InferONNX: Practical and Privacy-preserving Machine Learning Inference using Trusted Execution Environments

Konstantina Papafragkaki[1,2] and Giorgos Vasiliadis[1,3]

[1] Institute of Computer Science, Foundation for Research and Technology - Hellas, Heraklion, Greece
[2] Computer Science Department, University of Crete, Heraklion, Greece
[3] Department of Management Science and Technology, Hellenic Mediterranean University, Agios Nikolaos, Greece
{papafrkon,gvasil}@ics.forth.gr

Abstract. Machine learning is increasingly applied in critical domains where sensitive data is involved. When models are deployed on untrusted devices, this raises significant privacy concerns for both model providers and end-users. Trusted Execution Environments (TEEs), which offer hardware-based protection for data during processing, can mitigate these concerns. However, their limited memory resources pose challenges for deploying traditional machine learning frameworks.
In this paper, we propose InferONNX, a lightweight machine learning inference service designed to run within Intel SGX. It embeds a high-level, portable, and framework-agnostic model format into the enclave, enabling easy execution of a wide range of machine learning and deep learning models. To address the memory limitations of Intel SGX, InferONNX employs two key strategies: a compact runtime with a small memory footprint, and model partitioning to reduce the memory required during inference. By executing model partitions instead of the full model, the system achieves 1.5× to 4× faster inference depending on the model size.

Keywords: Confidential computing · Trusted execution · Intel SGX · Machine learning · Inference

1 Introduction

Machine learning and deep learning models have made significant progress and are being used extensively across a wide range of application domains and business sectors, such as image classification [43], speech recognition [31], natural language processing [22], and healthcare diagnostics [37,42]. In terms of privacy though, machine learning and deep learning inference present challenges for both model providers and end-users [29]. Execution on untrusted end-user devices requires maintaining the confidentiality of proprietary models. In principle, a model can be a considerable investment and a valuable asset for a company or organization. Such models can be leaked in untrusted host scenarios, as end-users have full access to the hardware and the software installed on their devices. On the contrary, the execution on the model provider premises, via cloud-enabled services, raises serious privacy concerns for the end-users, as they are required in many cases to outsource private or sensitive data (such as images, voices, and text). Such data can be exposed to external threats due to vulnerabilities in these environments.

Overall, it is imperative to provide mechanisms that satisfy the confidentiality of both users' data and models, so as to enable privacy-preserving inference on untrusted hosts. A practical approach to tackle these needs is to utilize hardware-enabled trusted execution environments (TEE). These environments can be used to protect data while in use, a concept known as confidential computing. A TEE provided by hardware isolates programs or program fragments, and their data from potentially malicious operating systems, hypervisors, or any other privileged process. As a result, TEEs are becoming increasingly widespread in the machine learning domain [14, 23, 26], and provide a compelling paradigm for machine learning tasks, including inference. However, deploying complex machine learning workloads on TEEs has to address the limited memory resources of TEEs. This is a major constraint for machine learning workloads, as not only the models, but also the frameworks are quite large. For instance, PyTorch v3.9.11 is about 760 MB and TensorFlow v2.13.1 is 1.3 GB. These memory and security constraints not only affect runtime performance, but also introduce overheads during model initialization. As shown in Figure 1, inference tasks pay increased overheads when models are loaded from disk (*cold-start*). The most straightforward way to mitigate cold-start latency is to keep models permanently in memory, a strategy adopted by the majority of previous work [24, 27, 38]. Unfortunately, this is not practical in typical scenarios where the size of the models exceeds the system memory capacity. Therefore, it is crucial to design mechanisms for efficiently loading models from disk.

This paper presents InferONNX, a lightweight machine learning inference service designed to run within Intel SGX. InferONNX's key insight is to embed the runtime environment of a high-level machine learning format, namely ONNX (Open Neural Network Exchange) [6], into Intel SGX. ONNX is chosen due to its high-level semantics, its wide popularity, and its small memory footprint: the resulting inference engine has a size of 46MB, consuming minimal enclave memory and thus leaving more memory resources available for the machine learning and deep learning models running atop InferONNX. To manage the limited memory resources of the machine, we further partition those models and store them on disk rather than keeping them in memory; the partitions are loaded from disk at runtime and executed sequentially.

The main contributions of this paper are:

– We design and implement InferONNX, a lightweight service for Intel SGX, that enables confidential inference using the ONNX language. InferONNX relies on disk-based storage for machine learning models to ensure scalabil-

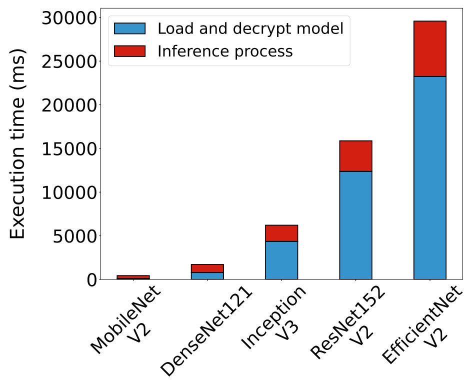


Fig. 1. Execution time breakdown for five popular machine learning models that span different sizes. The blue portion represents the time spent loading and decrypting the models from disk (cold start), while the red portion represents the execution of the inference process that produces the result (warm state).

ity and manage memory constraints effectively through model partitioning (1.5×–4× reduced overhead compared to executing the full model).

– We show that InferONNX can handle machine learning models of varying sizes, from the lightweight SqueezeNet1.0 to the larger EfficientNet V2, with a maximum overhead of 3.65× compared to their unprotected counterparts.

2 Background

This section provides background on Intel SGX and the use of Library Operating Systems (libOSes) to support secure application deployment within Intel SGX enclaves.

