ML Model Serving in Mixed-Privacy Scenarios

A Performance Study on AWS Nitro Enclaves

Hannah Pierce-Hoffman
COMPSCI 243 Final Project

***Abstract*—Developing secure cloud deployment strategies for ML model serving is critical for ML applications dealing with sensitive data, models, or both. A robust solution for secure ML model serving must also be able to handle mixed-privacy scenarios, in which inference requests are subject to varied levels of privacy concerns. AWS Nitro Enclaves, which create an isolated execution environment within a parent EC2 instance, offer one potential solution for secure model serving. In this work, we collect a series of system performance benchmarks for an ML server running within an AWS Nitro Enclave, with the intent of quantifying the performance impact of migrating an existing model serving pipeline into an enclave context. We also prototype a mixed-privacy model serving solution which uses an enclave server and a standard Dockerized server running side-by-side on the same EC2 instance. We find that the system performance overhead of migrating model serving into a Nitro Enclave is less than expected. Although the enclave server takes 4.7x as long to start compared to the standard server, and the maximum I/O throughput of the enclave server is 13.1% of the maximum I/O throughput for the standard server, we ultimately find that the enclave server is able to perform comparably to the standard server under typical internet load conditions. Additionally, we find that the maximum latency for the enclave server is only marginally higher than the maximum latency for the standard server when both servers are run side-by-side in a mixed-privacy scenario.**

***Index Terms*—model serving, security, AWS, Nitro Enclaves, mixed-privacy, performance, inference, latency**

I. Introduction

A. Background

Cloud computing now accounts for one-third of all IT expenses, as well as 3% of total global energy consumption [1] [2]. However, the rapid adoption of cloud platforms introduces a fundamental tension between security and convenience. Training or serving machine learning models in the cloud involves trusting models and data to a third party, which increases the attack surface of the application. For users performing computations with sensitive models or data, model serving in the cloud may pose a significant security risk. This is especially true for users working with biomedical, financial, or educational data.

In addition to full-privacy scenarios in which all data or models must be kept confidential, mixed-privacy scenarios are becoming increasingly common. A mixed-privacy scenario occurs when different users, or groups of users, are subject to varied privacy requirements. For example, some users may opt-in to enhanced privacy protections, or users may live in different countries with varying privacy laws. The GDPR, a comprehensive European privacy law passed in 2018, is one example of a location-specific privacy law which has incentivized many businesses to tighten privacy protections for their European users [3]. In order to maximize user experience, ML applications in the cloud must be able to gracefully handle mixed-privacy model serving, with minimal impact to latency and inference time.

Amazon Web Services (AWS) recently introduced Nitro Enclaves, an EC2 feature which has the potential to mitigate security concerns for ML model serving [4]. A Nitro Enclave is a secure computing environment which is created from physically partitioned CPUs and memory inside a parent EC2 instance. The Nitro Enclave has no interactive access, persistent storage, or external networking capability. All communication with the outside world goes through a VSOCK channel, a secure local socket connection to the parent instance. Nitro Enclaves also offer the option for cryptographic attestation, an additional level of verification which allows the enclave to cryptographically prove its identity to another party. With a significantly reduced attack surface, Nitro Enclaves are a promising option for confidential model serving. Since a Nitro Enclave server can operate alongside standard servers on a single parent EC2 instance, Nitro Enclaves also present a potential solution for model serving in mixed-privacy scenarios.

B. Central Questions

During the process of cloud adoption, users may wish to incorporate Nitro Enclaves into a full-privacy or mixed-privacy model serving pipeline. However, little work has been done to assess the performance impact of migrating model serving into a Nitro Enclave. Understanding the performance impacts of Nitro Enclaves on model serving is critical for building efficient model serving pipelines, so that other parts of the pipeline may be optimized to mitigate these impacts. Additionally, no previous work has developed a mixed-privacy model serving system which incorporates Nitro Enclaves. Developing effective solutions for mixed-privacy scenarios is important for any situation in which the security level must be tailored to the user, data, or model, as discussed above. Therefore, in this work we focus on the following two questions:

- How does migrating model serving into a Nitro Enclave affect key performance benchmarks such as deployment time, inference throughput, and latency?
- How can we effectively handle mixed-privacy requests in a single inference stream, using a standard server and an enclave server on a single parent EC2 instance?

C. Challenges

When addressing these questions, we must consider several key challenges relating to enclave development and benchmarking.

