VESSELS: Efficient and Scalable Deep Learning Prediction on Trusted Processors

Kyungtae Kim
Purdue University
kim1798@purdue.edu

Chung Hwan Kim
University of Texas at Dallas
chungkim@utdallas.edu

Junghwan "John" Rhee
University of Central Oklahoma
jrhee2@uco.edu

Xiao Yu
NEC Laboratories America
xiao@nec-labs.com

Haifeng Chen
NEC Laboratories America
haifeng@nec-labs.com

Dave (Jing) Tian
Purdue University
daveti@purdue.edu

Byoungyoung Lee
Seoul National University
byoungyoung@snu.ac.kr

ABSTRACT

Deep learning systems on the cloud are increasingly targeted by attacks that attempt to steal sensitive data. Intel SGX has been proven effective to protect the confidentiality and integrity of such data during computation. However, state-of-the-art SGX systems still suffer from substantial performance overhead induced by the limited physical memory of SGX. This limitation significantly undermines the usability of deep learning systems due to their memory-intensive characteristics.

In this paper, we provide a systematic study on the inefficiency of the existing SGX systems for deep learning prediction with a focus on their memory usage. Our study has revealed two causes of the inefficiency in the current memory usage paradigm: large memory allocation and low memory reusability. Based on this insight, we present VESSELS, a new system that addresses the inefficiency and overcomes the limitation on SGX memory through memory usage optimization techniques. VESSELS identifies the memory allocation and usage patterns of a deep learning program through model analysis and creates a trusted execution environment with an optimized memory pool, which minimizes the memory footprint with high memory reusability. Our experiments demonstrate that, by significantly reducing the memory footprint and carefully scheduling the workloads, VESSELS can achieve highly efficient and scalable deep learning prediction while providing strong data confidentiality and integrity with SGX.

ACM Reference Format:
Kyungtae Kim, Chung Hwan Kim, Junghwan "John" Rhee, Xiao Yu, Haifeng Chen, Dave (Jing) Tian, and Byoungyoung Lee. 2020. VESSELS: Efficient and Scalable Deep Learning Prediction on Trusted Processors. In *ACM Symposium on Cloud Computing (SoCC '20), October 19–21, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3419111.3421282

1 INTRODUCTION

Deep learning (DL) systems are widely used in many applications, such as face recognition, intelligent personal assistants, and object detection. Many of these systems run in a shared computing environment, a cloud infrastructure in particular, for cost reduction and scalability [5, 6, 8]. Despite the benefits, the increasing frequency of sophisticated data breaches in the cloud [3, 7] and the emergence of new attacks that steal machine learning data [21, 22, 27, 45, 50] have led to a major concern of running privacy-sensitive DL systems in an untrusted computing environment. While encryption can protect these data at rest in storage and their transfer (e.g., disks and networks), it does not protect them while they are *in use* during the computation.

In this regard, Intel Software Guard Extensions (SGX [18]), a hardware feature that provides a trusted execution environment, gained strong attention as it provides primitives to protect the confidentiality and integrity of sensitive data in use. SGX provides a private memory region (namely, *an enclave*) to load a program and protect its code and data from other untrusted programs during the execution. Public cloud providers, such as Microsoft Azure [12] and IBM Cloud [11],

SoCC '20, October 19–21, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8137-6/20/10…$15.00
https://doi.org/10.1145/3419111.3421282

Techniques	Performance Overhead
Eleos [43]	84.02x
TensorSCONE [34]	3.18x
TF Trusted [2]	20.11x

Table 1: Performance of recent SGX systems running DL prediction compared to an unprotected run. We used the InceptionV3 model [52]. Eleos is evaluated with Darknet [46] atop. TensorFlow Lite [9] is used as the baseline for TensorSCONE and TF Trusted.

offer SGX-capable computing platforms to support confidential computation. Using SGX, one can protect DL data in an untrusted, adversarial computing environment with a strong attack model. In recent years, both academia and industry have proposed techniques using SGX to protect machine learning computation [2, 34, 41, 54**?** , 55] and other applications [14, 16, 43].