2.1 Intel SGX

Intel SGX (Software Guard Extensions) introduces secure enclaves to protect the confidentiality and integrity of sensitive data and applications. These secure enclaves provide isolated regions within the CPU, ensuring that sensitive information is processed in isolation, even if the operating system, hypervisor or other parts of the system are compromised. Code and data within an enclave are stored in a protected region of memory called the Enclave Page Cache (EPC).

Since the EPC has limited capacity, Intel SGX includes a paging mechanism to manage applications requiring more memory. This mechanism encrypts EPC pages and transfers them to an untrusted DRAM buffer, maintaining security during this process. However, these operations involve encryption, decryption, and privileged instructions executed outside the enclave, resulting in performance overhead. Moreover, when EPC pages are reused, the Translation Lookaside Buffer (TLB) is cleared, introducing further delays.

The creation of an enclave begins with the `ECREATE` instruction, which initializes its control structure within the EPC. Memory setup and cryptographic measurements for remote attestation are managed by subsequent instructions like `EADD` and `EINIT`. Once prepared, the enclave code is executed through the `EENTER` instruction, which switches the processor to enclave mode. SGX also supports multi-threading within enclaves, with each thread's context maintained in a Thread Control Structure (TCS). The `EEXIT` instruction is used to terminate enclave execution and return control to the untrusted application.

2.2 Library OSes for Intel SGX

Deploying only part of an application in a Trusted Execution Environment (TEE) often requires manual code partitioning, along with recompilation and relinking of the entire application—even for components that remain outside the TEE. This static and tightly coupled development model limits support for applications that depend on runtime code extensibility. Moreover, additional complexities, such as handling cryptographic operations and enabling end-to-end encryption, further complicate the development process.

Library Operating Systems (libOSes) help address the challenges of deploying applications in TEEs by offering two key advantages: reducing the size of the Trusted Computing Base (TCB) and simplifying the development workflow. A smaller TCB improves overall system security by limiting the amount of trusted code, thereby reducing the potential for vulnerabilities. LibOSes achieve this by including only minimal, essential components—such as a shim C library—within the enclave, while leaving larger system libraries and runtimes outside the trusted boundary. At the same time, libOSes streamline secure application development by enabling the entire application stack—including code, libraries, and system functions—to run inside Intel SGX with minimal or no code changes. They transparently manage interactions with untrusted system components, such as I/O operations, making it easier to adapt existing applications to run securely within a TEE.

3 Design objectives

The widespread adoption of machine learning and deep learning models has led to a significant expansion of software stacks designed to improve efficiency. Typically, frameworks, such as TensorFlow and PyTorch, are used for training and optimizing machine learning models in an iterative process. Once a machine learning model has been trained, it can be deployed for inference. However, inference with these frameworks brings unnecessary overheads mainly due to the bloating size of the target application. To overcome this, applications can manually use optimized operators that utilize carefully-designed assembly instructions [1–3], or directly embed the model's architecture and parameters into the source code. By doing so, the target application is lightweight and efficient, as it does not rely on any external library. The procedure can be performed either manually, by porting the models on hand, or automatically, by using domain-specific deep learning compilers, such as TVM [12]. Both approaches offer quite improved performance, however manual porting requires substantial effort and is time-consuming. Compiler-based approaches, such as TVM, are also scalable in terms of model size and hardware heterogeneity, by producing appropriate executables that are also optimized for the target hardware. However, these benefits come at a cost when executing in an environment where the model is not fully trusted. In such scenarios, the compiled code operates outside the user's direct control, meaning the binary could potentially be tampered with; creating an opportunity for malicious code injection. This, in turn, could allow attackers to steal sensitive data, either directly or indirectly, for example through covert channels; even if the code is isolated within a secure enclave. One way to prevent this, is to confine the untrusted compiled code into a trusted sandbox, or have a trusted inference engine that executes machine learning models on a unified format, such as the ONNX. Using the latter, the inference engine can execute any machine learning model, without requiring any instrumentation or code validation/verification. In addition, embedding an ONNX runtime inside a TEE tackles the challenges of transparency, dynamic extensibility, and runtime safety—by piggy-backing on the characteristics of a high-level open source format for machine learning and deep learning models, such as ONNX.

Our approach offers significant developer economy compared to low-level abstractions, because of the productivity benefits stemming from a high-level widely-adopted format. Applications can leverage the ONNX model ecosystem transparently, without having to develop them from scratch—such as image analysis, object detection, and natural language processing. Finally, since ONNX is a high-level, declarative format for machine learning and deep learning models, its execution is constrained, minimizing the risk of unintended behavior and helping protect data owners from potential leaks by untrusted model providers.

4 Design

This section outlines the design of InferONNX. We describe its client-server architecture, the use of enclaves for secure ONNX model execution, and the partitioning strategy used to manage models that exceed the memory limits of Intel SGX.

4.1 Client-server architecture

We design an inference server that enables an end-to-end secure environment between clients and model providers, so as to protect the sensitive data of clients and the intellectual property of model providers. As shown in Figure 2, the server operates within hardware-based secure enclaves, isolating the execution environment and protecting sensitive data during processing. This data is encrypted and shielded from unauthorized access at the hardware level, ensuring that information within these enclaves remains secure.

To establish secure connections between the server and clients, we use the Transport Layer Security (TLS) protocol, which ensures encrypted communication, protecting the confidentiality and integrity of data exchanged over the network. Along with this, we implement access control by authenticating each client, ensuring that only clients with valid credentials can interact with the server. For clients within Intel SGX environments, Remote Attestation (RA) can be used to verify their authenticity. This process ensures that the client is genuine before any sensitive data is exchanged, establishing trust between the parties, and can also be optimized to reduce latency and enhance performance [10]. In contrast, clients running on conventional CPUs (without Intel SGX support) are assigned a unique private key, enabling the establishment of a mutually authenticated TLS session.