- **Limited VSOCK Bandwidth**: The constrained bandwidth of the VSOCK channel between the parent EC2 instance and the enclave presents the most significant barrier to model serving performance. It is necessary to quantify the magnitude of the VSOCK bandwidth constraint in order to avoid bottlenecks in an enclave serving system.
- **Memory Considerations**: Nitro Enclaves are constructed from a base Docker image, but the resulting Enclave Image File (EIF) is larger than the original Docker image. A Nitro Enclave also has different runtime memory requirements than the equivalent Docker container. It is necessary to account for this memory discrepancy when building, running, and storing enclaves, since it can impact the size and number of models which can be loaded on a single EC2 instance.
- **Engineering Complexity**: Deploying Nitro Enclaves as part of a model serving pipeline requires considerable knowledge of containers, serving utilities, AWS configuration, and Linux system files. Lack of comprehensive documentation for Nitro Enclaves presents an additional barrier to effective deployment.
- **Encryption and Attestation**: High-security data is often encrypted, and may only be decrypted within the trusted environment of the Nitro Enclave. Additionally, many existing applications involving Nitro Enclaves utilize the cryptographic attestation functionality to verify the identity of the enclave before running computations. Encryption and decryption often incur significant computational overhead [citation]. Cryptographic attestation has previously been shown to incur minimal computational overhead [citation], but use of attestation increases engineering complexity.

D. Solution

In this work, we build a sample model serving system inside a Nitro Enclave and generate several key benchmarks related to model deployment, inference throughput, and latency. We present these benchmarks as a starting point for users building end-to-end model serving pipelines with security concerns in mind. Additionally, we develop a sample mixed-privacy serving system as a proof of concept, and provide some initial benchmarks on its performance.

To ensure that VSOCK bandwidth limitations are adequately quantified, we measure the maximum throughput of the VSOCK channel compared to maximum throughput of a standard system port. Additionally, we perform load testing on an ML application running inside a Nitro Enclave and an ML application running inside a standard Docker container. To investigate the impact of enclave size on performance, we measure the storage size, RAM usage, and startup time of an enclave server and a standard server built with the same base Docker image. We also implement additional performance tests, including load testing over the internet and testing the mixed-privacy serving system with different proportions of secure and standard requests. For full details of our system architecture and benchmarks, see the Design section below.

To address the engineering complexity of working with Nitro Enclaves, we provide a comprehensive guide to reproducing our system, in an effort to bridge the engineering knowledge gap and synthesize documentation for future users. Please see our Github repo for steps and instructions on reproducing our work.

The final challenge we identified above involves quantifying the performance impact of encryption, decryption, and cryptographic attestation. We consider these aspects of performance benchmarking out-of-scope for this study, and instead focus on the use case in which users are primarily concerned with the raw performance capabilities of the enclave itself. This is relevant for any scenario in which users have already established encryption and key management solutions, but are seeking to migrate these solutions into a more secure enclave environment. In such a scenario, we follow a **software-based trust policy**, which places trust on the software's ability to isolate and protect a model serving system without the need for hardware validation of the encryption process. Users in this scenario trust the enclave's software to provide sufficient isolation, so in our performance benchmarks, we abstract away the additional layers of hardware security which would be provided by encryption or cryptographic attestation.

E. Key Results

Overall, we find that the system performance impact of model serving using Nitro Enclaves is lower than expected. Although enclaves consume more storage and startup resources, a running enclave actually uses much less RAM than a running Docker container for the same application. Similarly, while the lower bandwidth of the VSOCK channel increases the latency of an enclave server when handling requests made locally from the parent EC2 instance, this limitation is not observed when the enclave handles requests made over the internet via HTTP. Additionally, we demonstrate successful deployment and validation of a mixed-privacy serving system. We find that the mixed-privacy system behaves as expected, with latency for standard requests increasing when the request stream contains a higher proportion of standard requests, and vice versa if the request stream contains a higher proportion of secure requests.

II. Design

To evaluate the performance of mixed-privacy model serving in Nitro Enclaves, we start by constructing a baseline Docker container which serves predictions from a sample model. We replicate this container as a Nitro Enclave and deploy both containers on a parent EC2 instance, using Nginx as a reverse proxy to handle incoming HTTP requests. We then collect a series of measurements regarding system performance

and load handling behavior. To address the mixed-privacy scenario, we add a simple Node routing service to this setup. The routing service uses API forwarding to direct requests to either the enclave server or the standard server and returns both types of requests to the user. The following sections describe each component of our design in more detail. See Fig. 1 for an overview of our design.

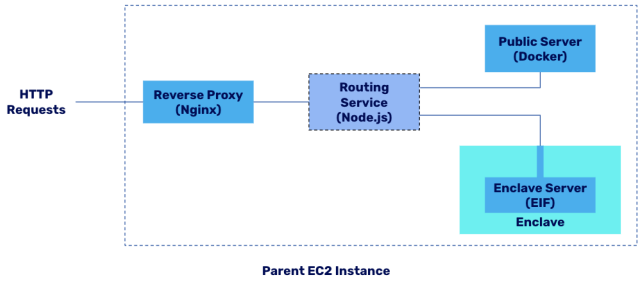

Fig. 1. Schematic of our experimental setup.

