SwiftSNNI: Optimized Scheduling for Secure Neural Network Inference (SNNI) on Multi-Core Systems

Kanwal Batool
k.batool@uva.nl
University of Amsterdam
Amsterdam, Netherlands

Saleem Anwar
s.anwar@hr.nl
Rotterdam University of Applied Sciences
Rotterdam, Netherlands

Francesco Regazzoni
f.regazzoni@uva.nl
University of Amsterdam
Amsterdam, Netherlands
USI Universitá della Svizzera italiana
Lugano, Switzerland

Andy Pimentel
a.d.pimentel@uva.nl
University of Amsterdam
Amsterdam, Netherlands

Zoltán Ádám Mann
zoltan.mann@uni-muenster.de
University of Münster
Münster, Germany

Abstract

Secure Neural Network Inference (SNNI) enables privacy-preserving inference on encrypted data with strong cryptographic guarantees. However, practical deployments suffer from high preprocessing overhead, significant communication costs, and sequential execution. These limitations lead to low throughput, underutilized system resources, long queueing delays, and poor scalability.

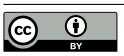
This work introduces *SwiftSNNI*, a unified, resource-aware scheduling framework for SNNI. It implements a hybrid offline–online strategy that orchestrates offline preprocessing ($T_{\text{pre},i}$) and online inference ($T_{\text{on},i}$) jobs to maximize parallelism. By formulating SNNI scheduling as a constrained optimization problem, *SwiftSNNI* overlaps $T_{\text{pre, i}}$ phase execution of future requests with active $T_{\text{on, j}}$ jobs. *SwiftSNNI* also incorporates optional advance notices to enable proactive $T_{\text{pre},i}$, which further reduces average input delay (D).

Evaluations using five benchmark neural networks (M1, M2, HiNet, AlexNet, VGG-16) under diverse workloads and stochastic arrival rates confirm substantial performance gains. Compared to a parallelized sequential baseline (MS-SHARK), *SwiftSNNI* achieves up to 97% lower average input delay (D), a 81% reduction in makespan ($\approx$ 5.4× speedup), and delivers 5.6× increase in throughput. Furthermore, *SwiftSNNI* reduces average waiting time (W) by over 99%, demonstrating robust starvation prevention for high-concurrency workloads. *SwiftSNNI* supports concurrent execution, scales to larger neural networks, and provides an efficient runtime for SNNI deployments. The [1] *SwiftSNNI* implementation is available online.

CCS Concepts

• **Security and privacy** → **Privacy-preserving protocols**; • **Software and its engineering** → **Scheduling**; *Software performance*; • **Computer systems organization** → *Real-time systems*; • **Computing methodologies** → *Neural networks.*

[1] https://github.com/KanwalBat00l/SwiftSNNI



ICPE '26, Florence, Italy

ACM ISBN 979-8-4007-2325-4/2026/05
https://doi.org/10.1145/3777884.3797005

Keywords

Secure Neural Network Inference, Task Scheduling, Multi-core systems, Offline–Online Scheduling, Resource Management, Performance Optimization, Concurrency.

ACM Reference Format:
Kanwal Batool, Saleem Anwar, Francesco Regazzoni, Andy Pimentel, and Zoltán Ádám Mann. 2026. SwiftSNNI: Optimized Scheduling for Secure Neural Network Inference (SNNI) on Multi-Core Systems. In *Proceedings of the 17th ACM/SPEC International Conference on Performance Engineering (ICPE '26), May 04–08, 2026, Florence, Italy*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3777884.3797005

1 Introduction

Driven by sustained advances in Machine Learning (ML), recent years have witnessed a growing demand for secure Machine Learning-as-a-Service (MLaaS), where pre-trained neural networks are deployed in the cloud to serve large-scale inference queries. However, MLaaS raises significant privacy and security concerns [37], which have led to a growing demand for Secure Neural Network Inference (SNNI). In this setting, pre-trained neural networks run inference on encrypted inputs. SNNI employs cryptographic primitives such as Homomorphic Encryption (HE) [23] and Secure Multiparty Computation (MPC) [9, 10, 111] to enable privacy-preserving inference in sensitive domains such as healthcare and finance [93]. Typically, the client has the input x, the server has the neural network F, and both parties jointly compute $F(x)$ so that the client only learns the output and the server learns nothing about x or $F(x)$. Figure 1 illustrates the SNNI process and its privacy guarantees.

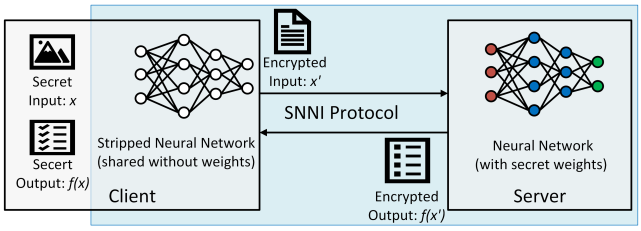


Figure 1: Illustration of SNNI.

In practice, SNNI servers must handle multiple concurrent requests over heterogeneous neural networks. Although state-of-the-art (SoTA) SNNI frameworks [5, 48] ensure correctness and privacy, their system-level performance remains a critical bottleneck. This work identified the key bottlenecks. Protocol-level overheads include extensive offline preprocessing and high communication costs for non-linear layers, while cryptographic operations are compute-intensive. System-level limitations arise because most SNNI frameworks follow an inter-request sequential execution model. Current SNNI frameworks process only one request at a time even on multi-core hardware, and treat cryptographic computations as monolithic jobs. This results in underutilized system resources, a high average input delay, and poor scalability under concurrent workloads.

To improve efficiency, numerous SNNI protocols adopt a two-phase *preprocessing model* [40, 42, 44, 58, 91, 108]: an *offline preprocessing phase (T_{pre})* that generates input-independent correlated randomness, and *an online inference phase (T_{on})* that performs input-dependent secure computation. The T_{pre} phase often takes more than 70% of the total runtime. Despite using multi-core optimizations (such as OpenMP [79] or Eigen [38]) for the modular arithmetic within a single inference, existing systems execute these two phases sequentially for each request. This fails to exploit their independence to overlap the T_{pre} phase of future requests with the T_{on} phase execution of current ones.

Throughput optimizations of SNNI frameworks exist, such as batching [44, 84] or HE-based methods, such as [35], ciphertext slot utilization [13], and spectral inference [105]. However, these approaches add memory overhead and batching delays, which reduce responsiveness under dynamic workloads. Conventional task scheduling methods [80, 98] overlook the two-phase structure and substantial offline preprocessing overhead of SNNI. Without coordination, $T_{\text{pre, i}}$ jobs and $T_{\text{on, i}}$ jobs compete for shared resources.

We examine SNNI frameworks from a scheduling perspective, where two key challenges arise: ① *Scalability and concurrency:* Sequential execution limits scalability. Each request involves computationally intensive cryptographic operations and variable memory requirements. Processing only one job at a time leaves system resources underutilized. During the single-threaded T_{pre}, the $(P-1)$ cores remain idle. This is true even if the T_{on} phase is parallelized across all cores. Multi-client workloads and large neural networks exacerbate this issue, which results in increased average input delay and reduced throughput. ② *Online task assignment:* Mapping incoming client requests to processors is non-trivial. The scheduler must respect CPU, memory, and cryptographic constraints while enabling concurrent execution without conflicts.

These challenges motivate our central research questions: (i) *How can multiple inference jobs run concurrently to maximize resource utilization and throughput while minimizing average input delay (D) ?* (ii) *How can incoming client requests be scheduled in real time under CPU, memory, and cryptographic constraints?*

To address SNNI scheduling challenges, we propose *SwiftSNNI*. This work is the first to formulate multi-client SNNI request scheduling as an explicit, constrained optimization problem. *SwiftSNNI* builds on 2PC with offline preprocessing [6] and manages the contention between requests from different clients for the server's CPU and memory resources. Our contributions are:

- We characterize how protocol-level dependencies and system-level constraints (cores, memory, and job dependencies) jointly limit concurrent request execution and throughput in secure neural network inference (SNNI) deployments.
- We design *SwiftSNNI*, a two-stage hybrid framework that coordinates offline preprocessing ($T_{\text{pre, i}}$) and online inference ($T_{\text{on, i}}$) jobs to maximize parallelism.
- We propose a resource-aware scheduler that supports both *First-Come-First-Served (FCFS)* and *priority-based* scheduling policies. It manages job assignment and concurrency across heterogeneous workloads and variable arrival rates. Advance notices enable early T_{pre} phase execution, which effectively hides the time cost of the T_{pre} phase and reduces average input delay D.
- We evaluated *SwiftSNNI* on five benchmark neural networks (M1, M2, HiNet, AlexNet, VGG-16) and demonstrated up to 97% lower average input delay (D) and a 81% shorter makespan ($\approx$ 5.4× speedup) compared to a parallelized sequential baseline (MS-SHARK) [40]. *SwiftSNNI* delivers a 5.6× increase in throughput and a 99% reduction in average waiting time (W), which demonstrates robust starvation prevention. Incorporating 25% advance notice requests further reduces D across all workloads.

Roadmap. The paper begins with the challenges and significance of SNNI in Section 1, followed by a background and a motivating example in Section 2. Section 3 formalizes the SNNI scheduling problem with key objectives and constraints. Section 4 presents *SwiftSNNI*, detailing the system architecture, technical execution flow and components interaction, and scheduling algorithm. The experimental methodology is described in Section 5, with performance evaluation and baseline comparisons in Section 6. Section 7 reviews related work and Section 8 concludes with future directions.

2 Background & Motivation

This section first reviews key concepts and background on SNNI (2.1) and then discusses its limitations with a motivating example (2.2) that drives the design of *SwiftSNNI*.

2.1 Secure Neural Network Inference (SNNI)

SNNI enables privacy-preserving ML inference using cryptographic primitives such as Homomorphic Encryption (HE) [23] and Secure Multiparty Computation (MPC) [112]. Classical SMC protocols include Yao's Garbled Circuit, GMW, and BGM [7, 75]. SNNI can be realized using the generic secure two-party computation (2PC) [36, 112]. Several SNNI frameworks optimize this paradigm for neural networks (see [72] for a survey).

Efficiency is improved using the *offline–online paradigm* [6]. In the offline preprocessing phase (T_{pre}), parties P_0 and P_1 in a specific job ($T_{\text{pre, i}}$) generate input-independent correlated randomness before inputs are known; and in an online inference phase (T_{on}), specific job ($T_{\text{on, i}}$) use this precomputed material enables fast input-dependent computation once client inputs are available. This correlated randomness can be produced via a trusted dealer [9, 42], a standard 2PC [7], or by using specialized 2PC approaches [25]. Each $T_{\text{on, i}}$ job has a strict dependency on its corresponding $T_{\text{pre, i}}$ job. While protocol-level optimizations exist, efficient scheduling and

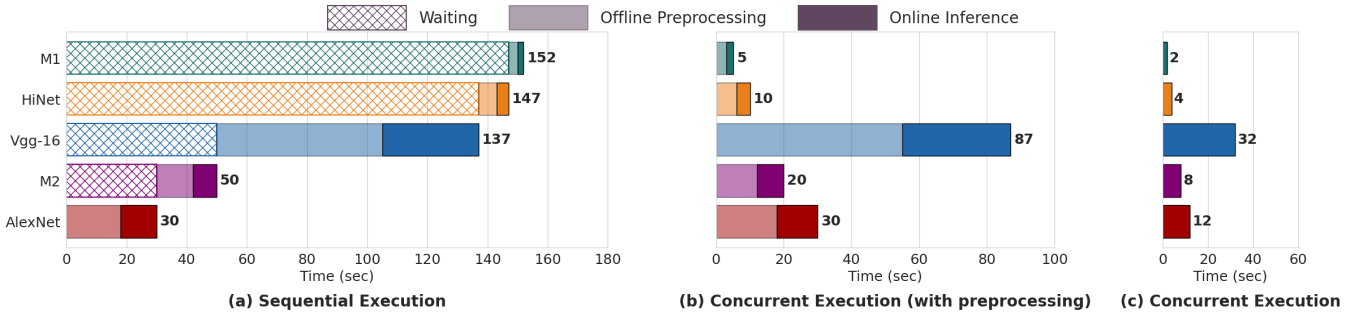


Figure 2: Timeline Comparison of Execution Strategies

resource management remain an open challenge in fully exploiting this two-phase structure across multiple concurrent requests.