The client-server interaction consists of two primary modes:

- **Model uploading.** In this mode, the server receives models, in ONNX format, securely from model providers. The models are encrypted using the AES-256-GCM mode and stored on disk. Each model is associated with a unique identifier (ID), through a hash table that also contains its cryptographic metadata (encryption key and initialization vector). The clients can use these IDs to submit inference requests on the corresponding models.
- **Inference serving.** In this mode, clients submit inference requests that include the model's ID and the corresponding input data. The server uses the ID to locate the appropriate model, loads it from disk into the enclave, decrypts it, and then initiates the inference process. Inference is performed by the ONNX interpreter described in Section 4.2, and the resulting output is returned to the client.

4.2 ONNX Interpreter enclaves

We deploy an interpreter within secure enclaves to execute machine learning and deep learning models in ONNX format. Unlike compiled execution, which requires pre-compilation of the models, the interpreter dynamically processes any model by reading its structure and executing the respective operations sequentially. This approach offers flexibility in handling a variety of model architectures, making it well-suited for environments where models may vary or need to be updated without recompilation. By executing models in a secure enclave, we ensure that sensitive data, such as inputs and model weights, remain protected throughout the execution process, leveraging the strong isolation capabilities of Intel SGX.

ONNX provides a universal format for machine learning and deep learning models, allowing models trained in different frameworks (such as TensorFlow, PyTorch, and Scikit-Learn) to be represented in a standardized form. ONNX represents models as computational graphs, where nodes correspond to operations (referred to as operators) and edges define the data flow between these operations. During execution, the ONNX interpreter reads the model's graph

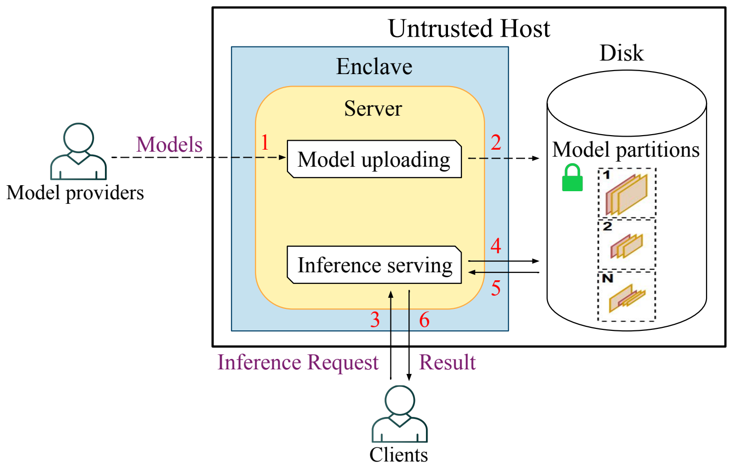


Fig. 2. Overview of InferONNX. The untrusted section (white) represents the untrusted device, while the trusted section (light blue) represents the enclave where the server operates. During *model uploading*, the model providers upload models to the server (step 1), which partitions, encrypts and stores them on disk (step 2). During *inference serving*, the clients submit inference requests to the server (step 3). The server performs inference by sequentially loading the relevant model partitions from disk into the trusted environment (steps 4–5), executing them in order, and then returning the final inference result to the client (step 6).

and iteratively processes each operator, applying the respective computations and data transformations. This sequential execution ensures that the operations are carried out in the correct order while maintaining the integrity of the model's intended functionality. The interpreter handles different types of operators by using pre-implemented functions.

4.3 Model partitioning

As we experimentally verify in Section 6.4, executing entire models within the enclave incurs performance overhead, which we mitigate through model partitioning. Model partitioning divides the model architecture into smaller, sequential components that can be processed more efficiently within the memory constraints of hardware-based enclaves. Model partitioning can be classified into two categories: (i) *intra-operator partitioning*, where the computation and input data of a single operator are split into multiple segments—each processing a subset of the data and passing intermediate results between segments; and (ii) *inter-operator partitioning*, where groups of operators are treated as independent execution units. In this work, we focus on inter-operator partitioning. While intra-operator partitioning could be useful in scenarios where a single operator exceeds the enclave's EPC capacity, it may introduce additional overhead due to the need for intermediate data transfers. We leave the exploration of intra-operator partitioning to future work.

The models are divided into multiple partitions based on the EPC size and the computational cost of individual operators, similar to [17]. To identify memory-intensive (heavy-weight) operators, we profile each model on both Intel SGX and a standard CPU, using execution time as a proxy for memory usage—since memory consumption cannot be reliably inferred from the operator's input size alone. Once these operators are identified, we perform model partitioning. This process involves traversing the computational graph in reverse order — from the last operator to the first — to accommodate both simple and complex topologies. As we iterate through the operators, we accumulate their estimated sizes. If the combined size of the current and previous operators exceeds the EPC capacity, we define a partition from the starting operator to the previous one. The next partition starts at the current operator, and size accumulation resets. If an operator is marked as heavy-weight, it is treated as a standalone partition, and the accumulation restarts from zero.

This partitioning procedure ensures that all partitions meet the system's memory requirements. Once precomputed, model providers upload the partitions to the server, where they are used for inference. During *inference serving*, clients submit requests to run inference using these sub-models. After loading all of them from disk within the enclave, the partitions are executed in a pipeline, with the output of one partition passed as input to the subsequent partition. The result of the final partition corresponds to the inference result of the entire model, which is then returned to the clients.

5 Implementation

This section outlines the implementation of InferONNX, focusing on the base runtime built with Occlum, the TLS-based secure communication layer, and the trusted inference engine that executes models within Intel SGX enclaves.

