Oberwagner, N. J. Yadwadkar, and M. Tiwari, *Sok: A systems perspective on compound ai threats and countermeasures*, 13 pages, 4 figures, 2 tables, 2024. DOI: 10.48550/arXiv.2411.13459. arXiv: 2411.13459 [cs.CR].

[81] G. Dhanuskodi, S. Guha, V. Krishnan, *et al.*, "Creating the first confidential gpus: The team at nvidia brings confidentiality and integrity to user code and data for accelerated computing", *Queue*, vol. 21, no. 4, pp. 68–93, 2023, Published: 07 September 2023. DOI: 10.1145/3623393.3623391.

[82] B. Hitaj, G. Ateniese, and F. Perez-Cruz, "Deep models under the gan: Information leakage from collaborative deep learning", in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, arXiv:1702.07464v3 [cs.CR], 2017.

[83] M. Misono, D. Stavrakakis, N. Santos, and P. Bhatotia, "Confidential vms explained: An empirical analysis of AMD SEV-SNP and Intel TDX", *Proceedings*

of the ACM on Measurement and Analysis of Computing Systems, vol. 8, no. 3, Article 36 (42 pages), 2024. DOI: 10.1145/3700418.

[84] J. Frankle, D. J. Schwab, and A. S. Morcos, *The early phase of neural network training*, Submitted on 24 Feb 2020. ICLR 2020 Camera Ready. Available on OpenReview., 2020. DOI: 10.48550/arXiv.2002.10365. arXiv: 2002.10365 [cs.LG].

[85] B. M. Hussein and S. M. Shareef, "An empirical study on the correlation between early stopping patience and epochs in deep learning", in *ITM Web of Conferences*, License CC BY 4.0, vol. 64, 2024. DOI: 10.1051/itmconf/20246401003.

[86] G. Ramshankar, C.-Y. Yang, and Y.-H. Lu, *Protect privacy of training data in deep neural networks on edge using trusted execution environments*, Preprint; explores the impact of TEE constraints on hyperparameters such as batch size and epochs in deep neural network training, 2023.

[87] K. Containers, *Kata containers: Lightweight virtual machines for enhanced security*, https://katacontainers.io, 2022.

[88] T. Wolf, L. Debut, V. Sanh, *et al.*, *Huggingface's transformers: State-of-the-art natural language processing*, arXiv:1910.03771 [cs.CL], 2019. DOI: 10.48550/arXiv.1910.03771. [Online]. Available: https://arxiv.org/abs/1910.03771.

[89] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library", in *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[90] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning", in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283.

[91] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "Glue: A multi-task benchmark and analysis platform for natural language understanding", *arXiv preprint arXiv:1804.07461*, 2018.

[92] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning", *Nature*, vol. 521, pp. 436–444, 2015. DOI: 10.1038/nature14539.

[93] M. Brajkovska, *Ba brajkovska repository: Readme*, https://github.com/brajkovskam/ba-brajkovska/blob/main/README.md, Accessed: 2023-10-11, 2023.