Protocol and System-Level Optimizations: Performance bottlenecks in SNNI stem from neural network sizes, memory intensive layers, and communication overhead. Existing optimizations [66] include:

- *Algorithmic and Arithmetic Optimizations:* Techniques based on modular rings and finite fields balance precision, security, and efficiency [16, 74, 76, 104, 105]. Homomorphic encryption with SIMD parallelism further accelerates encrypted linear layer computations [5, 35, 48, 54, 57].
- *Model-Level Optimizations:* Redesigning neural networks into lightweight or binary forms [34, 83, 87, 89] reduces computation and communication costs while maintaining high accuracy. Applying quantization techniques to reduce the precision of modeneural network weights to minimize ciphertext size and communication overhead [1, 87, 92].
- *System-Level and Hardware Optimizations:* Hardware acceleration using GPUs [32, 91, 101, 108] offer high parallelism, but suffer from high memory usage and inefficient handling of non-linear layers. CPU-based frameworks [40, 42, 44, 56, 73, 84, 108] utilize multi-threading (e.g., OpenMP [79]) to speed up the modular arithmetic within a single $T_{\text{on, i}}$ job. Compiler-driven frameworks such as CHET [22], CrypTFlow [63], EzPC [15], MP-SPDZ [56] and SecretFlow-SPU [71] automate data flow optimization, code generation, and protocol selection to improve runtime efficiency.

Despite these improvements, existing techniques primarily focus on *intra-request parallelism* using multiple cores to speed up a single inference. None of these techniques address *intra-request scheduling*, such as how to concurrently execute the $T_{\text{pre, i}}$ job of a pending request while the $T_{\text{on, j}}$ job of an active request is running. Consequently, system resources remain underutilized during the single-threaded T_{pre} phase of the sequential request pipeline. Efficient scheduling and resource management remain critical to realize high throughput (ρ) and low average input delay (D) under stochastic request arrivals.

2.2 Motivating Example

To illustrate the limitations of existing SNNI frameworks (Figure 2), consider a server with 16 CPU cores and 64 GB of memory capable of handling multiple concurrent requests. Each client submits a request specifying a neural network, each with distinct memory and core requirements for $T_{\text{pre, i}}$ and $T_{\text{on, i}}$.

In sequential execution, requests are processed one at a time. For example, the server runs AlexNet first, and allocates all required system resources exclusively. Other neural networks (e.g., VGG-16, HiNet) remain idle. This leads to underutilized system resources, and higher D, as short jobs wait behind long ones.

Concurrent execution improves utilization by overlapping independent jobs within system resource limits. The makespan is then bounded by the longest job rather than the sum of all jobs. However, each request still executes its $T_{\text{pre, i}}$ and $T_{\text{on, i}}$ jobs one after another. This sequential internal execution limits the total throughput and responsiveness.

Efficiency is further improved by exploiting the input independence of the job $T_{\text{pre, i}}$. Because this job does not require the private input data, it can be performed opportunistically during idle periods or in parallel with $T_{\text{on, i}}$ jobs of other requests. Precomputed offline preprocessing material allows $T_{\text{on,i}}$ job to start immediately upon request arrival (Figure 2). This overlap reduces the average input delay and increases throughput.

These insights motivate the design of *SwiftSNNI*. We now formally define SNNI as a constrained scheduling problem under compute and memory constraints, thus capturing the two-phase nature of SNNI protocols, and set the foundation for *SwiftSNNI*.

3 Problem Formulation

We consider a Machine Learning-as-a-Service (MLaaS) that offers secure neural network inference to a set of registered clients:

$$C = \{c_1, c_2, ..., c_K\}.$$

The server is equipped with:
- P homogeneous processor cores.
- M units of memory shared across all cores.

For simplicity, we assume that all processor cores have equal compute capabilities and a uniform access bandwidth to the shared memory.

Let $N = n_1, \ldots, n_{|N|}$ be the set of neural networks supported by the server. Each input i generates two jobs: an offline preprocessing job ($T_{\text{pre, i}}$) and an online inference job ($T_{\text{on, i}}$). These jobs correspond to the two functional phases of the SNNI protocol:

- Offline Pre-processing ($T_{\text{pre, i}}$): This job is model-specific but input-independent. It generates cryptographic material (e.g., Beaver triples) required for secure computation. The index i serves solely as a unique identifier for the specific instance of cryptographic material required to satisfy request i; the actual values of the private input are not needed for this job. Each generated preprocessing file can only be used once and requires a unique $T_{\text{pre, i}}$ for each online inference ($T_{\text{on, i}}$).
- Online Inference ($T_{\text{on}}(i)$): This job is both model-specific and input-specific. It completes the secure inference using the client's specific private input data i and the cryptographic material produced in the $T_{\text{pre, i}}$.

Let $\mathcal{R} = \{r_1, r_2, \ldots, r_{|\mathcal{R}|}\}$ be the set of all requests received by the server. For each request $r \in \mathcal{R}$ let

$$I_r = \{i_{r,1}, \ldots, i_{r,B_r}\}, \qquad B_r = |I_r|$$

be a batch of inputs. The total job set $\mathcal{T}$ for all inputs is defined as:

$$\mathcal{T} = \{T_{\text{pre},i},\ T_{\text{on},i} \mid r \in \mathcal{R},\ i \in I_r\}.$$

Each online inference job $T_{\text{on},i}$ may start only after its corresponding offline preprocessing job $T_{\text{pre},i}$ completes.

The system supports multiple clients $c \in C$, with each client c being capable of issuing requests independently. Each request r can include a batch of inputs I, which allows the system to process multiple input samples within a single request. Requests take the form:

$$r = (c, t, n, I), \text{ where:}$$

- $c \in C$ is the client issuing the request,
- $t > 0$ is the point in time when the request is issued,
- $n \in N$ is the selected neural network that the client requests for SNNI,
- I is a batch of $B = |I|$ client inputs (e.g. images) for which the client requests inference results. The inputs are never sent in plaintext; the server operates only on protected (encrypted) inputs.

A request (r) may be preceded by an advance notice (a) defined as follows:

$$a = (c, t', n, |I|, s), \text{ where:}$$

- t' is the time the client sends the advance notice a, occurring before the request r arrives at time t, such that $t' < t$.
- $|I|$ represents the batch size of the request r.
- s is the size of each input i for resource estimation.

The advance notice serves to inform the server that the client intends to submit a corresponding request in the future. This allows the server to start the offline preprocessing phase (T_{pre}) in advance if sufficient system resources are available. When such an advance notice is sent, the subsequent request includes a reference to it, written as:

$$r_a = (c, t, n, I, a)$$

SwiftSNNI assumes a synchronous communication model, where all clients remain responsive during both offline preprocessing (T_{pre}) and online inference phases (T_{on}). For each input $i \in I$, $T_{\text{on, i}}$ job starts only once its corresponding $T_{\text{pre, i}}$ job is finished. The execution of both jobs is subject to system-level compute and memory constraints.

Constraints. *SwiftSNNI* schedules jobs executions, while the Operating System (OS) scheduler assigns them to processor cores. These requests must satisfy the following constraints:

- Dependency: For every input i, the job $T_{\text{on,i}}$ starts only after the corresponding job $T_{\text{pre, i}}$ finishes.
- Core Usage: At any time t, the total number of concurrent jobs must not exceed the number of available cores P.
- Memory: The total memory of all concurrent active jobs at time t must not exceed the system limit M.

Objective. Let $\mathcal{R}_\tau$ be the set of all requests received by the server during the time interval $[0, \tau]$. The service provider aims to schedule both the $T_{\text{pre, i}}$ and the $T_{\text{on, i}}$ for incoming client requests in such a way that D, the asymptotic average input delay, is minimized, subject to the above constraints. It is defined as:

$$D = \lim_{\tau \to \infty} \frac{1}{|\mathcal{I}_\tau|} \sum_{i \in \mathcal{I}_\tau} d(i), \textit{where}$$

$$d_i = t_{\text{finish, i}} - t_{\text{arrival, i}},$$

- $\mathcal{I}_\tau$ is the set of all individual inputs in the requests in $\mathcal{R}_\tau$ within $[0, \tau]$,
- For each input $i \in \mathcal{I}_\tau$, let d_i denote the delay for that input i. This is defined as the difference between the time when the secure inference process on i was completed ($t_{\text{finish, i}}$) and the time ($t_{\text{arrival, i}}$) the secure inference was requested for input i. Note that this delay is measured independently of any advance notice (a) associated with the input i.

Building on this problem formulation, we now present *SwiftSNNI*, a unified scheduling framework designed to address the identified bottlenecks.

4 *SwiftSNNI*: A Unified Framework for Secure and Efficient SNNI Scheduling

This section introduces *SwiftSNNI*. We first describe its system architecture (4.1), component interaction, and technical execution flow (4.2), followed by the scheduling algorithm (4.3) that governs inter-request concurrency under CPU and memory constraints.

4.1 System Architecture

The *SwiftSNNI* architecture (as shown in Fig. 3) comprises three main components: 4.1.1 the computational infrastructure, including system resources, the model catalog & specifications and execution parameters; 4.1.2 the secure inference service engine, which contains the queue manager, request scheduler, the two-phase SNNI protocol, and the resource monitor; and 4.1.3 the client request interface (CRI). *SwiftSNNI* operates as a middleware layer that manages inter-request concurrency, which ensures that the offline and online phases of different SNNI requests are overlapped to maximize core utilization.

4.1.1 ***Computational Infrastructure.*** It consists of three elements:

System Resources. We consider a single server provisioned with P identical cores and a shared memory of size M. The server hosts multiple neural networks (n) and all active jobs compete for shared system resources such as CPU, memory, and network bandwidth.

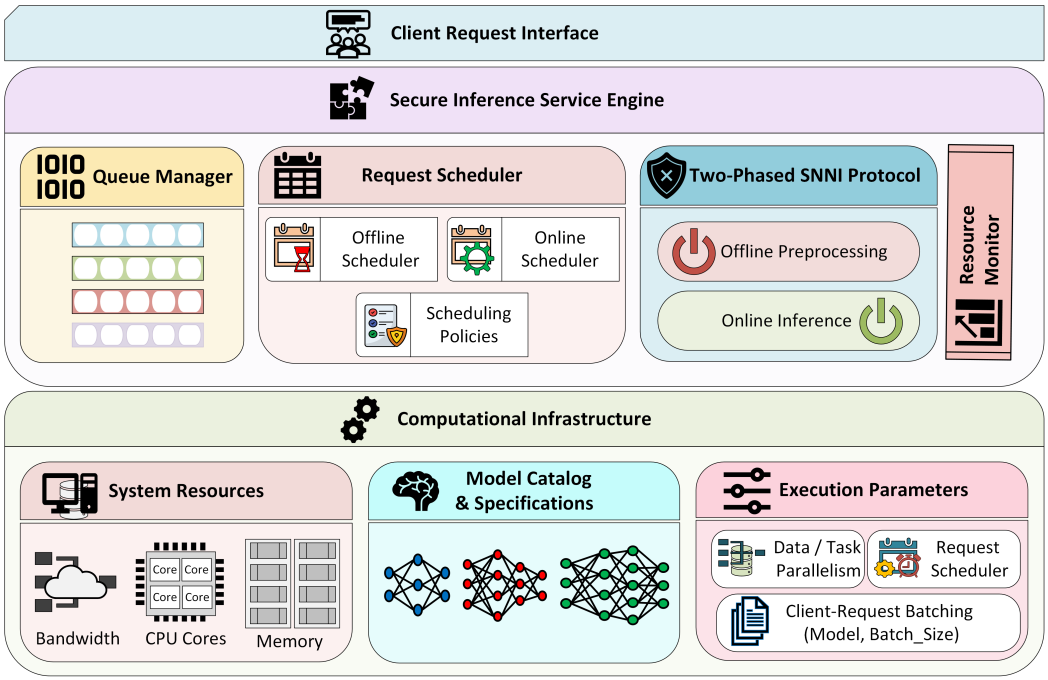


Figure 3: SwiftSNNI High-Level Architecture.