A. Model and Docker Setup

We use a ResNet18 model trained on the ImageNet dataset for all of our experiments [5] [6]. We choose the smallest ResNet option due to limitations on Nitro Enclave size, and due to the current lack of GPU support in Nitro Enclaves. We create a standard FastAPI endpoint which receives a batch of base64-encoded images, decodes the batch, and classifies each image into one of ImageNet's 1000 classes. We use Uvicorn to serve our FastAPI endpoint. We construct a Docker container which collects the required files and runs the Uvicorn server on a specified port. To minimize the size of the resulting Docker image, we use a multi-stage build, in which only the compiled files necessary for running the server are copied into the final image. Additionally, since Nitro Enclaves have no external Internet access, we include the PyTorch state dictionary and tokenizer in the base Docker image, rather than downloading them when the server starts.

B. EC2 Instance Setup

We use an EC2 `c6a.8xlarge` instance, which has 32 vCPUs and 64 GB of RAM. This large parent instance size allows us to run the enclave and standard server concurrently for the mixed-privacy experiments. To set up the instance, we install Docker and the `nitro-cli` command line tool for working with Nitro Enclaves [7] [8]. We also install Enclaver, a command line tool which provides wrapper functionalities for building and running enclaves with large memory sizes [9]. We then modify the default values in `/etc/nitro_enclaves/allocator.yml`, a system file which specifies the maximum CPU and memory allocation for enclaves on a given instance. We adjust the values in this file to allocate a maximum of 8 CPUs and 8 GB of RAM for running enclaves. We modify the firewall rules for this instance to allow incoming HTTP and TCP traffic on port 80. Finally, we set up a Nginx reverse proxy to forward incoming HTTP traffic from port 80 to port 9000 [10]. In the individual load handling experiments, we run either the enclave server or the standard server on port 9000. In the mixed-privacy experiments, we add a Node routing service, which is described in more detail below.

C. Enclave Setup

We transfer the Docker image for the standard server to the configured EC2 instance. This can be accomplished either by using Elastic Container Registry (ECR) as an intermediate, or by cloning our Git repository on the EC2 instance and building the Docker image on EC2.

In initial experiments, we used Elastic Container Registry (ECR) as an intermediate repository to transfer the Docker image from a local machine to the EC2 instance. In later experiments, we built the Docker image directly from source on the EC2 instance. Next, we use Enclaver to build an Enclave Image File (EIF) from the source Docker image. We specify a maximum memory allocation of 7 GB for the enclave, and an ingress on port 9000.

D. Start and Throughput Measurements

To measure the start time of each container, we launch the container in detached mode via a shell script with forwarding to a specific local port, then continuously check if we are able to make `curl` requests to the specified port. Once it is possible to make requests, we consider the container launched and record the time.

To measure the maximum throughput of the VSOCK channel as well as the TCP socket used to communicate with the standard server, we use the `iperf-vsock` utility [11]. This utility must be installed inside each container as well as on the parent EC2 instance, so we create a second Docker and enclave which include the `iperf` build files. Since we can't download any files from online in the course of enclave creation, these files must be included in the container structure. For each type of container, we start an `iperf` server inside the container, then start an `iperf` client on the parent EC2 instance and connect to the container. `iperf` sends as many bytes of information across the connection as possible for 10 seconds, collecting measurements every second. We record this information for each container.

E. Load Testing

We use Python's `concurrent.futures` library to generate concurrent requests for load testing. First, we run a load testing script locally on the parent EC2 instance, making TCP requests to port 9000, where either the standard server or the enclave server is running. We use a batch size of 2 for all experiments, and for evaluation consistency, all batches contain two copies of the same image of a cat. For each load scenario, we launch a specified number of concurrent threads, each of which makes two requests to port 9000 before completing. We then measure the minimum, maximum, and average latency of requests made with each number of threads.

For HTTP load testing, we make HTTP POST requests using the external IP of the EC2 instance, with Nginx acting as

a reverse proxy to forward requests from port 80 to port 9000 of the EC2 instance. We use a standard residential Internet connection with a download speed of ∼500 Mbps and an upload speed of ∼25 Mbps, as measured with the University of Michigan's Speed Test utility [12].

F. Mixed-Privacy Serving

For the mixed-privacy scenario, we use a Node.js intermediate server to smoothly handle routing to both endpoint servers [13]. The Node server listens on port 3000, where it receives incoming requests forwarded from port 80 by Nginx. Depending on the API prefix of each request (`/predict-secure/` or `/predict-public/`), the Node server forwards the request to either the enclave server or the standard server, which are running on different ports in this scenario.