Although SGX provides strong security in an untrusted computing environment, it has a critical performance issue, limiting its usability for DL computation. Existing studies [19, 53] have found that the main cause is the limited capacity of a physical SGX memory region for protected enclave pages, called Enclave Page Cache (EPC). Due to the hardware design restriction, the size of EPC is limited to 128 MB[1]. If an enclave program over-provisions more than the size of EPC, it suffers from a critical performance issue as it requires expensive *secure paging* between the EPC and non-EPC (i.e., unprotected) memory. For a DL system, this limitation is crucial because DL programs are memory-intensive and require large memory footprints – popular DL models (such as VGG16) require up to 1.1 GB memory footprint (§5). In a production DL system with multiple enclaves that handle many prediction requests in parallel, this problem is even more critical since all enclaves in the physical machine have to share and compete for a single EPC region, causing low scalability by EPC thrashing.

There are several approaches that alleviate the EPC issue. However, we found that even the state-of-the-art SGX systems [2, 34, 43] are still significantly limited in performance as shown in Table 1, and some of them trade functionality and accuracy for security. Our analysis found that these approaches can be characterized into two types: a user-level paging mechanism [43], and model size reduction [2, 34]. Both approaches are still limited in addressing the EPC issue. The user-level paging is designed for programs with much less intensive memory accesses (i.e., server applications). The model reduction does not address other large memory usages, such as memory buffers for input and intermediate activations in DL computation. Besides, such model reduction based on quantization [29] affects the model accuracy and limits the support for certain operations (e.g., floating point operations [2, 34]). Furthermore, none of the existing approaches address the scalability problem with multiple enclaves.

In this paper, we conduct a systematic study on the EPC usage problem of SGX-enabled DL prediction. By profiling the memory usage of popular DL models [26, 28, 33, 51, 58], we made the following key observations. First, DL computation is structured based on a sequence of individual neural network layers. The data needed for each layer computation are highly independent, and layers have separate memory buffers, resulting in *a bulk memory allocation* that keeps the EPC full. Second, most of these memory buffers are used only once for a short time and never used again throughout the entire DL execution without being deallocated. Such a memory usage pattern with *low reusability* causes frequent secure paging as many pages have to be loaded into the EPC in a short period of time. Lastly, we found that the memory usage of DL computation can be identified *before execution*. For instance, memory usage patterns, such as how much memory each layer will allocate and which layer accesses which memory buffer, can be accurately analyzed ahead to address the EPC problem. These observations offer an opportunity to make DL prediction more efficient and scalable under SGX protection.

Based on our observations, we propose Vessels, a highly efficient and scalable SGX system for DL prediction. *A vessel* is a customized enclave that minimizes the memory footprint and optimizes the performance of DL prediction for a target DL model. It does not incur any accuracy loss or limit the functionality of the protected DL as it does not require modification to the DL model or semantics unlike related approaches [2, 34]. Our system creates and manages multiple parallel *vessels* to enable scalable processing of many prediction requests by minimizing the chances of EPC thrashing. To facilitate the deployment in the cloud, we implement our prototype of Vessels in a Docker container [39].

Our design has several key techniques: (1) optimized memory usage planning, (2) on-demand parameter loading, and (3) EPC-aware prediction scheduling. Vessels examines a target DL model before computation to equip the vessel with *an optimized memory pool*. During the computation, this memory pool is shared by all layers (or sub-layers) following the optimized memory usage plan that *reduces the physical memory footprint* and *improves the reusability of the memory* significantly. In addition, Vessels *eliminates the bulk allocation* of large memory buffers for model parameters by retrieving them *in an on-demand fashion* into recycled memory buffers in the optimized memory pool. For the best possible scalability with many prediction requests, a new

[1]Protected programs can only use around 93 MB in practice as the rest of the 128 MB is used for metadata [19].

prediction is scheduled into a vessel with the consideration of its estimated memory usage by balancing CPU parallelism and the impact of EPC thrashing.