Multiple jobs can use the same neural network (n) simultaneously. While low-level operations such as thread-to-core mapping is delegated to the native OS, *SwiftSNNI* performs high-level orchestration to manage core and memory constraints.

To enable precise resource estimation, the catalog includes profiled metadata specifying the peak memory usage and CPU cycles required for both

Model Catalog & Specifications. *SwiftSNNI* maintains a catalog of supported neural networks $N = \{n_1, \ldots, n_{|N|}\}$. To enable precise resource estimation, the model catalog includes profiled metadata for each neural network architecture (n), which specifies the peak memory usage and CPU cycles required for both the T_{pre} and T_{on} phases as a function of input batch size. The request scheduler uses this information to plan execution and prioritize jobs. To protect the server's intellectual property, clients only see the neural network architecture and required input size. All other metadata and weights remain internal to server. The catalog is extensible and allows the service provider to include new neural networks.

Execution Parameters. These define the operational constraints and configuration for scheduling SNNI jobs. These include: (i) task and data parallelism, where task parallelism regulates the number of concurrent SNNI jobs, and data parallelism manages client-provided input batches $I_r = \{i_{r,1}, \ldots, i_{r,B_r}\}$, with batch size $B\_r = |I\_r|$; and (ii) scheduling policies used by the *Request Scheduler* to determine job order, prioritization, and system resource allocation. These execution parameters ensure the total esource footprint of all active jobs ($T_{\text{pre, i}}$, $T_{\text{on, i}}$) does not exceed the system limits P and M.

The inference timeout is a client-provided parameter sent through the*CRI*. It sets a maximum duration for each secure inference job. This safeguard prevents SNNI requests from blocking indefinitely. It ensures that system resources are released promptly if a job stalls. This mechanism detects slow executions and guarantees predictable behavior under high-concurrency workloads.

4.1.2 ***Secure Inference Service Engine (SISE).***. The SISE engine is the central orchestration module. Unlike existing SNNI frameworks that process requests as monolithic blocks, SISE decomposes each request into two independent schedulable job units: $T_{\text{pre, i}}$ and $T_{\text{on, i}}$. It includes the following modules:

Queue Manager. It coordinates the state transitions of requests (r_1), advance notices (a_1), and their associated jobs ($T_{\text{pre, 1}}$; $T_{\text{on, 1}}$) across four internal queues. These queues are specifically designed to decouple the asynchronous arrival of advance notices from the arrival of actual requests. This enables non-blocking execution of the $T_{\text{pre, i}}$ job. The internal queues are defined as follows:

- Q_{pre}: Holds new requests (r_i) or advance notices (a_i) that are awaiting a ($T_{\text{pre},i}$) job.
- Q_{transit}: Stores advance notices (a_i) that have completed their $T_{\text{pre},i}$ job, but have not yet been matched with a corresponding client request (r_i).
- Q_{waiting}: Contains incoming requests (r_i) that refer to advance notices (a_i) that are currently undergoing $T_{\text{pre},i}$ job.
- Q_{ready}: Includes requests (r_i) that have successfully completed their $T_{\text{pre},i}$ and have the necessary preprocessed files

available. These requests are fully prepared for the execution of the $T_{\text{on},i}$ job and are awaiting the allocation of system resources.

Request Scheduler. It is the core orchestration module of the SISE. The *Request Scheduler* manages inter-request concurrency and implements two distinct policies: *First-Come-First-Served (FCFS)* and *Priority-aware scheduling*. It uses *Execution Parameters* and feedback from the *Resource Monitor* to enforce system constraints.

To prevent the starvation of large jobs, the *Request Scheduler* calculates a dynamic priority score (p_i) for each pending $T_{\text{pre, i}}$ or $T_{\text{on, i}}$ job. This score is governed by an aging function $f(\delta_i)$ that increases priority based on waiting time. This prevents large neural networks from being postponed indefinitely. The mathematical formulation is detailed in Section 4.3.

SwiftSNNI implements this orchestration through two coordinated schedulers that run concurrently and share resource information through the *Resource Monitor*:

- *Offline Scheduler:* It executes $T_{\text{pre, i}}$ jobs for advance notices and new requests. It prioritizes filling idle gaps in CPU utilization by launching $T_{\text{pre, i}}$ jobs whenever spare cores are available.
- *Online Scheduler:* It handles real-time $T_{\text{on, i}}$ jobs and continuously monitors Q_{ready}. It initiates $T_{\text{on, i}}$ job for the next request (r_i) according to the active scheduling policy (either FCFS or Priority-aware) and available system resources.

Two-Phased SNNI Protocol. *SwiftSNNI* operates on top of existing two-phased SNNI protocols. These protocols separate computation into offline preprocessing (T_{pre}) and and online inference (T_{on}) phases. The T_{pre} job prepares essential preprocessed files with cryptographic materials (e.g., Beaver triples or correlated randomness). The corresponding T_{on} job uses these preprocessed files for secure predictions on private input. *SwiftSNNI* treats underlying protocols as black-box components. This abstraction allows *SwiftSNNI* integration with various secure two-party computation (2PC) frameworks without modifying their internal cryptographic operations or security proofs. It provides high-level orchestration while utilizing the strong security guarantees of established cryptographic primitives, including Homomorphic Encryption (HE) [23] and Secure Multiparty Computation (MPC) [9, 10, 111].

Resource Monitor. This component provides real-time tracking of system-wide CPU utilization, memory consumption, and the average input delay (D). To facilitate proactive scheduling, the *Resource Monitor* utilizes a lookup-table approach based on historical profiling [46, 109]. This allows the *Resource Monitor* to estimate specific resource requirements and execution durations for each incoming $T_{\text{pre, i}}$ and $T_{\text{on, i}}$ jobs.

By providing estimates of peak memory footprints and CPU cycle requirements, the *Resource Monitor* allows the *SwiftSNNI* to adjust the concurrency level dynamically. This maximizes throughput and ensures that the combined resource usage of active jobs never exceeds the system resource limits (P and M). Furthermore, the *Resource Monitor* detects runtime deviations from the historical profile. It provides a feedback loop that allows the *Request Scheduler* to refine its offline and online prioritization for both $T_{\text{pre, i}}$ and $T_{\text{on, i}}$ jobs, which in turn improves resource allocation for future requests.

4.1.3 Client Request Interface (CRI). The CRI serves as the front-end gateway, runs on a dedicated TCP port to manage communication between clients and the internal *Request Scheduler*. A key function of the CRI is the coordination of advance notices: if an incoming request (r_1) arrives with a reference to an advance notice (a_1) whose corresponding ($T_{\text{pre, 1}}$) job has already been completed, the *CRI* facilitates the immediate transition of that request to Q_{ready}. By bypassing the Q_{ready} stage and its associated queuing delay and execution time entirely, the CRI effectively "hides" the time cost of the $T_{\text{pre, 1}}$ from the client's perspective, which reduces the average input delay (D).

4.2 Technical Execution Flow and Component Interactions

Next, we illustrate an example. Figure 4 shows the step-by-step workflow of how a request moves through *SwiftSNNI*, from enqueueing and scheduling to execution of offline preprocessing ($T_{\text{pre, i}}$) and online inference ($T_{\text{on, i}}$) jobs under the privacy constraints imposed by the two-phased SNNI protocol.

① A registered client submits a request specifying the neural network and input batch. Requests can be full (with actual input) or an optional advance notice for the T_{pre} phase. Full requests may reference a previous advance notice or arrive independently.

To initiate the process, the client reads *Execution Parameters* from a local configuration file, such as the paths to the secure inference binaries, the command template used to run the inference, and the inference timeout. The client then connects to the *CRI* via TCP and sends the request. The client asynchronously waits for a response based on the predefined inference timeout. It receives an assigned execution port, the required thread count and proceeds with execution. This allows the client to support multiple concurrent requests without blocking.

② On the server side, the *CRI* authenticates the client, validates the incoming requests, and then passes them to the *Queue Manager*. As shown in Figure 4 (Transition between Step 3 and 4), the *Request Scheduler* immediately parses the request metadata to determine the entry queue based on the existence of an advance notice.

③ The *Queue Manager* organizes requests across four internal queues based on the ($T_{\text{pre, i}}$) status of the request and the requirements of the system resources (defined in 4.1.2). This multi-queue decoupling design is essential for handling asynchronous *Advance Notices* buffer work in Q_{transit} without stalling the execution of arrived requests in Q_{waiting}.

New requests (r_i) or advance notices (a_i) first enter Q_{pre} for the offline preprocessing job ($T_{\text{pre, i}}$). Once ($T_{\text{pre, i}}$) job is complete, advance notices (a_i) move to Q_{transit} until they match the corresponding request ($r_i a_i$). This queue acts as a buffer for advance notices (a_i). Requests referencing ongoing advance notices (a_i) remain in Q_{waiting}, which clearly distinguishes them from requests that are still in waiting for the $T_{\text{pre, i}}$ job. If a request arrives without the corresponding advance notice (a_i), it bypasses Q_{transit} and Q_{waiting} and proceeds directly to Q_{ready} once ($T_{\text{pre, i}}$) job is complete. These requests in Q_{ready} then enter the online inference job ($T_{\text{on,i}}$).

④ The *Request Scheduler* retrieves client requests from the *CRI*, then parses and wraps each request into a job object with metadata

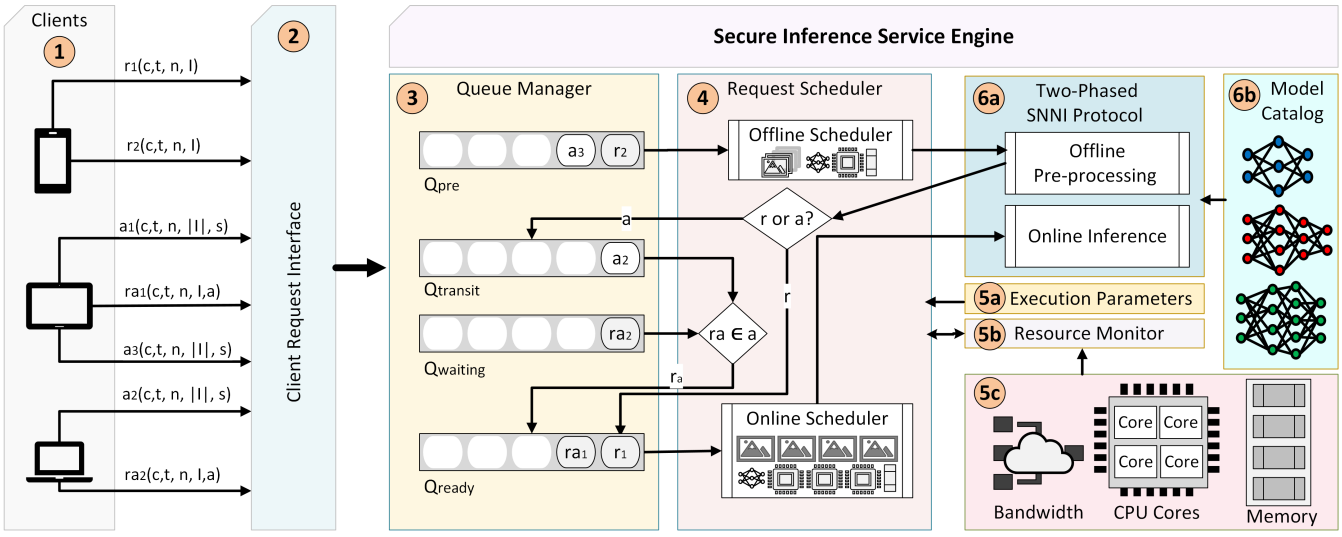


Figure 4: SwiftSNNI Scheduling Framework: Technical Execution Flow and Component Interactions.