To evaluate the routing service, we send a mixed stream of HTTP requests to the public IP address of the EC2 instance, randomly varying the API prefix of each request so it is treated as either secure or standard. We vary the proportion of secure requests in the request stream and measure the minimum, average and maximum response latency of each server under different proportions of secure requests.

G. Memory Performance Measurements

We collect memory performance measurements for each container during startup as well as during normal operation. In both cases, we use the `docker stats` command. To measure memory usage during startup, we run the command immediately after starting a container, and record the highest observed memory value during the 5-second period following startup. To measure memory usage during normal operation, we lightly load both containers simultaneously by sending a stream of mixed-privacy requests as described above. While the containers are handling these requests, we collect the output of the `docker stats` command at 3-second intervals for 100 seconds. We record the average memory usage in MiB (mebibytes) for both containers during this sampling period.

III. Evaluation

For all experiments described above, we performed a single trial unless otherwise specified. Experiments were performed with the EC2 instance in standard configuration, with no other active tasks except the current experiment. Below, we discuss the results of each experiment and potential implications.

A. Container Start Time and I/O Throughput

Fig. 2 shows the results of our container start time and I/O throughput measurements.

We find that the enclave container takes 4.7x longer to start, completing startup in 14.1 seconds compared to only 3.0 seconds for the standard container. This discrepancy in startup time is likely due to additional tasks that the enclave container needs to complete every time it starts. We highlight two enclave-specific startup tasks below:

- **Loading PCR Measurements**: At build time, the enclave generates several precise internal measurements of its own code, filesystem, and user applications, which are saved in Platform Container Registers (PCRs) [14]. These PCR measurements are provided as SHA values every time the enclave is asked to generate a cryptographic attestation document. Therefore, every time the enclave starts, some overhead is incurred from loading and validating the PCR values which were generated at build time.
- **Seeding Random Entropy**: Since an enclave's CPU and memory are physically isolated from the parent EC2 instance, the enclave can become starved of entropy, which typical systems collect from peripheral processes such as disk drive interrupts, network packet arrivals, or key presses [15]. To effectively generate cryptographic attestations, the enclave must have access to a source of high-quality entropy. Therefore, the Enclaver application seeds an entropy pool at enclave start time by reading a sequence of random numbers from the `/dev/random` drive on the parent EC2 instance [16]. Seeding this entropy pool at startup incurs a further start time overhead compared to the standard container, which does not need to pre-store any entropy values.

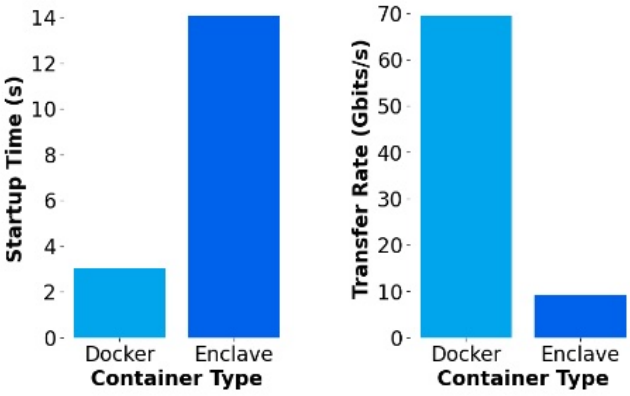

Fig. 2. Container startup times and maximum I/O throughput values.

The magnitude of this start time discrepancy in seconds is not very large, and since the enclave-specific startup tasks take a fixed amount of time regardless of the size of the enclave, we can infer that the number of additional seconds to start an enclave server would be similar regardless of the size of the enclave. However, the requirement for additional startup time would become significant in a scenario where it is necessary to start and stop the enclave server repeatedly. For example, if the enclave container were part of an autoscaling solution where servers are continually launched to handle high traffic, the additional startup time could contribute to increased latency for users.

For I/O throughput measured with `iperf-vsock`, we find that the maximum throughput of the VSOCK channel connecting the enclave to the parent EC2 instance is 9.28 Gbit/s, which is only 13.1% of the maximum throughput of the TCP/IP socket connecting the standard server to the parent

instance. This difference in I/O bandwidth is likely due to the difference in configuration between these two types of sockets. A VSOCK channel is designed for configuration between a hypervisor and its VMs, but repurposed for communication between a parent EC2 instance and a Nitro Enclave; while a TCP/IP socket is designed for inter-process communication on a single machine [17]. Therefore, the two socket types are subject to varied bandwidth limitations. Additionally, AWS documentation states that when multiple enclaves are launched from the same parent EC2 instance, they share a single VSOCK [14]. This implies that the bandwidth limitation would be twice as severe for two enclave servers running on the same EC2 instance, and that users concerned with minimizing latency should avoid running multiple enclave servers on a single parent instance.