We have evaluated our design with various pre-trained DL models. Our experimental results show that VESSELS outperforms the state-of-the-art SGX systems with DL prediction in both memory usage and run-time performance. Specifically, VESSELS achieves a significant reduction of EPC usage by 73-91% and improves the latency by 18-94% in a single prediction enclave, compared to our baseline SGX system. In our scalability test with many prediction requests, VESSELS shows an average of 195% improvement over the baseline SGX system demonstrating its practicality to handle real-world workloads in the cloud under SGX protection.

In summary, we make the following contributions.

- We conduct a systematic study on the memory usage of popular DL models under SGX protection and found inefficiency which causes the high cost of EPC thrashing.
- We propose VESSELS, an SGX system that addresses the EPC limitation for efficient and scalable DL prediction. VESSELS provides strong security to DL computation with only a marginal overhead and without compromising the functionality and accuracy of the prediction.
- We provide various quantitative results to show the inefficiency of the existing DL systems with SGX and the performance improvement that VESSELS achieves by overcoming the challenges in comparison.

2 BACKGROUND

In this section, we provide the background of Intel SGX and deep learning concepts related to our work.

2.1 Intel SGX

Intel SGX [18] is a specialized hardware feature to realize a trusted execution environment, preserving the confidentiality and integrity of code and data *in use* for user-space programs. A key component of SGX is a hardware-protected memory region, i.e., *an enclave*, which contains and protects the security-sensitive code and data against untrusted entities. These untrusted entities include both privileged and unprivileged programs. To support interactions between an enclave and untrusted programs (e.g., for system call invocation), SGX provides special instructions `ECALL` and `OCALL`, which enter and exit an enclave, respectively.

Enclave pages are encrypted and placed in a physical memory region, Enclave Page Cache (EPC), of which the size is up to 128 MB. Since a part of the EPC is pre-assigned for metadata (e.g., for the integrity checking of the pages), the size of the EPC available for enclave programs is only around 93 MB (23,936 pages) in practice [19]. Whenever the EPC space is insufficient for a new enclave page, the SGX driver evicts an old page from the EPC to the untrusted main memory and loads the new page into the EPC. This *secure paging* is expensive, costing up to hundreds of thousands of cycles for each swapping, as it does not only entail transitions to and from the enclave (causing TLB flush), but also involves page encryption and integrity checks. In addition, since the EPC is shared by all enclaves running in the physical machine, multiple enclaves can quickly exhaust the EPC, causing EPC thrashing which is similar to the thrashing of virtual memory.

2.2 Deep Learning and Models

Deep learning (DL) is a family of machine learning methods whose models are based on artificial neural networks, which use multiple layers to extract higher-level features in a progressive manner. Each layer is responsible for transforming one input data into a slightly more abstract representation. As the computation proceeds to deeper layers, a DL model can learn complex functions to extract feature representations, such as the object edges for image classification. A DL model is often stored in a file and consists of *hyper parameters* and *model parameters*. During the development of the neural network, *hyper parameters* are manually determined and fine-tuned to represent the network structure, such as the number of layers and connections between different layers. The weights on the connections, also known as *model parameters*, are learned automatically during training based on training data and a loss function. After deployment, the DL program takes the learned model and produces prediction outcome given input data (e.g., the class of an input image for image classification).

3 PROBLEM SCOPE

In this section, we describe the problem scope of VESSELS. Since this paper focuses on efficient and scalable DL on trusted processors, it has the following two problem scopes: (1) the protection is CPU only. and (2) the security realm that this paper focuses on is a prediction phase of DL. Our problem scope is aligned with that of previously proposed DL systems for trusted processors [2, 34].