(e.g., neural network type n, batch size B_r, timestamps, client ID, and references to any associated advance notice a (if one exists)). The *Request Scheduler* uses *Model Catalog* to check for existing preprocessed files for a requested neural network (n) and the input batch exist for this client, it is used. Otherwise, the request is enqueued for the new $T_{\text{pre, i}}$ job to generate preprocessed file specific to this client request.

⑤ Using *Execution Parameters* and dynamic feedback from the *Resource Monitor*, the *Request Scheduler* evaluates system resources, including CPU cores, memory, and bandwidth, before assigning jobs. The *Resource Monitor* tracks active threads, queued requests, and system resource usage to ensure that scheduling stays within system limits. The *Request Scheduler* monitors ready requests in Q_{ready} whose $T_{\text{pre, i}}$ job is complete and applies the active scheduling policy (*FCFS* or *Priority-aware scheduling*) to order the pending $T_{\text{on, i}}$ jobs. System resources are allocated to maximize throughput and minimize average input delays D, and clients are notified with execution details so that the job can proceed.

⑥ *SwiftSNNI* is a server-side scheduling framework. Execution begins when *SwiftSNNI* assigns a dedicated thread to each job, configured with its neural network (n), input batch size (B), thread count and network port. *SwiftSNNI* executes the necessary jobs of the *Two-Phased SNNI Protocol*. $T_{\text{pre, i}}$ jobs are performed as required. T_{pre} jobs submitted through advance notices are executed in the background. Preprocessed files are reserved for their respective online inference jobs $T_{\text{on, i}}$ jobs and regenerated when needed under a no-reuse policy.

Once the T_{pre} phase completes or a preprocessed file is available, the $T_{\text{on, i}}$ job is executed using the allocated system resources. During the T_{on} phase, *SwiftSNNI* uses the prepared preprocessed files and allocated system resources exclusively until the $T_{\text{on, i}}$ job is completed. Finally, the *SwiftSNNI* scheduling framework assigns jobs to available CPU cores using the active scheduling policy (either *FCFS* or *Priority-aware*), as discussed in Section 4.3.

⑦ Upon job completion of the $T_{\text{on, i}}$, *SwiftSNNI* logs execution metadata, including timestamps, execution status flags, and exit codes to mark the job as finished. Temporary preprocessing artifacts generated during $T_{\text{pre, i}}$ are cleared according to the no-reuse policy to ensure security and correctness for future SNNI requests. *SwiftSNNI* then releases the job, while the final system resource clean up and reuse is handled by the native OS. The *Resource Monitor* updates the neural network's profile with the actual runtime data and peak system resource consumption of the specific $T_{\text{pre, i}}$ and $T_{\text{on, i}}$ jobs. This feedback loop improves the accuracy of the profiling table (Π) and the execution length estimates (ℓ_i) for future scheduling cycles (see 4.3). Finally, the *CRI* returns the execution status or the encrypted inference results to the client.

4.3 SwiftSNNI Scheduling Algorithm

The *SwiftSNNI* scheduling algorithm 1 operates continuously while active jobs exist in the system. It supports two scheduling policies: *First-Come-First-Served (FCFS)* and *Priority-aware* strategies.

The Procedure 1 (HandleNewRequests) performs request classification upon arrival. The incoming client requests are sorted and enqueued into Q_{pre}, Q_{transit}, Q_{waiting}, or Q_{ready} according to their type, offline preprocessing (T_{pre}) requirements, and dependency state (Lines 1–10). Advance notices (a_i) are placed in Q_{pre}, and dependent requests ($r_{a,i}$) are held in Q_{waiting} until their corresponding a_i transitions to Q_{transit} or completes offline preprocessing ($T_{\text{pre, i}}$).

In *Priority-aware* mode, the *Request Scheduler* calculates a dynamic priority score(p_i) for each specific job. Both the *Offline* and *Online schedulers* use this score to prioritize jobs in their respective queues. The score applies to both ($T_{\text{pre, i}}$ or $T_{\text{on, i}}$) instances and is computed as:

$$\text{score}(p_i) = \ell_i - \beta \cdot \delta_i \tag{1}$$

where:

- ℓ_i denotes the estimated execution length (burst time) of the job T_i, retrieved from the profiling table (Π),

- δ_i is the waiting time (age) of the job since its arrival in the queue,

- β is a weighting factor that balances the trade-off between throughput and fairness.

Here, $f(\delta_i) = \delta_i$) is a linear aging function that maps the job's waiting time to a priority contribution. We subtract the δ_i term from the estimated burst time ℓ_i. As a job ages, its total priority score($\mathrm{p_i}$) decreases. Since the *Request Scheduler* selects the job with the minimum score, a lower score represents a higher priority. This ensures that large jobs priority increases over time and eventually, these jobs move to the front of the queue ahead of new, shorter jobs. This prevents starvation and ensures fairness for all requests. In *FCFS mode*, the *Request Scheduler* ignores Equation 1 and prioritizes jobs based strictly on their arrival order.

Procedure 2: *SwiftSNNI_*Execution governs the continuous scheduling and execution (Lines 11–38). It is the main scheduling loop. It continuously synchronizes with the *Resource Monitor* to track CPU and memory availability based on the profiling table (Π) (Line 13).

- *Offline Preprocessing (Lines 14–19)*: It orders $T_{\mathrm{pre,\,i}}$ jobs based on the active scheduling policy. In *Priority-aware mode*, it calculates the priority score($\mathrm{p_i}$) from Equation 1. It then sorts Q_{pre} in ascending order to prevent starvation of large neural networks. Finally, *SwiftSNNI* starts jobs that fit within memory M and core P limits.
- *Queue Transitions (Lines 20–25)*: It resolves job dependencies and promotes requests to Q_{ready} once their corresponding ($T_{\mathrm{pre,\,i}}$) job completes.
- *Online Scheduling (Lines 26–33)*: It manages $T_{\mathrm{on,\,i}}$ jobs. It selects the next job based on the active policy (earliest arrival for *FCFS* or minimum score($\mathrm{p_i}$) for *Priority-aware*). It identifies the least-loaded core (p^*). Then, the *Request Scheduler* assigns the job if system constraints are satisfied.
- *Execution Completion (Lines 34–38)*: This step finalizes the request. It releases system resources and logs performance data to update the profiling table (Π).

Time Complexity. The runtime overhead of *SwiftSNNI* is dominated by the dynamic prioritization and resource-checking of queued jobs. Let r be the total number of jobs currently in the Q_{pre} & Q_{ready} queues and P be the number of CPU cores.

Procedure 1 (Request Classification), queue transitions, and resource monitoring scale linearly with the number of new arrivals and are negligible. In each scheduling iteration of Procedure 2, the *Request Scheduler* calculates the priority score($\mathrm{p_i}$) and then sorts the queues to prevent starvation (Lines 14–19 and Lines 26–29). This step requires $O(r \log r)$ time. Next, it maps the highest-priority job to the least-loaded core (Lines 28–33). This involves a search across both the ready jobs and available cores, contributing $O(r \cdot P)$. Consequently, the worst-case time complexity per scheduling iteration is:

$$\mathbf{O(r \log r + r \cdot P)}$$

In practice, r and P are moderate in server-grade SNNI deployments. This makes the scheduling overhead insignificant compared to the compute-intensive secure inference execution time, which typically involves heavy modular arithmetic and high communication costs.

Algorithm 1: SwiftSNNI Scheduling Algorithm

Input:
$\mathcal{R}$: client requests,
P: CPU cores, M: Shared memory pool,
Q: queues, Q=Q_{pre}, Q_{transit}, Q_{waiting}, Q_{ready},
Π: Profiling table, E: Execution Parameters

Output: Scheduling and execution decisions

1 **Procedure 1: HandleNewRequests(**R, Q**)**
2 **ForEach** $r_i \in R$ **do**
3 **if** $r_i.type = a_i$ **then**
4 enqueue(Q_{pre}, a_i)
5 **else if** $r_i.type = r_{a,i} \wedge a_i \neq \emptyset \wedge a_i \in Q_{transit}$ **then**
6 enqueue(Q_{ready}, r_i)
7 **else if** $r_i.type = r_{a,i} \wedge a_i \neq \emptyset \wedge a_i \in Q_{pre}$ **then**
8 enqueue($Q_{\mathrm{waiting}}, r_i$)
9 **else**
10 enqueue(Q_{pre}, r_i)

11 **Procedure 2: SwiftSNNI_Execution(**P, M, Q, Π, E**)**
12 **while** *system has active jobs* **do**
13 update(ResourceMonitor, current_load(P), available_memory(M), Π);
 `// − Offline Preprocessing (Priority-based) −`
14 **ForEach** $T_{pre,i} \in Q_{pre}$ **do**
15 score ($\mathrm{p_i}$) $\leftarrow \ell_i - \beta \cdot \delta_i$
16 sort Q_{pre} in ascending order of score ($\mathrm{p_i}$);
17 **ForEach** $T_{pre,i} \in Q_{pre}$ **do**
18 **if** *fits_resources*($T_{pre,i}, P, M, \Pi$) **then**
19 start($T_{\mathrm{pre},i}$)
 `// − Queue Transitions −`
20 **ForEach** *completed* $T_{pre,i}$ **do**
21 **if** $T_{pre,i} = a_i$ **then**
22 move $a_i \rightarrow Q_{\mathrm{transit}}$;
23 move linked $r_{a,i} \in Q_{\mathrm{waiting}} \rightarrow Q_{\mathrm{ready}}$;
24 **else**
25 move $T_{\mathrm{pre},i} \rightarrow Q_{\mathrm{ready}}$;
 `// − Online Scheduling −`
26 **ForEach** $T_{on,i} \in Q_{ready}$ **do**
27 score ($\mathrm{p_i}$) $\leftarrow \ell_i - \beta \cdot \delta_i$
28 $T^* \leftarrow \arg\min_{T_{\mathrm{on},i} \in Q_{\mathrm{ready}}} score(\mathrm{p_i})$;
29 $p^* \leftarrow \arg\min_{p \in P} w_q(p)$;
30 **if** *fits_resources*(T^*, p^*, M, Π) **then**
31 assign(T^*, p^*);
32 $w_q(p^*) \leftarrow w_q(p^*) + \ell_i$;
33 remove(T^*, Q_{ready})
 `// − Execution Completion −`
34 **ForEach** *core* $p \in P$ **do**
35 **if** *job i finishes on p* **then**
36 release_resources(i, p, M);
37 logJob(i);
38 return(i.results)

Having established the scheduling framework, we now evaluate *SwiftSNNI* in a real-world deployment.

5 Experimental Methodology

This section outlines the experimental setup used to evaluate *SwiftSNNI*, covering the testbed configuration, benchmark neural networks, request generation process, baseline system, and key performance metrics.

Testbed Environment. Experiments were conducted on two identical compute nodes: one acting as the SNNI server and the other as the client host. Each node was equipped with an AMD EPYC 9654 (Genoa) processor with 48 physical cores (96 threads) and 256 GiB RAM . The nodes were interconnected via 25 Gb/s Ethernet and 400 Gb/s InfiniBand. The network benchmarks between the two instances reported an average RTT of 0.4 ms, TCP throughput of 23.7 Gb/s, and UDP jitter below 0.002 ms without packet loss. All secure inference components were implemented in C++.

Benchmarks. We used five pre-trained neural networks of varying sizes (M1, M2, HiNet, AlexNet, VGG-16) as benchmark workloads; their characteristics are summarized in Table 1.

Request Workloads. Two workloads of 500 requests each were evaluated:

- *Workload 1 (Mixed):* 200 requests for M1, 100 requests each for M2 and HiNet, 75 requests for AlexNet, and 25 requests for VGG-16, distributed to reflect a balanced mix of lightweight and heavy neural networks.
- *Workload 2 (Light-Dominant):* 330 requests for M1, 55 for M2, 75 for HiNet, 35 for AlexNet, and 5 for VGG-16. This emphasizes smaller neural networks to reflect practical, high-frequency usage patterns.