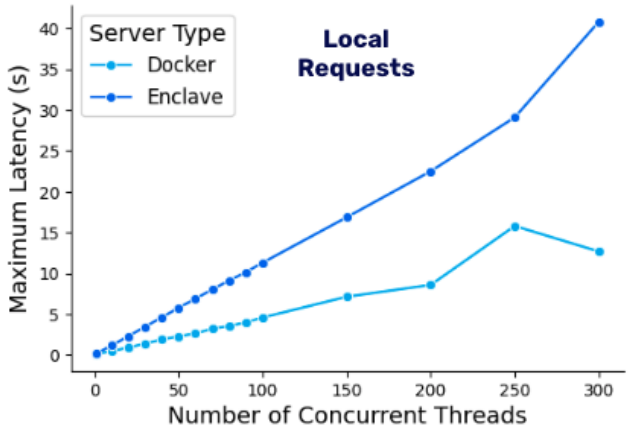

Fig. 3. Local load test from inside the parent EC2 instance.

B. Load Testing

Fig. 3 shows the results of local load testing, in which the load testing script is run on the parent EC2 instance and sends API calls directly to the port where the enclave server or the standard server is running. Fig. 4 shows the results of the same load testing script sending API calls to the external IP of the EC2 instance over a residential network. Note that for the internet load test, we are not able to send more than 100 concurrent requests with our experimental batch size of 2, since higher numbers of requests overload the bandwidth of the residential network.

In the local load testing scenario, we observe significantly higher latency for the enclave server. The maximum latency of the enclave server is 2.86x slower than the standard server when serving 100 concurrent requests, and this gap widens to a 3.62x decrease when serving 300 concurrent requests. This latency limitation is expected behavior based on our previous measurement of the restricted VSOCK bandwidth. As the enclave server becomes more heavily loaded with concurrent requests, the VSOCK channel becomes a bottleneck, and requests must wait longer to pass through the VSOCK channel due to its lower throughput. We can infer that the performance gap would continue to widen as we increase the number of requests, until each channel reaches its maximum capacity.

Interestingly, in the internet load testing scenario, we do not observe the same gap in performance. Although the local load testing scenario shows a clear performance difference even at lower numbers of concurrent requests, we observe no difference in maximum latency in any of our experiments for the internet load testing scenario. As expected when sending requests over the internet, we do observe a significant increase in latency at all load levels compared to the local load test. At a concurrency level of 100 requests, the maximum latency for the standard server increases by 9.76x compared to the local load test. However, the lack of a latency gap between the standard server and the enclave server is an unexpected result. This result suggests that when sending requests over the internet, the speed of internet transfer becomes the limiting factor, rather than the VSOCK bandwidth of the enclave server. It is possible that as requests travel over the internet, they experience variations in travel time which cause them to become more evenly spaced, resulting in a distributed stream of incoming requests which does not stress the VSOCK bandwidth.

To fully understand the impact of the internet on server performance, further work is necessary, including sending concurrent requests from multiple different machines and sending requests from other EC2 instances in the same region as the parent instance. However, we can tentatively conclude that in a typical serving scenario where clients connect to the enclave server over the internet, limited VSOCK bandwidth is less likely to cause an obvious decrease in latency which is observable from the client's viewpoint.

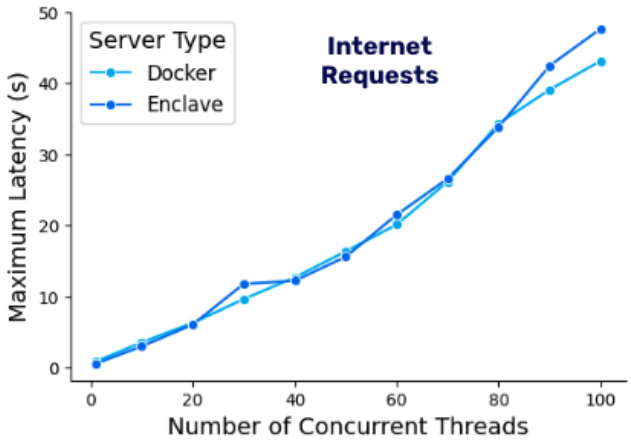

Fig. 4. Internet load test using a residential network.