CPU-only protection. Hardware-assisted trusted execution environments (including Intel SGX and AMD SEV) assert that its trusted boundaries only include the CPU, and all other computational units (particularly hardware accelerators such as GPUs and FPGAs) are untrusted. As such, in order to fully (and correctly) leverage the security guarantees offered by a trusted execution environment, DL has to be solely running within the CPU. If any data during DL is transferred to a GPU or an FPGA, neither the confidentiality

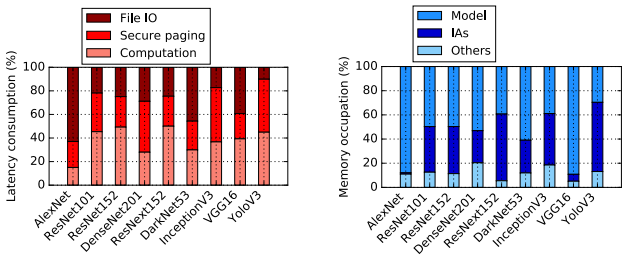
nor integrity security guarantees of the trusted execution environment would hold.

DL prediction system. While DL is generally divided into two phases, learning and prediction, we focus on protecting the prediction phase. Training requires an extensive computing power thereby generally off-loaded to untrusted hardware accelerators. In contrast, prediction requires much less computing power and it is performed frequently whenever a user application requests for service (e.g., from an edge device). As such, the efficiency and scalability of a DL prediction system in the cloud are significant factors in user application performance [32]. Although a DL prediction system can benefit from the cost reduction and flexibility of a cloud infrastructure, privacy-sensitive organizations have long been concerned about running prediction in an untrusted environment. Such reasons make a DL system for trusted processors a desirable computing environment for prediction as it does not require acceleration and it demands a strong confidentiality protection.

4 THREAT MODEL

This paper assumes that adversaries aim to uncover user's privacy, in particular, by stealing DL user input, training data or prediction results. An outcome of activation in each layer is another target of attacks as it can help infer the private input (or output) [23]. Similarly, we assume that adversaries may access model parameters to reconstruct the training data [22, 36]. Hyper parameters, however, remain insensitive and public because they do not reveal any information of input data.

Relying on the Intel SGX protection, we assume strong adversaries that may exploit any untrusted software and hardware as an attack surface in the target computing environment. More specifically, we assume that adversaries may control or compromise other application and privileged software (e.g., OS and hypervisor) in order to attack the protected DL prediction system by SGX. Furthermore, adversaries may try to reveal sensitive data by accessing untrusted hardware components, such as the physical memory. The only trusted components are the processors and our DL prediction system running inside the enclaves. With respect to other concerns beyond the SGX security, such as advanced side-channel attacks [17, 40], one may prevent them with the assistance of recent (existing) defenses [42]; however, they are orthogonal to this research as we focus on improving the performance of DL prediction without compromising the level of security that SGX provides.



(a) Latency. (b) Memory usage.

Figure 1: Latency and memory usage distributions of DL prediction in a straw-man SGX system.

5 SYSTEMATIC ANALYSIS OF DL PREDICTION WITH INTEL SGX

In this section, we conduct a comprehensive study of DL prediction in terms of its low-level execution details regarding per-layer memory usages and their dependencies. The primary intuition that motivates our work comes through the following research question: *What are the main factors that contribute to the large latency of DL prediction with SGX?* To answer this question, we analyze the execution of DL prediction enclaves with SGX and systematically identify the root cause of the slowdown.

5.1 DL Prediction Systems with SGX

In our analysis, we conduct experiments on several SGX systems that perform DL prediction inside an enclave, and we analyze their execution to identify the root cause. In particular, we first examine a straw-man SGX system that runs DL prediction without any performance optimization, and then analyze three recent SGX systems that are designed to address the performance problem. As we clarify in §3, we focus on CPU-only prediction workloads.