Request Generation Scenarios. MNIST-10, CIFAR-10, and ImageNet-1000 lack arrival timestamps. We synthetically generate client requests using a Poisson distribution with different rates (λ), as commonly used to simulate stochastic arrival patterns consistent with prior works [18, 19, 39, 55, 94, 113]. Inter-arrival times are exponentially distributed to reflect memoryless client behavior. Let λ denote the Poisson rate parameter (expected number of requests per unit time over the time interval $[0, \tau]$),

$$\lambda_\tau = \frac{|\mathcal{R}_\tau|}{\tau}, where$$

- $\mathcal{R}_\tau$ is the total number of requests received within $[0, \tau]$ interval.
- τ is the total duration of the observation interval.
- λ is the average arrival rate over that interval.

Baseline: Multi-threaded Sequential SHARK (MS-SHARK). As a primary comparison baseline, we use the original SHARK [40] implementation. This configuration follows an inter-request sequential execution model, where requests are processed one by one in their arrival order. Following its native design, the baseline (MS-SHARK) [40] utilizes OpenMP [79] for multithreading and Eigen [38] for optimized matrix operations to fully parallelize the modular arithmetic of a single secure inference (intra-request parallelism). While the baseline (MS-SHARK) [40] employs single-threaded preprocessing with a fixed batch size of one ($B = 1$), the T_{pre} phase is parallelized across all P available CPU cores. In this setup, job-to-core management is handled exclusively by the default OS-level scheduler without an orchestration layer or a specialized scheduler. This configuration represents the standard, high-performance deployment of SHARK on server-grade hardware. This baseline serves to isolate and measure the benefit of *SwiftSNNI*'s inter-request concurrency and overlapping of offline preprocessing/online inference phases over a parallelized sequential approach that suffers from resource under-utilization during offline preprocessing and head-of-line blocking.

Key Metrics. Performance is evaluated using:

- *Asymptotic average input delay (D):* As defined in Section 3, D is the average delay per input from arrival to completion, which includes both T_{pre} and T_{on} phases.
- *Throughput (ρ):* The total number of inputs processed per unit of time:
$$\rho = \frac{|\mathcal{I}_\tau|}{\tau_{total}}$$
- *Average Waiting Time (W):* This measures the time an input spends waiting before the $T_{\text{on, i}}$ job begins. It includes queuing and $T_{\text{pre, i}}$ delays, but excludes the time spent on $T_{\text{on, i}}$ job itself.
$$W = \frac{1}{|\mathcal{I}_\tau|} \sum_{i \in \mathcal{I}_\tau} \left(t^{\text{on, i}}_{\text{start}} - t_{\text{arrival, i}}\right), where$$

Table 1: Architectural specifications and profiled resource requirements of benchmark neural networks. Conv: Convolutional, FC: Fully Connected, ReLU: Rectified Linear Unit.

Benchmarks	Dataset	Specifications				Layers			
		Input size	Parameter memory (MB)	Activation memory (MB)	FLOPS	Conv	FC	ReLU	Maxpool
M1 [67]	MNIST-10 [118]	28×28, grayscale	1.6	≈ 2	30 M	2	1	2	2
M2 [67]	CIFAR-10 [29]	32×32, RGB	4	≈ 6	50 M	4	1	4	2
HiNet [20]		32×32, RGB	2.4	≈ 4	40 M	2	1	2	1
AlexNet [62]	ImageNet-1000 [24]	224×224 RGB	240	≈ 20	720 M	5	3	5	3
VGG-16 [97]		224×224 RGB	552	≈ 40	15.3 B	13	3	13	5

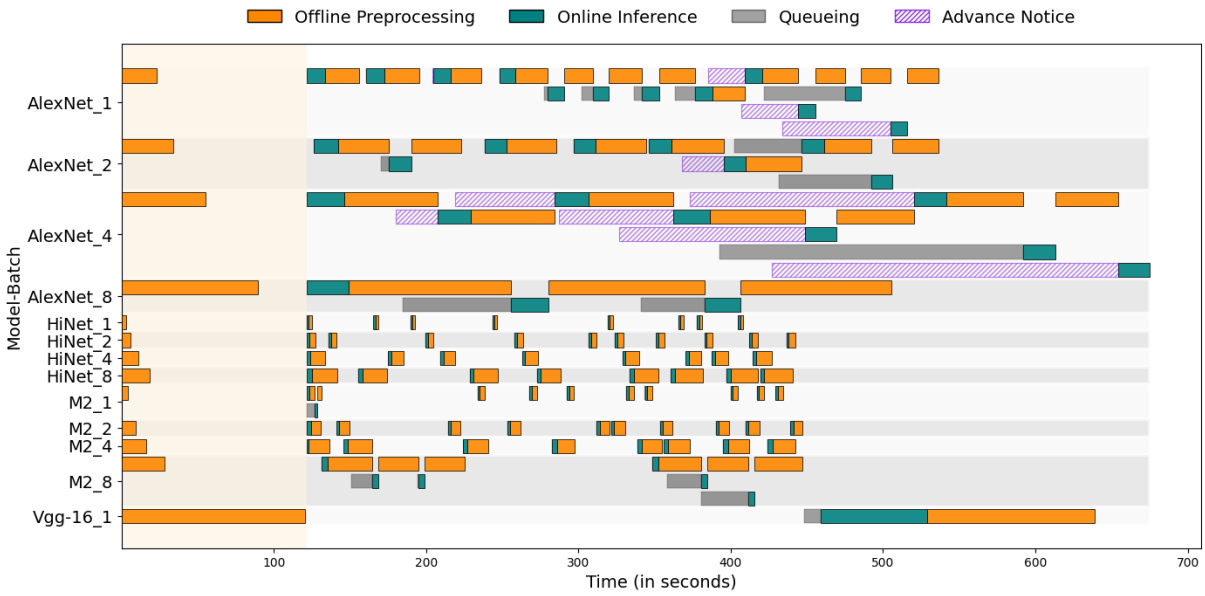


Figure 5: SwiftSNNI Scheduler: Execution Timeline of 100 requests across benchmark neural networks.

- $\mathcal{I}_\tau$ is the set of all inputs in the requests received during the interval $[0, \tau]$,
- $t_{\text{arrival, i}}$ is the arrival time of input i, and
- $t_{\text{start}}^{\text{on, i}}$ is the time the $T_{\text{on, i}}$ begins for input i.

- *Makespan:* This is the total time to complete all 500 requests in a workload.
- *Speedup:* This is the ratio of the MS-SHARK [40] makespan to the *SwiftSNNI* makespan.
- *Advance Notice Impact (ΔD):* This measures the reduction in D when using advance notices (a_i) to compare scenarios with and without advance notices.

6 SwiftSNNI Evaluation

This section evaluates *SwiftSNNI* under varying batch sizes, thread counts, and request arrival rate intensities to assess scalability and responsiveness. The experiments use two workloads of 500 requests each described in Section 5. Request arrivals follow a Poisson process. We use arrival rates (λ) of 1.0, 0.5, and 0.25 req/s to represent high, moderate, and low intensities. These λ rates correspond to average inter-arrival times of 1 s, 2 s, and 4 s respectively. Compared to the MS-SHARK [40]) baseline and the key metrics defined in Section 5, we report the asymptotic average input delay (D), the average waiting time (W), throughput (ρ), makespan, and speedup. To evaluate the benefit of proactive scheduling, a fixed 25% of requests are announced Δt time units in advance via advance notices (a_i) to enable early $T_{\text{pre, i}}$.

Batch Size & Thread Scaling. Empirical results show that batch size exhibits a non-linear effect: moderate batches (2–8) yield the best balance between throughput (ρ) and average input delay (D). Larger batches ($\geq$16) increase the average waiting time (W) for smaller requests. Thread scaling improves throughput (ρ) up to 16 threads. Beyond this, cache contention and synchronization overhead limit further gains. These results justify the *Execution Parameters* chosen for *SwiftSNNI* to prevent large batch requests from monopolizing CPU cores, thereby maintaining parallel efficiency. Both lightweight (M1, HiNet, M2) and heavy neural networks (AlexNet, VGG16) follow this trend.

Impact of Request Intensity. As request intensity increases, queuing delays raise both average input delay (D) and average waiting time (W). In Workload 1, higher arrival rates increase W because the T_{pre} phase is the system's limiting factor. At high intensities (λ = 1), the server cannot generate preprocessed files (T_{pre}) fast enough to keep up with incoming arrivals. *SwiftSNNI* maintains high throughput (ρ) and CPU utilization by overlapping the $T_{\text{pre, i}}$ jobs of queued requests with the $T_{\text{on, j}}$ jobs of active requests. Figure 5 shows that burst requests cause transient overloads that increase W. However, *SwiftSNNI*'s priority-based scoring (p_i) prevents the *Head-of-Line blocking* observed in MS-SHARK [40]. *SwiftSNNI* minimizes queue buildup by dynamically allocating cores to pending $T_{\text{pre, i}}$ jobs.

Scheduling Policy Analysis & Accuracy. The *Resource Monitor* relies on the profiling table (Π) to estimate the execution length ℓ_i for each job. During experiments, these estimates were highly accurate. The estimated values for both $T_{\text{pre, i}}$ and $T_{\text{on, i}}$ jobs matched the actual runtimes within a 5% margin. This precision allowed *SwiftSNNI* to enforce resource constraints effectively. Furthermore, we conducted a partial ablation study by varying the advance notice levels. Without proactive $T_{\text{pre, i}}$ job execution, the average input delay (D) increased across all neural networks. We also observed that disabling the priority-based scoring (p_i) caused heavy neural networks like VGG-16 to suffer from starvation. Without the

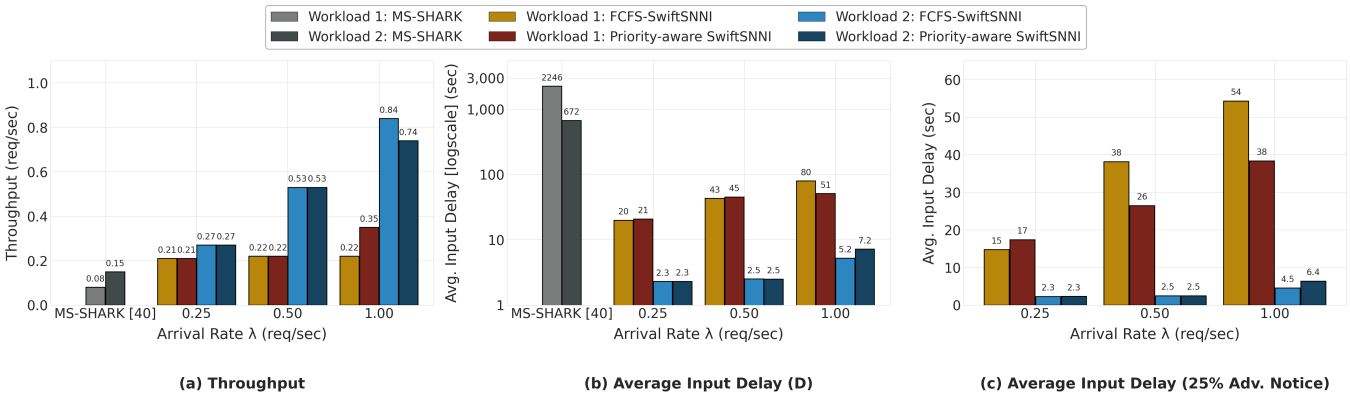


Figure 6: Evaluation of key performance metrics for *Workload 1 (Mixed)* and *Workload 2 (Light-Dominant)*.

concurrent orchestration of the SISE (refer to 4.1.2), *SwiftSNNI* performance reverted to the MS-SHARK [40] baseline.