5.1 Base runtime

To simplify the development of InferONNX, we adopt Occlum [39], a libOS tailored for Intel SGX. Occlum enables secure execution of application processes entirely within the enclave. The Occlum runtime uses a packaged file system, known as the Occlum image, to configure and manage the enclave environment. This image includes the InferONNX binaries, configuration files, and required libraries, totaling approximately 59MB. At runtime, it is used to initialize the enclave and securely execute the InferONNX server alongside the deployed models.

In its current design, InferONNX runs a single Occlum image at a time, avoiding concurrent sessions and multi-client access. This serialized execution model reduces the attack surface and helps mitigate side-channel threats, such as timing and cache attacks, that are known to affect Intel SGX [9, 11, 19, 44].

5.2 Secure communications layer

We integrate the MbedTLS library [16], which provides full support for TLSv1.2 and v1.3 and is optimized for resource-constrained environments. Each client request is handled over a dedicated TLS session, ensuring end-to-end encryption between the client and the InferONNX server. Upon receiving a request, the server authenticates the client and establishes a secure channel before proceeding with either *model uploading* or *inference serving*, depending on the request type. This communication model ensures the security of data in transit, maintaining both confidentiality and integrity throughout the client-server interaction.

5.3 Trusted inference engine

InferONNX performs inference operations, using Tract [32], a Rust-based inference engine that supports various model formats, including ONNX. Tract leverages Rust built-in memory-safety features, and its lightweight design, at approximately 46MB, makes it particularly well-suited for memory-constrained environments, such as Intel SGX.

6 Evaluation

In this section, we evaluate the performance of InferONNX, focusing on the overheads introduced by Intel SGX and the impact of model partitioning.

6.1 Experimental Setup

Platform. We evaluate InferONNX on a single machine with an Intel Core i7-7700 3.60 GHz CPU and 32GB RAM, running Ubuntu 20.04 and supporting Intel SGX v1.0 [13]. In Intel SGX v1.0, each enclave is limited to 128MB, reduced to around 100MB due to metadata and system-reserved memory, with static memory initialization for the enclave's lifetime. Intel SGX v2.0, by contrast, supports scalable memory through dynamic allocation and thread creation, but this flexibility comes with a relaxed threat model, offering full protection against cyber-attacks but only partial protection from physical attacks. While Intel SGX v2.0 could provide better performance and scalability in certain cases, the inference process in our system does not require frequent memory allocations, making Intel SGX v1.0's static memory model sufficient for our needs. Further exploration is needed to assess if an Intel SGX v2.0-based solution would offer meaningful performance gains and whether those gains would justify the security trade-offs.

Workload. We use models from image classification and general-purpose tasks, including SqueezeNet1.0, MobileNet V2, DenseNet121, EfficientNet Lite4, Inception V3, and ResNet101/152 V2. Most of these models were downloaded from the ONNX Model Zoo [4], while the ResNet models were obtained from a GitHub repository [7].

6.2 Memory usage profiling

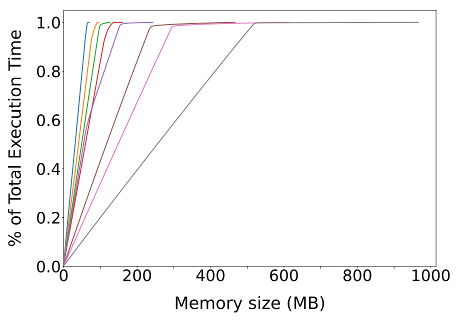
We use the Valgrind Massif tool [5] to profile the memory usage of InferONNX. Massif tracks cumulative memory allocations throughout a process and captures real-time memory snapshots. The peak value among these snapshots reflects the highest memory usage observed during inference for a given model or partition. This peak is used to estimate the memory footprint of each operator during partitioning. In this analysis, we focus on 'heavy-weight' operators—those incurring a performance overhead of at least 12×—since partitioning offers little to no benefit for lighter operators.

Table 1. The sizes of various models (on disk) and the corresponding number of partitions needed for each of them to fit the EPC size.

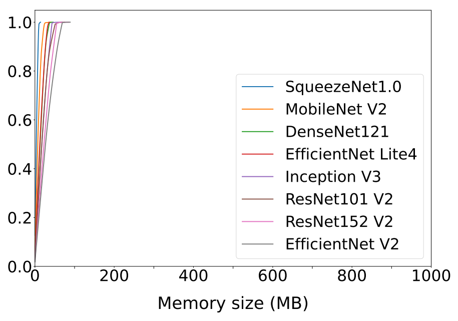
Model	Disk Size (MB)	Partitions (#)
SqueezeNet1.0	4.8	3
MobileNet V2	13.5	6
DenseNet121	31.2	31
EfficientNet Lite4	49.5	36
Inception V3	90.9	23
ResNet101 V2	170	25
ResNet152 V2	230	34
EfficientNet V2	451	77

Table 1 presents details on the disk sizes of various models and the number of partitions required for each. We observe that models with similar disk sizes do not necessarily require the same number of partitions. This is due to the heterogeneous structure of machine learning models, where differences in layer composition, operator memory usage, and execution patterns significantly impact memory demands during inference. Consequently, partitioning decisions depend not only on disk size but also on the internal characteristics of each model.

Figure 3a illustrates the memory requirements of each model, highlighting the percentage of snapshots that exceed the EPC capacity (∼100MB). We observe that smaller models, such as SqueezeNet1.0 and MobileNet V2, consume less than 100MB of memory. However, since the Occlum image occupies around 59MB, only about 41MB of usable memory remains for models. This constraint requires partitioning for these models to fit within the available memory. As the model size increases, the memory demands grow accordingly, with the two largest models, ResNet152 V2 and EfficientNet V2, requiring around 470MB and 530MB of memory at the 97th percentile. To verify the effectiveness of the partitioning technique, we also show the memory requirements of each model's partitions during sequential execution in Figure 3b. We observe that, after partitioning, all models remain under the 100MB range, confirming the technique's efficiency.