5.1.1 Straw-man System. We use a straw-man system to represent a general SGX system that launches an enclave for confidential DL prediction execution (Straw-man). We have implemented this system based on an existing DL framework, Darknet [46], and Intel SGX SDK, with minor changes to Darknet to run prediction in an enclave. Table 2 shows the result of our experiment on Straw-man with 9 popular models. As seen in the table, the latency of DL prediction under SGX protection is an order of magnitude higher than the unprotected prediction (7x higher on average). For all models, *the peak memory usage is much larger than the size of the EPC* (93 MB except metadata) hence triggering secure paging. The number of secure page swaps during the execution is substantial across all models (from an order of

Model	# Layers	Model Size	Peak Memory Usage	Prediction Latency			# Secure Paging
				Unprotected	Protected	Overhead	
AlexNet [33]	13	238 MB	274 MB	1.03s	21.56s	20.9x	0.46 M
ResNet101 [26]	137	159 MB	319 MB	4.83s	24.31s	5.03x	0.43 M
ResNet152 [26]	204	220 MB	441 MB	6.73s	32.54s	5.83x	0.51 M
DenseNet201 [28]	304	66 MB	376 MB	2.54s	12.74s	5.01x	0.23 M
ResNext152 [58]	204	217 MB	566 MB	6.88s	36.17s	5.25x	0.70 M
DarkNet53 [46]	77	159 MB	273 MB	4.43s	23.51s	5.30x	0.41 M
InceptionV3 [52]	145	92 MB	337 MB	8.34s	38.63s	4.51x	0.93 M
VGG16 [51]	24	528 MB	1,121 MB	7.43s	117.79s	15.85x	3.76 M
YoloV3 [47]	106	237 MB	840 MB	53.05s	162.98s	3.07x	4.24 M

Table 2: Performance and memory usage of DL prediction in a straw-man SGX system (STRAW-MAN). We executed only one prediction enclave at a time to ensure that other enclaves do not affect the result.

hundred thousands to millions), indicating that they must be addressed to achieve low-latency prediction with SGX.

Interestingly, the model size is less than 50% of the peak EPC memory for most of the models, suggesting that there are other large memory usages during the execution. To better understand the impact of secure paging and other operations, we provide a breakdown of the latency and memory usage in Figure 1. As shown in Figure 1a, *secure paging is responsible for a major part of the prediction latency* (`Secure paging`) – it takes up over 28% of the total prediction latency. `File IO` represents the latency through `OCALL` to load various data (including input data, model parameters, and classification labels) from the disk. `Computation` denotes the latency spent for arithmetic computation and memory accesses (excluding the time spent for secure paging).

Intuitively, the overhead induced by secure paging comes from *a large volume of memory usage* for different data, as presented in Figure 1b. Following the largest memory space that the model parameters occupy (`Model`), the second largest portion (18%) of the memory is used by *intermediate activations* (`IAs`). These IAs are generated by neural layers at run-time and thus *difficult to quantitatively compress* without affecting the prediction accuracy or latency [57]. The rest of the memory is occupied by hyper parameters, classification labels, and other miscellaneous data (`Others`).

5.1.2 State-of-the-art Systems. In addition to the straw-man system, we further analyze the performance problem on recent state-of-the-art SGX systems [2, 34, 43] running DL prediction. These systems are designed to improve the performance of an SGX enclave by addressing the EPC problem in two different ways: (1) user-level paging inside the enclave and (2) model size reduction through quantization. The result of our experiment on these systems is presented in Table 3.

	Peak Memory	Model Reduction	# User-level Paging	# Secure Paging
Eleos [43]	430 MB	✗	135 M	-
TensorSCONE [34]	533 MB	✓	-	0.17 M
TF Trusted [2]	1,331 MB	✓	-	0.15 M

Table 3: Performance and memory usage of state-of-the-art SGX systems running DL prediction with InceptionV3. In Eleos, the enclave is configured with the maximum page cache size and tested with Darknet atop.

User-level paging. Focusing on the elimination of secure paging for an enclave, Eleos [43] proposes a user-level paging mechanism to run inside the enclave. Given a portion of the EPC (smaller than 93 MB) as a fixed budget, Eleos leverages the space as the page cache for independent paging inside the enclave to avoid enclave exits. This user-level paging is 3-4x faster than secure paging. However, in our analysis we found that Eleos suffers from *a significant number of user-level page swaps* while running DL prediction, over 100x page swaps compared to secure paging. The root cause was the fine-grained and frequent memory accesses to individual elements of large (float-point) vectors, which Eleos is not designed to handle with high efficiency. For every element accessed, Eleos requires an address translation and this results in a substantial performance overhead (Table 1) due to *the excessive number of elements to compute* – the prediction latency is over 15 times higher than STRAW-MAN. Such a memory access paradigm is common in memory-intensive DL programs and thus it is desired to have a scalable SGX system with many frequent memory accesses.