We evaluate *SwiftSNNI* using two internal scheduling policies: *First-Come-First-Served (FCFS)* and *Priority-aware scoring*. *SwiftSNNI* results show that the best scheduling policy depends on the workload type. *Priority-aware SwiftSNNI* is superior for the mixed requests in Workload 1. However, *FCFS-SwiftSNNI* performs better for the light-dominant requests in Workload 2. *Priority-aware SwiftSNNI* often clusters similar jobs together based on their short execution lengths. Due to *SwiftSNNI*'s model-batch exclusivity constraint, this clustering causes extra average waiting time (W). Multiple jobs compete for the same preprocessed file $T_{\text{pre, i}}$. *FCFS-SwiftSNNI* avoids this by interleaving different requests according to arrival order. This reduces resource contention and improves efficiency for lightweight workloads.

Aggregate Performance. Figure 6 and Table 2 summarize the system-wide results. Across all configurations, *SwiftSNNI* consistently outperforms the MS-SHARK [40] baseline. Advance notices (25%) reduce average input delay (D) by up to *25.32s* in Workload 1 and *0.78s* in Workload 2. This confirms that proactive $T_{\text{pre, i}}$ jobs execution effectively hides the time cost of the T_{pre} phase. These performance gains exceed what could be achieved through static containerization. A container-per-model setup creates fixed resource silos that lead to fragmentation. In contrast, *SwiftSNNI* dynamically reallocates cores to eliminate the idle intervals identified in *SwiftSNNI*'s motivating example (refer to 2.2). This global resource pooling is the primary driver of the $\approx 5.6\times$ throughput (ρ) increase. Furthermore, *SwiftSNNI* achieves over 99% reduction in average waiting time (W) at high request intensities. This demonstrates

Table 2: Performance comparison of *SwiftSNNI* (First-Come-First-Served (FCFS) vs. Priority-aware) against the MS-SHARK baseline [40], (marked with $*$ is shown for reference); best results are in bold.

Workload	Scheduling Policy	Arrival Rate λ (req/sec)	Avg. Input Delay D (s)			Throughput ρ (req/sec)	Speedup (x times)	Avg. Waiting Time W (sec)	MakeSpan (sec)
			No Advance Notice	25% Advance Notice	ΔD (sec)				
1 (Mixed)	*MS-SHARK [40]	-	2246.09	-	-	0.08	-	2233.21	6440.53
	FCFS-*SwiftSNNI*	0.25	**19.88**	**14.74**	5.14	0.21	2.71	**15.94**	2367.76
		0.50	43.08	38.15	4.93	0.22	2.85	38.99	2247.47
		1.00	79.60	54.28	**25.32**	0.22	2.85	75.39	2254.55
	Priority-aware *SwiftSNNI*	0.25	20.87	17.38	3.49	0.21	2.67	16.89	2407.76
		0.50	45.14	26.46	18.68	0.22	2.77	41.07	2319.72
		1.00	50.86	38.34	12.52	**0.35**	**4.56**	46.43	**1410.84**
2 (Light-Dominant)	*MS-SHARK [40]	-	671.89	-	-	0.15	-	665.38	3257.26
	FCFS-*SwiftSNNI*	0.25	**2.30**	**2.29**	0.01	0.27	1.73	0.03	1877.84
		0.50	2.51	2.49	0.02	0.53	3.45	0.17	941.51
		1.00	5.24	4.53	0.716	**0.84**	**5.46**	2.26	**596.01**
	Priority-aware *SwiftSNNI*	0.25	2.31	2.31	0.00	0.27	1.73	0.03	1877.86
		0.50	2.50	2.50	0.00	0.53	3.45	0.17	939.82
		1.00	7.18	6.40	**0.78**	0.74	4.79	**4.30**	677.44

the effectiveness of *Priority-aware SwiftSNNI* scoring policy in preventing starvation. Overall, these results confirm that *SwiftSNNI* provides high concurrency and efficient CPU utilization under stochastic workloads.

7 Related Work

This section reviews existing techniques for inference optimization.

Model Optimization and Parallelism. Neural network compression methods, such as quantization [51], reduce neural network size but may degrade accuracy. Data parallelism [77] and model parallelism [96] exploit device-level computational resources without modifying neural network weights. *SwiftSNNI*'s scheduling is orthogonal to these methods and can be combined with compression or parallelization techniques [47] to further improve efficiency.

Scheduling. Classical offline scheduling from operations research optimizes job assignments in manufacturing [106], healthcare [82], and operations [30]. Online scheduling addresses stochastic or real-time scenarios [64, 81]. Traditional strategies, including gang scheduling [80] and multi-resource allocation [2], focus on independent jobs. However, these methods do not meet the two-phase dependencies of SNNI or the rapid decision requirements of SNNI.

Secure Neural Network Inference (SNNI). SoTA SNNI frameworks [26, 40, 48, 105] ensure cryptographic correctness and efficiency, but provide limited system-level optimization and process requests sequentially or rely on default OS-level scheduling, which limits throughput under concurrency.

Several optimizations improve homomorphic CNN inference: CryptoNets [35] packs multiple inputs per ciphertext for higher throughput but increases memory usage; LoLa [13] encrypts one input per ciphertext to reduce latency, while both rely on batch processing, which can delay individual inferences [50]; Falcon [105] uses spectral inference and input replication to further reduce latency and computation, outperforming traditional vectorized methods [21, 54] for multi-input workloads.

Preprocessing-based 2PC protocols such as MiniONN [67], GAZELLE [54], EzPC [15], and GALA [114], among others [4, 5, 17, 33, 34, 43, 49, 53, 65, 68, 70, 73, 78, 95, 102, 115], demonstrate efficient online inference. SecureML [76] and CrypTFlow2 [86] also show practical applicability [1, 42, 52, 69, 84, 88, 91, 103, 110]. However, none explicitly manage offline preprocessing (T_{pre}) and online inference (T_{on}) as separate, schedulable jobs, which leads to inefficient system resource usage under concurrent requests.

Beyond 2PC, three-party computation (3PC) frameworks include ABY3 [74], BLAZE [85], and SWIFT [60], with additional approaches [3, 26–28, 99, 104, 105, 107]. Secure inference has also progressed to 4PC [14, 16, 45, 61], 5PC [59], and general nPC protocols [12, 31, 58, 90, 116, 117]. These frameworks improve protocol-level performance and security, but leave scheduling and concurrency unaddressed. Function Secret Sharing (FSS) [9–11] further improves 2PC preprocessing [8, 42, 91, 100] and underpins actively secure 3PC frameworks [41, 52, 103], but assumes static workloads.

Prior work ensures strong cryptographic guarantees and preprocessing optimizations, yet practical SNNI deployment still faces concurrent scheduling and resource management challenges that *SwiftSNNI* explicitly addresses.

8 Conclusion

This work presented *SwiftSNNI*, the first framework to formulate Secure Neural Network Inference (SNNI) as a system-level scheduling problem. Unlike existing SNNI frameworks that process requests sequentially, *SwiftSNNI* explicitly coordinates offline preprocessing and online inference jobs. This enables concurrent, resource-aware execution across multiple clients and heterogeneous neural networks.

Comprehensive evaluations on five benchmark neural networks show that *SwiftSNNI* achieves a 97% lower average input delay and an 81% shorter makespan ($\approx 5.4\times$ speedup) compared to the parallelized sequential baseline (MS-SHARK). *SwiftSNNI* delivers a 5.6× increase in throughput in throughput and reduces average waiting time (W) by over 99%. These results confirm that *Priority-aware SwiftSNNI* scoring policy effectively prevents starvation and maximizes resource utilization. These findings demonstrate that explicit inter-request scheduling is essential for scalable and practical SNNI deployments.

Two main limitations remain: large preprocessing file sizes increase memory pressure and I/O overhead, and very large-batch requests can still impact parallel efficiency. Future work will explore dynamic concurrency tuning, memory-aware scheduling, adaptive handling of large-batch requests, and multi-server load balancing. We also plan to investigate reinforcement learning–based schedulers to further improve scalability, responsiveness and resource utilization in dynamic cloud environments.

9 Acknowledgment

This work has received funding from the European Union's Horizon Europe research and innovation programme under the LICORICE project (Grant Agreement No. 101168311).

References

[1] Nitin Agrawal, Ali Shahin Shamsabadi, Matt J Kusner, and Adrià Gascón. 2019. QUOTIENT: Two-party secure neural network training and prediction. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*. 1231–1247.

[2] James H. Anderson and John M. Calandrino. 2006. Parallel Real-Time Task Scheduling on Multicore Platforms. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. 89–100. doi:10.1109/RTSS.2006.32

[3] Nuttapong Attrapadung, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Takahiro Matsuda, Ibuki Mishina, Hiraku Morita, and Jacob C. N. Schuldt. 2021. Adam in Private: Secure and Fast Training of Deep Neural Networks with Adaptive Moment Estimation. *CoRR* abs/2106.02203 (2021). arXiv:2106.02203 https://arxiv.org/abs/2106.02203

[4] Marshall Ball, Brent Carmer, Tal Malkin, Mike Rosulek, and Nichole Schimanski. 2019. Garbled Neural Networks are Practical. *Cryptology ePrint Archive* (2019).

[5] Shashank Balla and Farinaz Koushanfar. 2023. HELiKs: HE Linear Algebra Kernels for Secure Inference. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 2306–2320. doi:10.1145/3576915.3623136

[6] Donald Beaver. 1992. Efficient Multiparty Protocols Using Circuit Randomization. In *Advances in Cryptology — CRYPTO '91*, Joan Feigenbaum (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 420–432.

[7] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 1988. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing* (Chicago, Illinois, USA) *(STOC '88)*. Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/62212.62213

[8] Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. 2021. Function Secret Sharing for Mixed-Mode and Fixed-Point Secure Computation. In *Advances in Cryptology – EUROCRYPT 2021: 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17–21, 2021, Proceedings, Part*

II (Zagreb, Croatia). Springer-Verlag, Berlin, Heidelberg, 871–900. doi:10.1007/978-3-030-77886-6_30

[9] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2015. Function Secret Sharing. In *Advances in Cryptology - EUROCRYPT 2015*, Elisabeth Oswald and Marc Fischlin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–367.

[10] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2016. Function Secret Sharing: Improvements and Extensions. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1292–1303. doi:10.1145/2976749.2978429

[11] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2019. Secure Computation with Preprocessing via Function Secret Sharing. In *Theory of Cryptography: 17th International Conference, TCC 2019, Nuremberg, Germany, December 1–5, 2019, Proceedings, Part I* (Nuremberg, Germany). Springer-Verlag, Berlin, Heidelberg, 341–371. doi:10.1007/978-3-030-36030-6_14

[12] Lennart Braun, Daniel Demmler, Thomas Schneider, and Oleksandr Tkachenko. 2022. MOTION – A Framework for Mixed-Protocol Multi-Party Computation. *ACM Trans. Priv. Secur.* 25, 2, Article 8 (March 2022), 35 pages. doi:10.1145/3490390

[13] Alon Brutzkus, Oren Elisha, and Ran Gilad-Bachrach. 2018. Low Latency Privacy Preserving Inference. *CoRR* abs/1812.10659. arXiv:1812.10659 http://arxiv.org/abs/1812.10659

[14] Megha Byali, Harsh Chaudhari, Arpita Patra, and Ajith Suresh. 2019. FLASH: Fast and Robust Framework for Privacy-preserving Machine Learning. Cryptology ePrint Archive, Paper 2019/1365. https://eprint.iacr.org/2019/1365

[15] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. 2019. EzPC: Programmable and Efficient Secure Two-Party Computation for Machine Learning. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. 496–511. doi:10.1109/EuroSP.2019.00043

[16] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. 2019. Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning. Cryptology ePrint Archive, Paper 2019/1315. doi:10.14722/ndss.2020.23005

[17] Ke Cheng, Jiaxuan Fu, Yulong Shen, Haichang Gao, Ning Xi, Zhiwei Zhang, and Xinghui Zhu. 2023. Manto: A Practical and Secure Inference Service of Convolutional Neural Networks for IoT. *IEEE Internet of Things Journal* 10, 16 (2023), 14856–14872. doi:10.1109/JIOT.2023.3251982

[18] Yujeong Choi, Yunseong Kim, and Minsoo Rhu. 2021. Lazy Batching: An SLA-aware Batching System for Cloud Machine Learning Inference . In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 493–506. doi:10.1109/HPCA51647.2021.00049

[19] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: a Low-Latency Online Prediction Serving System. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'17)*. USENIX Association, USA, 613–627.