C. Mixed-Privacy Scenario

How should users address the case where some, but not all, requests are subject to additional privacy concerns? We prototype the Node.js routing service described in II-F as one potential solution for mixed-privacy serving. Fig. 5 shows the performance of the routing service under a light load scenario

of HTTP requests sent over the internet. We observe that when the request stream is mostly composed of standard requests, the standard server becomes more heavily loaded, and standard requests take longer to complete. As the proportion of secure requests in the request stream increases, the enclave server becomes more heavily loaded, and latency for the enclave server increases. When the proportion of standard and secure requests is equal, the maximum latency for secure requests is 1.27x higher than the maximum latency for standard requests. This discrepancy may reflect the bandwidth limitation of the VSOCK channel, but it is also small enough in terms of actual value (about 0.11 seconds) that it may be noise.

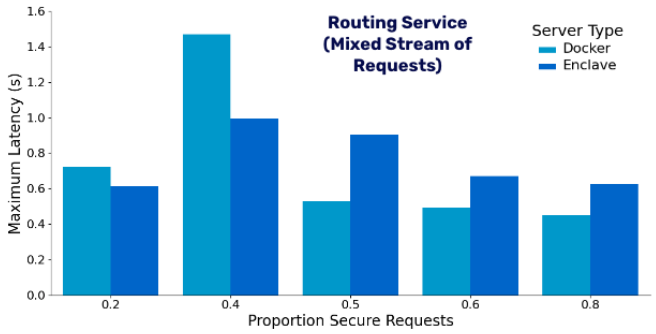

Fig. 5. Varying the proportion of secure requests in a mixed-privacy scenario.

D. Memory Allocation

Table 1 summarizes the results of our memory performance experiments. We discuss each aspect of these experiments in detail below.

- **Storage Memory**: In terms of storage, the enclave image takes up 1.07x as much disk space as the Docker image. This additional storage overhead is incurred when adding enclave components, such as the `nitro-cli` interface, the Enclaver interface, and encryption libraries, as additional layers on top of the existing Docker image. It is possible to observe the size of each individual layer by running the `docker image history` command for either container.
- **Startup Memory**: In the 5-second period while each container is starting, the enclave server consumes 4.55x as much RAM as the standard server, with a peak memory allocation of 1023 MiB. It is possible that this additional memory overhead arises from re-computing and validating PCR hashes on startup, as discussed in III-A. In order to properly validate stored PCR values, the enclave may need to temporarily duplicate parts of its code or filesystem, which could account for a spike in RAM.
- **Runtime Memory**: Perhaps the most unexpected memory result is the amount of memory used by each container during typical runtime, when both servers are lightly loaded in the mixed-privacy scenario. After the initial spike in enclave memory allocation during startup, we observe that enclave RAM drops off sharply to only 3.3 MiB, while RAM allocation for the standard server remains around 225 MiB. There are two possible explanations for this discrepancy. The first is that the tool we use to measure memory allocation, the `docker stats` command, may be failing to capture some aspect of enclave memory usage. This would be understandable, since the memory pages used by the enclave are physically separated from the memory pages for the rest of the system, and `docker stats` may be unable to capture this separation. However, we cross-checked this result against the output of the `cat /proc/meminfo` command, and the two commands give similar results for the amount of runtime memory allocated to the enclave server and the standard server. The second possible explanation is that the enclave server may simply be much more RAM-efficient than the standard server during runtime. Since an enclave is constructed to have no interactive or shell access, the enclave server is not required to run any processes supporting interactivity, and can streamline its resource usage to the single ML server specified in the build file. It is possible that this configuration difference accounts for the lower memory usage of the enclave during typical runtime, but more research is needed to confirm whether this is the case.
- **Build Memory**: One aspect of memory usage which is worth discussing, but which we do not empirically record, is the amount of RAM used when building each type of container. As discussed in II-B, it is necessary to allocate 8 GB of RAM to the enclave server before building, or the build will fail. Since this large amount of RAM does not appear to be fully utilized during startup or runtime, we must assume that it is primarily needed for building the enclave. This memory overhead may be used for computing PCR values, similar to the overhead observed at start time. However, more research is needed to understand the exact cause, and to understand how the build overhead incurrred for enclave images compares to the build overhead for Docker images.

TABLE I
SERVER MEMORY ALLOCATION

Server Type	Enclave	Standard
Storage Memory Allocation (GB)	1.1	1.04
Startup Memory Allocation (MiB)	1023	225
Average Runtime Memory Allocation (MiB)	3.278	256.7

IV. RELATED WORK

In this section, we will discuss the landscape of related research. We will focus on three categories: research which specifically focuses on performance aspects of Nitro Enclaves, research which addresses ML model serving in other secure environments, and research which addresses the performance impacts of security and encryption protocols.