Model reduction. TensorSCONE [34] and TF Trusted [2] alleviate the EPC problem of SGX-enabled DL prediction

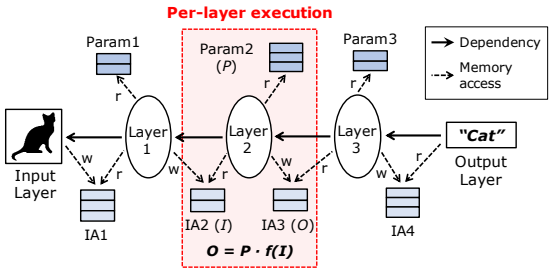


Figure 2: Memory accesses of individual layers in DL prediction (r: memory reads, w: memory writes, *I*: input IA, *O*: output IA, and *P*: model parameters of the layer).

by reducing the memory footprint of the DL model. TensorSCONE is based on a secure SGX container, SCONE [14] while TF Trusted [2] relies on an SDK, Asylo [4], for the adoption of SGX protection on DL prediction. The model reduction is achieved by using TensorFlow Lite [9], a lightweight DL prediction framework originally designed for resource-scarce embedded devices. TensorFlow Lite reduces the size of a DL model through "integer-arithemetic-only" quantization [29]. In our experiment, the quantized model resulted in a much lower number of secure page swaps in these systems than Straw-man (around 85% reduction). However, *their prediction latency is still an order of magnitude higher than the unprotected execution* (Table 1). On top of that, such model reduction *degrades model accuracy and does not allow certain operations* during the prediction (e.g., 2-3% accuracy reduction and unsupported floating-point operations [29]). This limitation is difficult to completely avoid as the approach requires modification to the model parameters and operation semantics. Existing model reduction techniques thus focus on minimizing those negative impacts as much as possible (§9).

Our analysis on these SGX systems motivates a need for a more comprehensive diagnosis on the memory usage of DL prediction for various data, including the model, IAs and other data, to identify the root cause of their *yet-significant performance overhead* and address the EPC problem *without accuracy and functionality loss.*

5.2 Memory Usage of DL Prediction

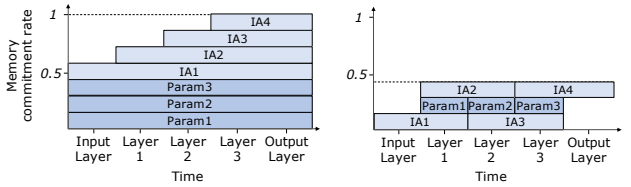
In order to precisely identify the root causes of the EPC problem, we conduct an in-depth diagnosis of the large memory usage of DL prediction during execution. Our diagnosis is based on program analysis on a number of DL frameworks (Darknet [46], TensorFlow [13], and Caffe [31]). We used both dynamic program analysis [37] and manual source code inspection.

Per-layer execution. DL prediction is computed through the individual layers of the neural network in the DL model. Figure 2 illustrates the execution of a simple feed-forward network for an image classification prediction. The example network consists of three hidden layers (Layers 1-3) in addition to the input and output layers. The layers are executed *sequentially* from the input to the output layers through the hidden layers. The computation of each layer is semantically independent. The input layer projects the input data to the first intermediate activations (IA) (`IA1`) and the output layer produces the prediction result based on the last IA (`IA4`). Each hidden layer performs its own computation on a bounded set of memory buffers to produce an IA. For instance, Layer 2 performs operations (e.g., convolution, dot product, and activation) after reading the model parameters (*P*) from `Param2` and input (*I*) from `IA2`, and then writes the output (*O*) to `IA3`.