[20] CSIRO ML & AI. 2023. Deep Learning Training with Multi-Party Computation. https://github.com/csiro-mlai/deep-mpc/tree/more-models. Accessed: 2025-11-04.

[21] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. 2020. EVA: an encrypted vector arithmetic language and compiler for efficient homomorphic computation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 546–561. doi:10.1145/3385412.3386023

[22] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. 2019. CHET: An Optimizing Compiler for Fully-Homomorphic Neural-Network Inferencing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 142–156. doi:10.1145/3314221.3314628

[23] Richard A. DeMillo, Richard J. Lipton, David P. Dobkin, and Anita K. Jones. 1978. *Foundations of Secure Computation.* Academic Press, Inc., USA.

[24] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[25] Jack Doerner and Abhi Shelat. 2017. Scaling ORAM for Secure Computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 523–535. doi:10.1145/3133956.3133967

[26] Ye Dong, Xiaojun Chen, Weizhan Jing, Kaiyun Li, and Weiping Wang. 2023. Meteor: Improved Secure 3-Party Neural Network Inference with Reducing Online Communication Costs. Cryptology ePrint Archive, Paper 2023/100. https://eprint.iacr.org/2023/100

[27] Ye Dong, Xiaojun Chen, Xiangfu Song, and Kaiyun Li. 2023. FlexBNN: Fast Private Binary Neural Network Inference With Flexible Bit-Width. *IEEE Transactions on Information Forensics and Security* 18 (2023), 2382–2397. doi:10.1109/TIFS.2023.3265342

[28] Ye Dong, Wen-jie Lu, Yancheng Zheng, Haoqi Wu, Derun Zhao, Jin Tan, Zhicong Huang, Cheng Hong, Tao Wei, Wenguang Chen, and Jianying Zhou. 2025. PUMA: Secure Inference of LLaMA-7B in Five Minutes. *Security and Safety* (Oct. 2025). doi:10.1051/sands/2025014

[29] Raveen Doon, Tarun Kumar Rawat, and Shweta Gautam. 2018. Cifar-10 Classification using Deep Convolutional Neural Network. In *2018 IEEE Punecon*. 1–5. doi:10.1109/PUNECON.2018.8745428

[30] S. Ayca Erdogan, Alexander Gose, and Brian T. Denton. 2015. Online appointment sequencing and scheduling. *IIE Transactions* 47, 11 (2015), 1267–1286. arXiv:https://doi.org/10.1080/0740817X.2015.1011355 doi:10.1080/0740817X.2015.1011355

[31] Xiaoyu Fan, Kun Chen, Guosai Wang, Mingchun Zhuang, Yi Li, and Wei Xu. 2022. NFGen: Automatic Non-linear Function Evaluation Code Generator for General-purpose MPC Platforms. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 995–1008. doi:10.1145/3548606.3560565

[32] Lars Folkerts, Charles Gouert, and Nektarios Georgios Tsoutsos. 2021. REDsec: Running Encrypted Discretized Neural Networks in Seconds. Cryptology ePrint Archive, Paper 2021/1100. https://eprint.iacr.org/2021/1100

[33] Zahra Ghodsi, Nandan Kumar Jha, Brandon Reagen, and Siddharth Garg. 2021. Circa: Stochastic ReLUs for Private Deep Learning. arXiv:2106.08475 [cs.LG] https://arxiv.org/abs/2106.08475

[34] Zahra Ghodsi, Akshaj Kumar Veldanda, Brandon Reagen, and Siddharth Garg. 2020. CryptoNAS: Private Inference on a ReLU Budget. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 1423, 11 pages.

[35] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *Proceedings of The 33rd International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 48)*, Maria Florina Balcan and Kilian Q. Weinberger (Eds.). PMLR, New York, New York, USA, 201–210. https://proceedings.mlr.press/v48/gilad-bachrach16.html

[36] O. Goldreich, S. Micali, and A. Wigderson. 1987. How to play any mental game, or a completeness theorem for protocols with an honest majority. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing* (New York, New York, USA) *(STOC '87)*. Association for Computing Machinery, New York, NY, USA, 218–229. doi:10.1145/28395.28420

[37] Ioannis Grigoriadis, Eleni Vrochidou, Iliana Tsiatsiou, editor="Saraswat Mukesh Papakostas, George A.", Chandreyee Chowdhury, Chintan Kumar Mandal, and Amir H. Gandomi. 2023. Machine Learning as a Service (MLaaS)—An Enterprise Perspective. In *Proceedings of International Conference on Data Science and Applications*. Springer Nature Singapore, Singapore, 261–273.

[38] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen: C++ template library for linear algebra. https://libeigen.gitlab.io/ Accessed: February 15, 2026.

[39] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S. McKinley, and Björn B. Brandenburg. 2017. Swayam: Distributed Autoscaling to Meet SLAs of Machine Learning Inference Services with Resource Efficiency. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference* (Las Vegas, Nevada) *(Middleware '17)*. Association for Computing Machinery, New York, NY, USA, 109–120. doi:10.1145/3135974.3135993

[40] Kanav Gupta, Nishanth Chandran, Divya Gupta, Jonathan Katz, and Rahul Sharma. 2025. SHARK: Actively Secure Inference Using Function Secret Sharing . In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 2472–2490. doi:10.1109/SP61157.2025.00175

[41] Kanav Gupta, Neha Jawalkar, Ananta Mukherjee, Nishanth Chandran, Divya Gupta, Ashish Panwar, and Rahul Sharma. 2024. SIGMA: Secure GPT Inference with Function Secret Sharing. In *Privacy Enhancing technologies Symposium (PETS) 2024*. https://www.microsoft.com/en-us/research/publication/sigma-secure-gpt-inference-with-function-secret-sharing/

[42] Kanav Gupta, Deepak Kumaraswamy, Nishanth Chandran, and Divya Gupta. 2022. LLAMA: A Low Latency Math Library for Secure Inference. *Proceedings on Privacy Enhancing Technologies* 2022, 4 (2022), 274–294.

[43] Meng Hao, Hongwei Li, Hanxiao Chen, Pengzhi Xing, Guowen Xu, and Tianwei Zhang. 2022. Iron: Private Inference on Transformers. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 15718–15731. https://proceedings.neurips.cc/paper_files/paper/2022/file/64e2449d74f84e5b1a5c96ba7b3d308e-Paper-Conference.pdf

[44] Christopher Harth-Kitzerow, Yongqin Wang, Rachit Rajat, Georg Carle, and Murali Annavaram. 2025. PIGEON: A High Throughput Framework for Private Inference of Neural Networks using Secure Multiparty Computation. *Proceedings on Privacy Enhancing Technologies* (2025).

[45] Aditya Hegde, Nishat Koti, Varsha Bhat Kukkala, Shravani Patil, Arpita Patra, and Protik Paul. 2022. Attaining GOD Beyond Honest Majority with Friends and Foes. In *Advances in Cryptology – ASIACRYPT 2022: 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei,*

Taiwan, December 5–9, 2022, Proceedings, Part I (Taipei, Taiwan). Springer-Verlag, Berlin, Heidelberg, 556–587. doi:10.1007/978-3-031-22963-3_19

[46] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. 2021. Characterization and Prediction of Deep Learning Workloads in Large-Scale GPU Datacenters . In *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, Los Alamitos, CA, USA, 1–15. https://doi.ieeecomputersociety.org/

[47] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Curran Associates Inc., Red Hook, NY, USA, Article 10, 10 pages.

[48] Zhicong Huang, Wen jie Lu, Cheng Hong, and Jiansheng Ding. 2022. Cheetah: Lean and Fast Secure Two-Party Deep Neural Network Inference. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 809–826. https://www.usenix.org/conference/usenixsecurity22/presentation/huang-zhicong

[49] Siam Umar Hussain, Mojan Javaheripi, Mohammad Samragh, and Farinaz Koushanfar. 2021. COINN: Crypto/ML Codesign for Oblivious Inference via Neural Networks. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 3266–3281. doi:10.1145/3460120.3484797

[50] Yoshiaki Inoue. 2021. Queueing Analysis of GPU-based Inference Servers with Dynamic Batching: A Closed-Form Characterization. *Perform. Eval.* 147, C (May 2021), 17 pages. doi:10.1016/j.peva.2020.102183

[51] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2704–2713. doi:10.1109/CVPR.2018.00286

[52] Neha Jawalkar, Kanav Gupta, Arkaprava Basu, Nishanth Chandran, Divya Gupta, and Rahul Sharma. 2024. Orca: FSS-based Secure Training and Inference with GPUs . In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 597–616. doi:10.1109/SP54263.2024.00063

[53] Nandan Kumar Jha, Zahra Ghodsi, Siddharth Garg, and Brandon Reagen. 2021. DeepReDuce: ReLU Reduction for Fast Private Inference. 4839–4849 pages.

[54] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 1651–1669. https://www.usenix.org/conference/usenixsecurity18/presentation/juvekar

[55] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. 2019. GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 34, 16 pages. doi:10.1145/3302424.3303958

[56] Marcel Keller. 2020. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 1575–1590. doi:10.1145/3372297.3417872

[57] Donghwan Kim, Jaiyoung Park, Jongmin Kim, Sangpyo Kim, and Jung Ho Ahn. 2024. HyPHEN: A Hybrid Packing Method and Its Optimizations for Homomorphic Encryption-Based Neural Networks. *IEEE Access* 12 (2024), 3024–3038. doi:10.1109/ACCESS.2023.3348170

[58] Brian Knott, Shobha Venkataraman, Awni Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. 2021. CRYPTEN: Secure Multi-party Computation meets Machine Learning. In *Proceedings of the 35th International Conference on Neural Information Processing Systems (NIPS '21)*. Curran Associates Inc., Red Hook, NY, USA, Article 379, 13 pages.

[59] Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, and Bhavish Raj Gopal. 2022. PentaGOD: Stepping beyond Traditional GOD with Five Parties. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1843–1856.

[60] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. 2021. SWIFT: Super-fast and Robust Privacy-Preserving Machine Learning. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2651–2668. https://www.usenix.org/conference/usenixsecurity21/presentation/koti

[61] Nishat Koti, Arpita Patra, Rahul Rachuri, and Ajith Suresh. 2021. Tetrad: Actively Secure 4PC for Secure Training and Inference. *CoRR* abs/2106.02850 (2021). arXiv:2106.02850 https://arxiv.org/abs/2106.02850

[62] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger (Eds.), Vol. 25. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[63] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. 2020. CrypTFlow: Secure TensorFlow Inference. In *IEEE Symposium on Security and Privacy (SP)*. 336–353.