- **Performance measurements on Nitro Enclaves**: To our knowledge, only one other work directly deals with performance measurements in the context of Nitro Enclaves [15]. This work focuses on developing the Nitriding toolkit, which claims to simplify enclave application development. To establish comparison baselines, the authors measure HTTP request latency for a standard server and an enclave server under increasing load scenarios. However, the authors only measure latency for HTTP requests made locally from the parent EC2 instance, rather than requests made over the internet. Additionally, the authors use an extremely minimal application which only responds with the string "hello world", rather than a full ML server as we use in this work.
- **ML model serving in secure execution environments**: Several previous works deal with moving some or all of the ML model serving pipeline into other types of secure environments, such as Intel's SGX Enclaves. The authors of Slalom [18] develop a framework for performing DNN inference inside any secure enclave, but unlike in our work, they outsource computation of linear layers to an untrusted GPU. Additionally, they do not include Nitro Enclaves in their evaluation cases. The authors of Origami [19] also outsource evaluation of some layers to an untrusted accelerator, but they use *cryptographic blinding*, adding random noise to some layers' feature maps to prevent an adversary from retrieving the data. Both of these works evaluate the performance overhead of encrypting and decrypting data within secure enclaves, which is out of scope for our study. DarkNeTZ [20] focuses on the performance impact of running model inference in secure execution environments on *edge devices* such as mobile phones.
- **Performance impacts of encryption**: One other relevant category of previous research is works that consider the performance impact of encryption for ML model serving, with or without specifically considering secure execution environments. The authors of [21] provide a comprehensive set of performance benchmarks for model inference with several popular encryption frameworks. The authors of PFMLP [22] develop an optimized encryption algorithm which improves model training time when used in a federated learning context. PFLMP is one of many papers dealing with encrypted model training, which typically incurs even greater computational overhead than encrypted inference. In CHEX-MIX [23], the authors mitigate the high computational overhead of running homomorphic encryption (HE) in the cloud by using a combination of HE and secure execution environments for model inference. In general, the encryption performance benchmarks from these studies could be combined with the system benchmarks from our study to give a full understanding of the performance overhead of running ML inference on encrypted data within a secure execution environment such as a Nitro Enclave.

V. Discussion

In this study, we generated a series of system performance benchmarks for ML model serving in an enclave server, comparing the enclave server to a standard server running in a Docker container. We also validated a proof of concept for a mixed-privacy serving system, in which the enclave server and the standard server run side-by-side and handle requests from the same stream. Below, we summarize our key findings:

- **The enclave server incurs the most system performance overhead during startup.** We observe a 4.7x increase in start time for the enclave server, as well as a 4.55x increase in RAM usage during startup.
- **Although the VSOCK channel is significantly limited in bandwidth, this limitation does not affect latency when serving internet requests.** We observe that the VSOCK channel can only handle 13.% of the maximum I/O throughput of the TCP/IP socket used by the standard server. This bandwidth restriction increases latency during local load testing, but when requests are made over the internet, the internet itself becomes the bottleneck instead.
- **The enclave server is very memory-efficient during typical operation.** We observe that the enclave image file only takes up 1.07x as much disk space as the corresponding Docker image, and after an initial spike in memory usage during startup, the memory requirement of the enclave server drops to only 1.3% of the memory required for the standard server.

In general, our results suggest that when migrating ML model serving into a Nitro Enclave, users should be most concerned with longer start times and high memory usage during startup, which may be especially relevant if the enclave is part of an autoscaling solution which continually starts and stops its servers. Additionally, users dealing with a very high volume of concurrent requests may wish to deploy multiple enclaves where they would typically use a single standard server. Although we did not observe a latency impact when sending multiple internet requests from a single client, the lower bandwidth of the VSOCK channel suggests that eventually the enclave server would begin to suffer when handling concurrent internet requests from hundreds of different clients. However, users should not be concerned with higher memory overhead during typical runtime, and may actually be able to conserve memory by migrating some non-interactive processes into enclaves. Finally, users dealing with mixed-privacy ML model serving should be able to successfully route requests to enclave and standard servers running concurrently, with minimal negative impact to the performance of either server. In many cases, the most significant performance impact of using an enclave server will not come from system performance configuration but from the encryption protocol used. Therefore, users with an established encryption and key management solution who wish to add Nitro Enclaves to their model serving pipeline should be able to do so without major reconfiguration of their solution architecture.

As ML model serving in the cloud becomes more ubiquitous, we can expect to see a rise in adversarial attacks on both data and models. Developing solutions which effectively protect sensitive computational assets is critical for the future of secure ML. As Nitro Enclaves and other secure cloud computing environments continue to develop, users should consider adopting these technologies to differentiate their security offerings.

REFERENCES

[1] Shuraida, Shadi, and Ryad Titah. "An Examination of Cloud Computing Adoption Decisions: Rational Choice or Cognitive Bias?" Technology in Society, vol. 74, Aug. 2023, p. 102284, https://doi.org/10.1016/j.techsoc.2023.102284.