Layer dependency. We observe that the layers are executed in the reverse direction of their *dependencies*. For example, Layer 2 has a dependency on Layer 1 since it takes the output of Layer 1 (`IA2`) as its input. Similarly, Layer 3 has dependency on Layer 2 as the output of Layer 2 (`IA3`) in turn becomes the input of Layer 3. Because *a dependent layer* requires *the dominant layer*[2] which is the layer to writes its output to the memory *before* the dependent layer reads it, the execution of DL prediction must follow the sequence of the layers through the dependencies. In a more complex neural network, a layer may have more than one dominant layers (e.g., a layer that concatenates outputs from multiple dominant layers). We found that in many networks, however, a layer typically only has one dominant layer for the input exclusively (but for no other data), and thus layers in general have *highly independent computation*. In addition, the dependency information can be *explicitly retrieved* from the hyper parameters of a model along with other information, such as the layer operations and the sizes of the model parameters and IAs of each layer.

Memory management. Our detailed analysis on the execution of DL prediction found *significant inefficiency* in memory management. Specifically, in the current design of DL programs, the memory buffers for model parameters and IAs are allocated (i.e., *reserved*) in the virtual memory *at the beginning of the execution all at once*. The pages for these memory buffers are *committed* to the physical memory (i.e., the EPC) when the program accesses them for the first time. Importantly, the commitment of these pages is kept *throughout the entire DL execution* and de-allocated only after the output layer generates the prediction result. Figure 3 shows how the committed memory grows over time as the layers

[2]Taking a similar term to a dominator node in control flow graphs.



(a) Current. (b) Optimal.

Figure 3: Current and (virtually) optimal lifetime of committed memory buffers in a DL program during prediction.

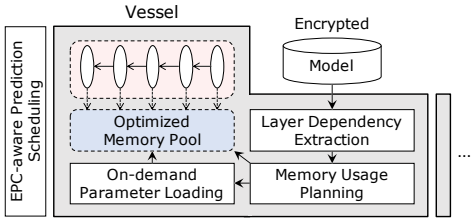


Figure 4: Architecture of VESSELS.

are executed in the current design of DL memory management (Figure 3a) in comparison with an optimal design that may be feasible (Figure 3b). As illustrated, the current memory management of a DL program has a high and increasing commitment rate. Such usage of the physical memory results in *a high secure paging rate* since the large memory buffers fill up the EPC in the early stage of the execution. The details of the current DL memory management are as follows.

First, the model parameters of the entire network (i.e., `Param1`, `Param2`, and `Param3` in Figure 3a) are loaded into the memory together at once from the disk. In DL computation, however, each layer only requires its own set of model parameters in the memory (but no other parameters), as they are *exclusively* assigned to only one corresponding layer. During the execution of other layers, the pages are in the committed state *unnecessarily* and increase the EPC usage of the DL prediction. Similarly, the IAs are committed in memory for a long time although they are only required for a short time period. Unlike model parameters, IAs are generated dynamically by the layers during the prediction. We found that after an IA is committed to the memory, it is only accessed by *a very small number of layers* (layers that produces it and other dependent layers) for a short period of time and *never accessed* for the rest of the time.

Such *low memory reusability* in the design resulted in high occurrences of secure paging in our experiment, and thus a resolution is desired to achieve optimal memory usage. Most DL models have a much larger number of layers than our example network (up to 304 in the models we tested), and thus the inefficiency can be more significant in practice. Notably, such a paradigm of inefficient memory usage may not be problematic for DL prediction without SGX since the main memory is abundant in most computing environments. However, this causes *a severe performance drop* in protected DL prediction with SGX ($5.1) due to the small EPC capacity and the inefficient memory usage that we discovered.

In a production DL system that receives many prediction requests for a short period of time, this problem becomes much more serious as multiple enclaves in a physical machine have to run in parallel and compete for a small EPC region. This results in excessive secure paging (EPC thrashing) and *low scalability* of the system with multiple enclaves. Consequently, this problem motivates us to develop a new DL system with optimized memory management (similar to Figure 3b) for efficient and scalable prediction with SGX.