(a)



(b)

Fig. 3. Memory requirements of models when running as a whole (a), and when running in partitions (b). The average percentage of memory snapshots is shown, indicating whether they exceed the EPC capacity.

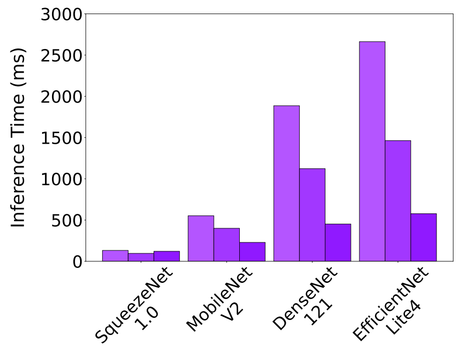
6.3 Baseline performance

In this section, we evaluate the performance of machine learning inference on the CPU without Intel SGX, disk encryption, or TLS connections. The primary metric is inference time, measured in milliseconds (ms), capturing the duration from when a client sends a request to when the server returns the result over an unencrypted, plain connection. This baseline serves as a reference point to quantify and justify the overheads introduced by Intel SGX, secure disk access, and TLS in subsequent experiments.

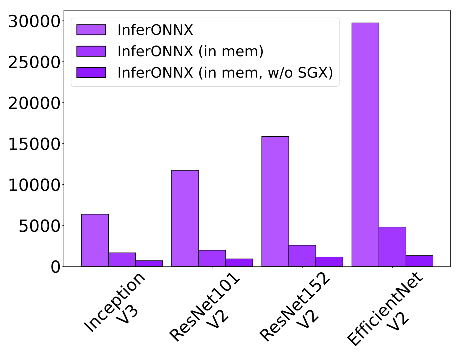
As shown in Table 2, execution is faster when models are stored in memory. However, when models are loaded from disk, execution times increase by a factor of 1.27× to 2×. The smallest model, SqueezeNet1.0, experiences a more pronounced slowdown. A similar trend is observed with MobileNet V2, the second smallest model, while larger models show a more moderate increase in execution time, ranging from 1.27× to 1.4×.

Table 2. Base inference execution time (in milliseconds) on CPU without Intel SGX or TLS, with models either stored in memory or loaded from disk.

Models	Stored in memory	Loaded from disk
SqueezeNet1.0	34	70
MobileNet V2	162	247
DenseNet121	381	512
EfficientNet Lite4	511	647
Inception V3	597	771
ResNet101 V2	839	1093
ResNet152 V2	1073	1374
EfficientNet V2	1257	1754



(a)



(b)

Fig. 4. Performance evaluation when running inference on small (a) and large (b) models, across three configurations: the approach where the full models are loaded from disk and decrypted, the in-memory approach, and the baseline where the execution is on CPU, without Intel SGX.

6.4 Performance of InferONNX during full model execution

In this section, we analyze the overheads introduced by Intel SGX, along with the computational and I/O costs associated with loading and decrypting the full models from disk. To isolate these effects, we run InferONNX without using model partitioning; instead, the full model is loaded from disk on each inference request. This setup is designed to highlight the overhead of stressing the limited EPC memory of Intel SGX during end-to-end inference. For comparison, we evaluate two additional configurations: (i) InferONNX running outside the Intel SGX enclave to quantify SGX-related overheads, and (ii) InferONNX running with models preloaded into memory, eliminating disk access and decryption to expose only the computational cost within the enclave.

Figure 4a and 4b present the performance of InferONNX across small and large models respectively. We define small models as those whose memory requirements are within or only slightly above the EPC limit—such as SqueezeNet1.0, MobileNet V2, DenseNet121, and EfficientNet Lite4. The remaining models exceed this threshold and are classified as large. For small models, the impact of SGX-induced page swapping is minimal, as their memory demands fit within the EPC capacity. In contrast, large models experience more pronounced performance degradation due to frequent page swapping, as illustrated in Figure 4b. The largest model, EfficientNet V2—requiring approximately five times the EPC capacity—suffers the highest overhead, with a slowdown of 3.65×. Other large models incur overheads ranging from 2.15× to 2.38×. These performance penalties are primarily attributed to the overhead of encryption, decryption, and secure data transfers between the enclave and untrusted memory. Despite this, as shown in Table 3, the Instructions Per Cycle (IPC) remains stable across all models, ranging from 2.27 to 2.67. This consistency indicates that CPU efficiency is largely unaffected, and that the main bottleneck stems from SGX-induced page swapping.

Next, we compare the performance of InferONNX when models are preloaded into memory versus when they are loaded from disk. The results highlight the overhead introduced by disk I/O, as loading models from disk and performing decryption during inference leads to additional latency. For small models, this overhead results in a performance penalty of 1.37× to 1.82×. For large models, the overhead is even more significant, with performance degradation ranging from 3.84× to approximately 6.2×. These penalties are primarily due to the time spent on disk access, SGX-induced page swapping, and the decryption process during runtime.

Table 3. Instructions Per Cycle (IPC) values across different models.

Model	IPC
SqueezeNet1.0	2.62
MobileNet V2	2.66
DenseNet121	2.38
EfficientNet Lite4	2.67
Inception V3	2.35
ResNet101 V2	2.46
ResNet152 V2	2.58
EfficientNet V2	2.27

6.5 Performance of InferONNX with model partitioning

We now evaluate the performance of InferONNX with model partitioning. Due to Intel SGX's memory constraints, model partitioning is employed to enhance efficiency by loading large models from disk in smaller, manageable segments. To assess its impact, we compare the execution times of full models with those of their partitioned versions, loaded and executed sequentially.