[2] Long, Saiqin, et al. "A Review of Energy Efficiency Evaluation Technologies in Cloud Data Centers." Energy and Buildings, vol. 260, Apr. 2022, p. 111848, https://doi.org/10.1016/j.enbuild.2022.111848.

[3] Zaeem, Razieh Nokhbeh, and K. Suzanne Barber. "The Effect of the GDPR on Privacy Policies: Recent Progress and Future Promise." ACM Trans. Manage. Inf. Syst., vol. 12, no. 1, Dec. 2020, pp. 1–20. March 2021, https://doi.org/10.1145/3389685.

[4] Amazon Web Services, Inc. "Nitro Enclaves." https://aws.amazon.com/ec2/nitro/nitro-enclaves/.

[5] He, Kaiming, et al. "Deep Residual Learning for Image Recognition." 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), IEEE, 2016, https://doi.org/10.1109/cvpr.2016.90.

[6] Deng, Jia, et al. "ImageNet: A Large-Scale Hierarchical Image Database." 2009 IEEE Conference on Computer Vision and Pattern Recognition, 2009, pp. 248–55, https://doi.org/10.1109/CVPR.2009.5206848.

[7] Merkel, Dirk. "Docker: Lightweight Linux Containers for Consistent Development and Deployment." Linux Journal, vol. 2014, no. 239, Mar. 2014, p. 2. March 2014, https://dl.acm.org/doi/10.5555/2600239.2600241.

[8] Paraschiv, Andra, and Alexandru Vasile. "Nitro Enclaves CLI." Rust. 2019. https://github.com/aws/aws-nitro-enclaves-cli.

[9] Haering, Russell, Rob Szumski, Alex Crawford, and Eugene Yakubovich. "Enclaver." Rust. 2022. https://github.com/edgebitio/enclaver.

[10] Reese, Will. "Nginx: The High-Performance Web Server and Reverse Proxy." Linux Journal, vol. 2008, no. 173, Sept. 2008, p. 2. September 2008, https://dl.acm.org/doi/fullHtml/10.5555/1412202.1412204.

[11] Garzarella, Stefano. "Iperf-VSOCK." C. 2023. https://github.com/stefano-garzarella/iperf-vsock.

[12] University of Michigan. "University of Michigan Speed Test." https://speedtest.it.umich.edu/.

[13] Dahl, Ryan. "Node.Js." JavaScript. 2014. https://github.com/nodejs/node.

[14] Amazon Web Services. "Nitro Enclaves Concepts." Nitro Enclaves User Guide, 2023. https://docs.aws.amazon.com/enclaves/latest/user/nitro-enclave-concepts.html.

[15] Winter, Philipp, et al. "Nitriding: A Tool Kit for Building Scalable, Networked, Secure Enclaves." arXiv [cs.CR], 8 June 2022, http://arxiv.org/abs/2206.04123.

[16] Haering, Russell. "Enclaver Architecture." EdgeBit, 2023. https://edgebit.io/enclaver/docs/0.x/architecture/.

[17] "Vsock(7) - Linux Manual Page," October 30, 2022. https://man7.org/linux/man-pages/man7/vsock.7.html.

[18] Tramèr, Florian, and Dan Boneh. "Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware." arXiv [stat.ML], 8 June 2018, http://arxiv.org/abs/1806.03287.

[19] Narra, Krishna Giri, et al. "Privacy-Preserving Inference in Machine Learning Services Using Trusted Execution Environments." arXiv [cs.LG], 7 Dec. 2019, http://arxiv.org/abs/1912.03485.

[20] Mo, Fan, et al. "DarkneTZ: Towards Model Privacy at the Edge Using Trusted Execution Environments." Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services, Association for Computing Machinery, 2020, pp. 161–74, https://doi.org/10.1145/3386901.3388946.

[21] Haralampieva, Veneta, et al. "A Systematic Comparison of Encrypted Machine Learning Solutions for Image Classification." Proceedings of the 2020 Workshop on Privacy-Preserving Machine Learning in Practice, Association for Computing Machinery, 2020, pp. 55–59, https://doi.org/10.1145/3411501.3419432.

[22] : Fang, Haokun, and Quan Qian. "Privacy Preserving Machine Learning with Homomorphic Encryption and Federated Learning." Future Internet, vol. 13, no. 4, Apr. 2021, p. 94, https://doi.org/10.3390/fi13040094.

[23] Natarajan, Deepika, et al. "CHEX-MIX: Combining Homomorphic Encryption with Trusted Execution Environments for Two-Party Oblivious Inference in the Cloud." Cryptology ePrint Archive, 2021, https://eprint.iacr.org/2021/1603.