[64] Kangbok Lee, Feifeng Zheng, and Michael L. Pinedo. 2019. Online scheduling of ordered flow shops. *European Journal of Operational Research* 272, 1 (None 2019), 50–60. doi:10.1016/j.ejor.2018.06.008

[65] Ryan Lehmkuhl, Pratyush Mishra, Akshayaram Srinivasan, and Raluca Ada Popa. 2021. Muse: Secure Inference Resilient to Malicious Clients. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2201–2218. https://www.usenix.org/conference/usenixsecurity21/presentation/lehmkuhl

[66] Ji Lin, Yongming Rao, Jiwen Lu, and Jie Zhou. 2017. Runtime Neural Pruning. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/a51fb975227d6640e4fe47854476d133-Paper.pdf

[67] Jian Liu, Mika Juuti, Yao Lu, and Nadarajah Asokan. 2017. Oblivious Neural Network Predictions via MiniONN Transformations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 619–631. doi:10.1145/3133956.3134056

[68] Xuanqi Liu, Zhuotao Liu, Qi Li, Ke Xu, and Mingwei Xu. 2024. Pencil: Private and Extensible Collaborative Learning without the Non-Colluding Assumption. *arXiv preprint arXiv:2403.11166* (2024). arXiv:2403.11166 [cs.CR] https://arxiv.org/abs/2403.11166

[69] Xiaoning Liu, Yifeng Zheng, Xingliang Yuan, and Xun Yi. 2023. Securely Outsourcing Neural Network Inference to the Cloud With Lightweight Techniques. *IEEE Transactions on Dependable and Secure Computing* 20, 01 (Jan. 2023), 620–636. doi:10.1109/TDSC.2022.3141391

[70] Qian Lou, Yilin Shen, Hongxia Jin, and Lei Jiang. 2021. SAFENet: A Secure, Accurate and Fast Neural Network Inference. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=Cz3dbFm5u-

[71] Junming Ma, Yancheng Zheng, Jun Feng, Derun Zhao, Haoqi Wu, Wenjing Fang, Jin Tan, Chaofan Yu, Benyu Zhang, and Lei Wang. 2023. SecretFlow-SPU: A Performant and User-Friendly Framework for Privacy-Preserving Machine Learning. In *2023 USeNIX annual technical conference (USeNIX ATC 23)*. USENIX Association, Boston, MA, 17–33. https://www.usenix.org/conference/atc23/presentation/ma

[72] Zoltán Ádám Mann, Christian Weinert, Daphnee Chabal, and Joppe W Bos. 2023. Towards Practical Secure Neural Network Inference: The Journey So Far and the Road Ahead. *Comput. Surveys* 56, 5, Article 117 (Nov. 2023), 37 pages. doi:10.1145/3628446

[73] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. 2020. Delphi: A Cryptographic Inference Service for Neural Networks. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2505–2522. https://www.usenix.org/conference/usenixsecurity20/presentation/mishra

[74] Payman Mohassel and Peter Rindal. 2018. ABY3: A Mixed Protocol Framework for Machine Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 35–52. doi:10.1145/3243734.3243760

[75] Payman Mohassel, Mike Rosulek, and Ye Zhang. 2015. Fast and Secure Three-party Computation: The Garbled Circuit Approach. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 591–602. doi:10.1145/2810103.2813705

[76] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 19–38. doi:10.1109/SP.2017.12

[77] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. arXiv:2104.04473 [cs.CL] https://arxiv.org/abs/2104.04473

[78] Lucien K. L. Ng and Sherman S. M. Chow. 2021. GForce: GPU-Friendly Oblivious and Rapid Neural Network Inference. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2147–2164. https://www.usenix.org/conference/usenixsecurity21/presentation/ng

[79] OpenMP. November 2024. https://www.openmp.org Accessed: February 15, 2026.

[80] John K. Ousterhout. 1982. Scheduling Techniques for Concurrent Systems. In *IEEE International Conference on Distributed Computing Systems*, Vol. 82. 22–30. https://api.semanticscholar.org/CorpusID:210842950

[81] Bowen Pang, Xiaolei Xie, Feng Ju, and James Pipe. 2022. A dynamic sequential decision-making model on MRI real-time scheduling with simulation-based optimization. *Health Care Management Science* 25, 3 (sep 2022), 426–440. doi:10.

1007/s10729-022-09592-6 Publisher Copyright: © 2022, The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature..

[82] Bowen Pang, Xiaolei Xie, Yongjia Song, and Li Luo. 2019. Surgery Scheduling Under Case Cancellation and Surgery Duration Uncertainty. *IEEE Transactions on Automation Science and Engineering* 16, 1 (2019), 74–86. doi:10.1109/TASE.2018.2834486

[83] Qi Pang, Jinhao Zhu, Helen Möllering, Wenting Zheng, and Thomas Schneider. 2024. BOLT: Privacy-Preserving, Accurate and Efficient Inference for Transformers. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 4753–4771. https://eprint.iacr.org/2023/1893

[84] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2021. ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation. In *30th USENIX Security Symposium (USENIX Security 21)*. 2165–2182. https://eprint.iacr.org/2020/1225

[85] Arpita Patra and Ajith Suresh. 2020. BLAZE: Blazing Fast Privacy-Preserving Machine Learning. *Cryptology ePrint Archive, Paper 2020/042* (2020). doi:10.14722/ndss.2020.24202

[86] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. 2020. CrypTFlow2: Practical 2-Party secure inference. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 325–342. doi:10.1145/3372297.3417274

[87] M Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin Lauter, and Farinaz Koushanfar. 2019. XONN: XNOR-based Oblivious Deep Neural Network Inference. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(SEC'19)*. USENIX Association, USA, 1501–1518.

[88] M Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M Songhori, Thomas Schneider, and Farinaz Koushanfar. 2018. Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security* (Incheon, Republic of Korea) *(ASIACCS '18)*. Association for Computing Machinery, New York, NY, USA, 707–721. doi:10.1145/3196494.3196522

[89] Bita Darvish Rouhani, M Sadegh Riazi, and Farinaz Koushanfar. 2017. DeepSecure: Scalable Provably-Secure Deep Learning. arXiv:1705.08963 [cs.CR] https://arxiv.org/abs/1705.08963

[90] Wenqiang Ruan, Mingxin Xu, Wenjing Fang, Li Wang, Lei Wang, and Weili Han. 2023. Private, Efficient, and Accurate: Protecting Models Trained by Multi-party Learning with Differential Privacy. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, IEEE Computer Society, Los Alamitos, CA, USA, 1926–1943. doi:10.1109/SP46215.2023.10179422

[91] Théo Ryffel, Pierre Tholoniat, David Pointcheval, and Francis Bach. 2021. ARIANN: Low-Interaction Privacy-Preserving Deep Learning via Function Secret Sharing. (2021). arXiv:2006.04593 [cs.LG] https://arxiv.org/abs/2006.04593

[92] Amartya Sanyal, Matt Kusner, Adria Gascon, and Varun Kanade. 2018. TAPAS: Tricks to Accelerate (encrypted) Prediction As a Service. In *International conference on machine learning*. PMLR, 4490–4499.

[93] Iqbal H Sarker. 2021. Machine Learning: Algorithms, Real-World Applications and Research Directions. *SN Computer Science* 2, 160 (2021). doi:10.1007/s42979-021-00592-x

[94] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: A GPU cluster engine for accelerating DNN-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 322–337. doi:10.1145/3341301.3359658

[95] Liyan Shen, Ye Dong, Binxing Fang, Jinqiao Shi, Xuebin Wang, Shengli Pan, and Ruisheng Shi. 2022. ABNN^2: Secure Two-Party Arbitrary-Bitwidth Quantized Neural Network Predictions. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 361–366. doi:10.1145/3489517.3530680

[96] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053 [cs.CL] https://arxiv.org/abs/1909.08053

[97] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv:1409.1556 https://arxiv.org/abs/1409.1556

[98] Oliver Sinnen. 2007. *Task Scheduling for Parallel Systems*. John Wiley & Sons, Inc., USA. 269–280 pages. doi:10.1002/0470121173

[99] Lushan Song, Jiaxuan Wang, Zhexuan Wang, Xinyu Tu, Guopeng Lin, Wenqiang Ruan, Haoqi Wu, and Weili Han. 2022. pMPL: A Robust Multi-Party Learning Framework with a Privileged Party. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 2689–2703. doi:10.1145/3548606.3560697

[100] Kyle Storrier, Adithya Vadapalli, Allan Lyons, and Ryan Henry. 2023. Grotto: Screaming fast (2+1)-PC or $\mathbb{Z}2n$ via (2,2)-DPFs. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 2143–2157. doi:10.1145/3576915.3623147

[101] Sijun Tan, Brian Knott, Yuan Tian, and David J Wu. 2021. CryptGPU: Fast Privacy-Preserving Machine Learning on the GPU. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1021–1038. doi:10.1109/SP40001.2021.00098

[102] Han Tian, Chaoliang Zeng, Zhenghang Ren, Di Chai, Junxue Zhang, Kai Chen, and Qiang Yang. 2022. Sphinx: Enabling Privacy-Preserving Online Learning over the Cloud. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2487–2501. doi:10.1109/SP46214.2022.9833648

[103] Sameer Wagh. 2022. Pika: Secure Computation using Function Secret Sharing over Rings. *Proceedings on Privacy Enhancing Technologies* (2022).

[104] Sameer Wagh, Divya Gupta, and Nishanth Chandran. 2019. SecureNN: 3-Party Secure Computation for Neural Network Training. *Proceedings on Privacy Enhancing Technologies* (2019), 26–49. doi:10.2478/popets-2019-0035

[105] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. 2020. FALCON: Honest-Majority Maliciously Secure Framework for Private Deep Learning. *arXiv preprint arXiv:2004.02229* (2020). https://arxiv.org/abs/2004.02229

[106] Feifan Wang, Sepehr Fathizadan, Feng Ju, Kyle Rowe, and Nils Hofmann. 2020. Print Surface Thermal Modeling and Layer Time Control for Large-Scale Additive Manufacturing. *IEEE Transactions on Automation Science and Engineering* 18, 1 (2020), 244–254. doi:10.1109/TASE.2020.3001047

[107] Songlei Wang, Yifeng Zheng, and Xiaohua Jia. 2023. SecGNN: Privacy-Preserving Graph Neural Network Training and Inference as a Cloud Service. *IEEE Transactions on Services Computing* 16, 04 (July 2023), 2923–2938. doi:10.1109/TSC.2023.3241615

[108] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. 2022. Piranha: A GPU Platform for Secure Computation. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 827–844. https://www.usenix.org/conference/usenixsecurity22/presentation/watson

[109] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 945–960. https://www.usenix.org/conference/nsdi22/presentation/weng

[110] Xuanang Yang, Jing Chen, Kun He, Hao Bai, Cong Wu, and Ruiying Du. 2023. Efficient Privacy-Preserving Inference Outsourcing for Convolutional Neural Networks. *IEEE Transactions on Information Forensics and Security* 18 (2023), 4815–4829. doi:10.1109/TIFS.2023.3287072

[111] Andrew C. Yao. 1982. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science (SFCS '82)*. IEEE Computer Society, USA, 160–164. doi:10.1109/SFCS.1982.38

[112] Andrew Chi-Chih Yao. 1986. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. IEEE, 162–167. doi:10.1109/SFCS.1986.25

[113] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 1049–1062. https://www.usenix.org/conference/atc19/presentation/zhang-chengliang

[114] Qiao Zhang, Chunsheng Xin, and Hongyi Wu. 2021. GALA: Greedy ComputAtion for Linear Algebra in Privacy-Preserved Neural Networks. https://arxiv.org/abs/2105.01827

[115] Mengxin Zheng, Qian Lou, and Lei Jiang. 2025. Primer: Fast Private Transformer Inference on Encrypted Data. In *Proceedings of the 60th Annual ACM/IEEE Design Automation Conference* (San Francisco, California, United States) *(DAC '23)*. IEEE, IEEE Press, 1–6. doi:10.1109/DAC56929.2023.10247719

[116] Wenting Zheng, Ryan Deng, Weikeng Chen, Raluca Ada Popa, Aurojit Panda, and Ion Stoica. 2021. Cerebro: A Platform for Multi-Party Cryptographic Collaborative Learning. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2723–2740. https://www.usenix.org/conference/usenixsecurity21/presentation/zheng

[117] Wenting Zheng, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. 2019. Helen: Maliciously Secure Coopetitive Learning for Linear Models. In *2019 IEEE Symposium on Security and Privacy (SP)*, Vol. 1. IEEE, 724–738. doi:10.1109/SP.2019.00045

[118] Wanlei Zhu. 2018. Classification of MNIST Handwritten Digit Database using Neural Network. *Proceedings of the research school of computer science. Australian National University, Acton, ACT* 2601 (2018). https://api.semanticscholar.org/CorpusID:202741200