As shown in Figure 5a, small models, such as SqueezeNet1.0 and MobileNet V2, are not significantly affected by partitioning, as their execution times remain similar to when running the full model. However, as model size increases, the benefits of partitioning become more apparent. DenseNet121, with a size of 31.2MB, shows a 1.46× improvement, while EfficientNet Lite4, at approximately 50MB, achieves a 1.48× improvement. For large models, the impact of partitioning is even more pronounced, as shown in Figure 5b. Partitioning leads to further improvements in execution times, with performance gains becoming more substantial as the model size increases. These improvements range from 2.64× for Inception V3 (90.9MB) to 4.04× for the largest model, EfficientNet V2 (451MB).

Overall, model partitioning can significantly improve execution efficiency, especially for larger models. However, its effectiveness is influenced by factors such as model size and the overhead introduced by disk accesses. As a result, clients must find the optimal balance between memory utilization and execution time, tailoring the partitioning strategy to meet the specific needs of their use case and system constraints.

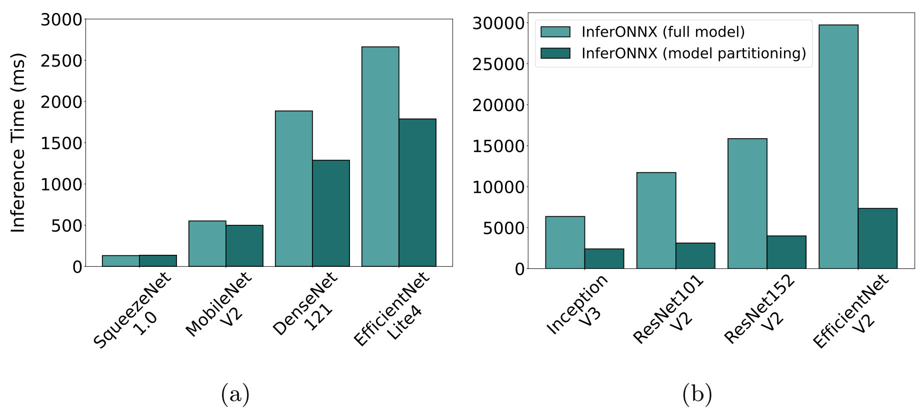


Fig. 5. Performance evaluation of InferONNX during inference on small (a) and large (b) models, comparing full model execution with model partitioning.

7 Related work

In this section, we present existing approaches for confidential machine learning inference, including cryptographic techniques and TEEs.

Cryptographic approaches have long been a key solution for securing sensitive data. Rivest et al. [36] first introduced Homomorphic Encryption (HE), which enables computations on encrypted data without decrypting it, ensuring that third parties can handle data securely. Gentry [18] later developed Fully Homomorphic Encryption (FHE), which supports arbitrary computations on ciphertext, making it a powerful but computationally expensive solution for secure data processing. Multi-Party Computation (MPC) is another cryptographic approach, allowing multiple parties to perform a joint computation while keeping their input private. Some approaches combine these methods for improved performance. Bourse et al. [8] proposed Fast HE Discretized Neural Network (Fast HE DiNN), which leverages both HE and discrete neural networks to reduce the computation complexity of HE. Although this method sacrifices some accuracy, it improves efficiency in cases where DiNNs are used for training instead of discretizing during inference. Xue et al. [46] addressed limitations in HE by proposing a multi-key FHE scheme, which improves privacy protection for client data but does not guarantee model privacy. Several works, such as SecureML [33], MiniONN [30] and Chameleon [35] have leveraged MPC for secure inference. Gazelle [21] combined FHE and MPC to achieve better performance, though it requires two-party computation.

To overcome the limitations of cryptographic methods, hardware-based TEEs like Intel SGX [13], ARM TrustZone [45], and ARM CCA [25], are being used to create isolated execution environments for machine learning operations. One example is GuaranTEE [40], which builds on ARM CCA to provide a secure framework for protecting machine learning inference on edge devices. In contrast, our work utilizes Occlum, a libOS for Intel SGX, which simplifies the development and deployment of applications within Intel SGX. Fortanix Enclave Development Platform [15] is another widely used libOS for Intel SGX, offering strong cloud-native integration and support for multiple programming languages, particularly Rust, which offers memory safety features.

Intel SGX faces strict memory constraints, which necessitate further optimization techniques, such as model partitioning, to improve performance while staying within enclave memory limits. Slalom [41] presents a system for secure machine learning inference that partitions model execution between a trusted enclave (Intel SGX) and an untrusted CPU, enabling efficient execution while maintaining model confidentiality. TEESlice [28] highlights the limitations of existing post-training model partitioning approaches under knowledgeable adversaries and proposes a partition-before-training method that isolates privacy-sensitive weights within TEEs, while offloading the remaining, less sensitive weights to GPUs. Soter [38] partitions sensitive model layers, executing them inside Intel SGX enclaves, while the remaining layers run on GPUs to accelerate inference. While this approach reduces computation time, it introduces performance overhead due to frequent CPU-GPU context switching, especially for large models. Additionally, the integrity checks required to ensure correct results can increase latency by up to 1.27x in some cases. Approaches like Soter and TEESlice leverage GPU acceleration to address the memory constraints of Intel SGX, however they still require careful orchestration of data movement and memory usage to avoid performance bottlenecks and ensure security guarantees. SecureTF [34] provides a secure enclave-based runtime specifically tailored for TensorFlow models. MEDIA [27] partitions Deep Neural Networks (DNNs) into multiple partitions in an edge cloud environment, addressing the limitations posed by cyclic graphs. It optimizes inference by routing models to one of N servers, selecting the server that achieves the best inference time. Unlike our approach, which strictly partitions models based on EPC capacity, MEDIA prioritizes inference time, allowing some degree of exceeding the EPC limit as long as overall performance remains efficient. Finally, MLCapsule [20] enables client-side execution while keeping the model and computations confidential, allowing service providers to protect their intellectual property and business models. Similarly, our system supports this model using ONNX, a format that further confines execution to the client's environment.

8 Limitations

Model partitioning is a practical method for managing execution within the memory limitations of hardware enclaves. Nonetheless, the current approach has certain constraints. One such limitation arises when an individual operator in the model exceeds the capacity of the EPC. In these cases, partitioning has limited effect, as the large operator introduces execution overhead that cannot be avoided through inter-operator partitioning alone. The current design does not address such cases, and support for intra-operator partitioning remains an area for future investigation.

Another limitation concerns models in which partitions depend on intermediate state produced by preceding operators. Preserving and managing this state across partition boundaries introduces additional memory requirements. When executing model partitions sequentially, the cumulative memory footprint can exceed the EPC capacity. In such cases, the intended benefits of partitioning are diminished, as the overhead from enclave memory constraints—such as page swapping—remains significant, potentially affecting overall performance.

9 Conclusions

In this work, we propose InferONNX, a lightweight machine learning inference service designed to run within Intel SGX. Our approach enables model providers to securely deploy their models, allowing clients to perform inference on sensitive data while preserving both model confidentiality and data privacy. In addition, it tackles Intel SGX's memory constraints using two key mechanisms: a compact runtime with a minimal memory footprint and model partitioning that reduces memory usage during inference. Our evaluation shows that InferONNX reduces the overhead associated with full model execution by approximately 1.5× to 4×.

Acknowledgment

We thank the shepherd and the anonymous reviewers for their helpful comments. This work was supported by dAIEdge funded by the European Commission under Grant 101120726. The source code is available at https://github.com/Konstantina155/InferONNX.

References

1. Anakin inference framework. https://github.com/PaddlePaddle/Anakin
2. Mobile AI Compute Engine (MACE) inference framework. https://github.com/XiaoMi/mace
3. NCNN inference framework. https://github.com/Tencent/ncnn
4. ONNX Model Zoo. https://onnx.ai/models/
5. Valgrind Massif: a heap profiler. https://valgrind.org/docs/manual/ms-manual.html

6. Bai, J., Lu, F., Zhang, K., et al.: ONNX: Open Neural Network eXchange. https://github.com/onnx/onnx (2019)
7. Bao, B.: ONNX models. https://github.com/BowenBao/models-1
8. Bourse, F., Minelli, M., Minihold, M., Paillier, P.: Fast Homomorphic Evaluation of Deep Discretized Neural Networks. In: 38th Annual International Cryptology Conference. pp. 483–512 (2018)
9. Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., Sadeghi, A.R.: Software grand exposure: SGX cache attacks are practical. In: Proceedings of the 11th USENIX Conference on Offensive Technologies. pp. 11–11 (2017)
10. Chalkiadakis, N., Deyannis, D., Karnikis, D., Vasiliadis, G., Ioannidis, S.: The Million Dollar Handshake: Secure and Attested Communications in the Cloud. In: 2020 IEEE 13th International Conference on Cloud Computing (CLOUD). pp. 63–70 (2020). https://doi.org/10.1109/CLOUD49709.2020.00022
11. Chen, G., Chen, S., Xiao, Y., Zhang, Y., Lin, Z., Lai, T.H.: SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In: 2019 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 142–157 (2019)
12. Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., Guestrin, C., Krishnamurthy, A.: TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In: Proceedings of the 13th USENIX conference on Operating Systems Design and Implementation. pp. 579–594 (2018)
13. Costan, V., Devadas, S.: Intel SGX Explained. IACR Cryptology ePrint Archive pp. 1–118 (2016)
14. Duy, K.D., Noh, T., Huh, S., Lee, H.: Confidential machine learning computation in untrusted environments: A systems security perspective. IEEE Access **9**, 168656–168677 (2021)
15. FortanixEDP: Fortanix Enclave Development Platform. https://edp.fortanix.com/
16. G. Peskine, M. Pégourié-Gonnard, et al.: Mbed-TLS library. https://github.com/Mbed-TLS/mbedtls
17. Gallego, A., Odyurt, U., Cheng, Y., Wang, Y., Zhao, Z.: Machine Learning Inference on Serverless Platforms Using Model Decomposition. In: Proceedings of the IEEE/ACM 16th International Conference on Utility and Cloud Computing. pp. 1–6. Association for Computing Machinery (2024)
18. Gentry, C.: Fully Homomorphic Encryption Using Ideal Lattices. In: Proceedings of the forty-first annual ACM symposium on Theory of computing. pp. 169–178 (2009)
19. Götzfried, J., Eckert, M., Schinzel, S., Müller, T.: Cache Attacks on Intel SGX. In: Proceedings of the 10th European Workshop on Systems Security. pp. 1–6 (2017)
20. Hanzlik, L., Zhang, Y., Grosse, K., Salem, A., Augustin, M., Backes, M., Fritz, M.: MLCapsule: Guarded Offline Deployment of Machine Learning as a Service. In: 2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW). pp. 3295–3304 (2021)
21. Juvekar, C., Vaikuntanathan, V., Chandrakasan, A.: GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In: Proceedings of the 27th USENIX Conference on Security Symposium. pp. 1651–1669 (2018)
22. Lauriola, I., Lavelli, A., Aiolli, F.: An introduction to deep learning in natural language processing: Models, techniques, and tools. Neurocomputing **470**, 443–456 (2022)
23. Lee, T., Lin, Z., Pushp, S., Li, C., Liu, Y., Lee, Y., Xu, F., Xu, C., Zhang, L., Song, J.: Occlumency: Privacy-preserving remote deep-learning inference using sgx. In:

The 25th Annual International Conference on Mobile Computing and Networking. pp. 1–17 (2019)

24. Li, F., Li, X., Gao, M.: Secure MLaaS with Temper: Trusted and Efficient Model Partitioning and Enclave Reuse. In: Proceedings of the 39th Annual Computer Security Applications Conference. pp. 621–635 (2023)
25. Li, X., Li, X., Dall, C., Gu, R., Nieh, J., Sait, Y., Stockwell, G.: Design and Verification of the Arm Confidential Compute Architecture. In: 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). pp. 465–484 (2022)
26. Li, Y., Zeng, D., Gu, L., Chen, Q., Guo, S., Zomaya, A., Guo, M.: Lasagna: Accelerating secure deep learning inference in sgx-enabled edge cloud. In: Proceedings of the ACM Symposium on Cloud Computing. pp. 533–545 (2021)
27. Li, Y., Zeng, D., Gu, L., Guo, S., Zomaya, A.Y.: DNN Partitioning and Assignment for Distributed Inference in SGX Empowered Edge Cloud. In: 2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS). pp. 635–644 (2024)
28. Li, Ding and Zhang, Ziqi and Yao, Mengyu and Cai, Yifeng and Guo, Yao and Chen, Xiangqun: TEESlice: Protecting Sensitive Neural Network Models in Trusted Execution Environments When Attackers have Pre-Trained Models. ACM Trans. Softw. Eng. Methodol. (2024)
29. Liu, B., Ding, M., Shaham, S., Rahayu, W., Farokhi, F., Lin, Z.: When machine learning meets privacy: A survey and outlook. ACM Computing Surveys (CSUR) **54**(2), 1–36 (2021)
30. Liu, J., Juuti, M., Lu, Y., Asokan, N.: Oblivious Neural Network Predictions via MiniONN transformations. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 619–631 (2017)
31. Malik, M., Malik, M.K., Mehmood, K., Makhdoom, I.: Automatic speech recognition: a survey. Multimedia Tools and Applications **80**, 9411–9457 (2021)
32. Mathieu Poumeyrol, et al.: Tract inference engine. `https://github.com/sonos/tract`
33. Mohassel, P., Zhang, Y.: SecureML: A system for scalable privacy-preserving machine learning. In: 2017 IEEE Symposium on Security and Privacy (SC). pp. 19–38 (2017)
34. Quoc, D.L., Gregor, F., Arnautov, S., Kunkel, R., Bhatotia, P., Fetzer, C.: secureTF: A Secure TensorFlow Framework. In: Proceedings of the 21st International Middleware Conference. pp. 44—59 (2020)
35. Riazi, M.S., Weinert, C., Tkachenko, O., Songhori, E.M., Schneider, T., Koushanfar, F.: Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications. In: Proceedings of the 2018 on Asia Conference on Computer and Communications Security. pp. 707–721 (2018)
36. Rivest, R.L., Adleman, L., Dertouzos, M.L.: On data banks and privacy homomorphisms. Foundations of Secure Computation **4**(11), 169–180 (1978)
37. Shamshirband, S., Fathi, M., Dehzangi, A., Chronopoulos, A.T., Alinejad-Rokny, H.: A review on deep learning approaches in healthcare systems: Taxonomies, challenges, and open issues. Journal of Biomedical Informatics **113**, 103627 (2021)
38. Shen, T., Qi, J., Jiang, J., Wang, X., Wen, S., Chen, X., Zhao, S., Wang, S., Chen, L., Luo, X., Zhang, F., Cui, H.: SOTER: Guarding Black-box Inference for General Neural Networks at the Edge. In: Proceedings of the 2022 USENIX Annual Technical Conference. pp. 1651–1669 (2022)
39. Shen, Y., Tian, H., Chen, Y., Chen, K., Wang, R., Xu, Y., Xia, Y., Yan, S.: Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In: Pro-

ceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 955–970 (2020)

40. Siby, S., Abdollahi, S., Maheri, M., Kogias, M., Haddadi, H.: GuaranTEE: Towards Attestable and Private ML with CCA. In: Proceedings of the 4th Workshop on Machine Learning and Systems. pp. 1—-9 (2024)
41. Tramèr, F., Boneh, D.: Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019 (2019)
42. Tran, K.A., Kondrashova, O., Bradley, A., Williams, E.D., Pearson, J.V., Waddell, N.: Deep learning in cancer diagnosis, prognosis and treatment selection. Genome Medicine **13**, 1–17 (2021)
43. Wang, P., Fan, E., Wang, P.: Comparative analysis of image classification algorithms based on traditional machine learning and deep learning. Pattern Recognition Letters **141**, 61–67 (2021)
44. Wang, W., Chen, G., Pan, X., Zhang, Y., Wang, X., Bindschaedler, V., Tang, H., Gunter, C.A.: Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 2421–2434 (2017)
45. Winter, J.: Trusted computing building blocks for embedded linux-based ARM trustzone platforms. In: Proceedings of the 3rd ACM workshop on Scalable trusted computing. pp. 21–30 (2009)
46. Xue, H., Huang, Z., Lian, H., Qiu, W., Guo, J., Wang, S., Gong, Z.: Distributed Large Scale Privacy-Preserving Deep Mining. In: 2018 IEEE Third International Conference on Data Science in Cyberspace (DSC). pp. 418–422 (2018)