6 DESIGN OF VESSELS

In this section, we present the design of our system, VESSELS. VESSELS is designed to address the memory usage inefficiency of current DL systems with SGX, based on the discovery that we made in our systematic analysis. Figure 4 shows the overall architecture of VESSELS. In our system, *a vessel* is an optimized enclave for a given DL model. Using the layer dependency information extracted from the model, a vessel minimizes the memory footprint of the DL program in the EPC and thus achieves efficient DL prediction. Multiple vessels can be run in parallel to achieve high scalability with many prediction requests in a multi-core environment, considering both the EPC and core utilization. We describe the details of our key techniques as follows.

6.1 Layer Dependency Extraction

Given an encrypted DL model in the disk, a vessel first reads the hyper parameters in the model with an `OCALL` and decrypts them into a temporary memory buffer before the computation of the prediction. In our system, the hyper parameters are loaded into a network graph to represent the neural network structure in the memory. Each node in the graph represents a layer and has the hyper parameters that determine the layer computation, such as the operations to perform, a number of model parameters, and a number of IA elements to produce. Each directed edge connects two nodes to represent a data flow between two layers through the IA.

Layer dependency graph. Our graph analyzer examines the network graph and constructs *a layer dependency graph* that has the per-layer memory sizes and dependency information as its nodes and edges, respectively. To construct this graph, the analyzer identifies the dependent layers for each layer in the network graph where these dependent layers

Algorithm 1 Memory usage planning using a layer dependency graph.

```
1:  function GetOptimizedMemoryBuffers(G)
2:      B ← new list
3:      for l ∈ G.layers do
4:          if l is a hidden layer then
5:              B.append( new Param( GetParamSize(l), l, 1 ) )
6:          d ← max(l.D)                          ▷ Last dependent layer
7:          B.append( new IA( GetIASize(l), l, d − l + 1 ) )
8:      B′ ← sort B by the last attribute          ▷ Sort by ascending lifespan
9:      return B′
10: function GetMemoryUsagePlan(G)
11:     P ← new list
12:     M ← new |G.layers| array of zeros
13:     for b ∈ GetOptimizedMemoryBuffers(G) do
14:         end = b.begin + b.lifespan
15:         b.offset ← max(M[b.begin] ... M[end])
16:         M[b.begin] ... M[b.end] ← b.offset + b.size
17:         P.append( new Allocate(b.begin, b.offset, b.size) )
18:     Size ← max(M)                             ▷ Memory pool size
19:     return P, Size
```

are the destinations of all edges that start from the layer. In addition, it leverages the hyper parameters in each layer to calculate the sizes of the per-layer memory buffers for the model parameters and IA. Unlike other programs exhibiting non-deterministic behaviors at runtime, such memory sizes *can be accurately identified* in a DL program because the allocation of the memory buffers must follow the pre-determined hyper parameters.

6.2 Memory Usage Planning

In our system, each vessel is equipped with *an optimized memory pool* and this memory pool operates by following *a memory usage plan* that our technique generates.

Optimized memory pool. An optimized memory pool is an EPC-committed contiguous memory space that facilitates high memory reusability. In a vessel, all layers in the DL program use this memory pool to allocate memory buffers for model parameters and IAs. The size of the memory pool is *fixed* and determined *before the computation* of the layers by the memory usage planning. Because the memory usage plan *recycles* a significant amount of memory in the memory pool, the size of the memory pool is much smaller than the memory usage before the optimization. This allows our system to keep only a small number of pages for each vessel inside the EPC, resulting in efficient utilization of the EPC.

Memory usage plan. Algorithm 1 presents how a memory usage plan is generated. Using a layer dependency graph (G) as an input, this algorithm finds *the optimized lifespans* of the memory buffers and generates a plan to allocate them while reusing the memory space. Figure 5 shows an example memory usage plan.

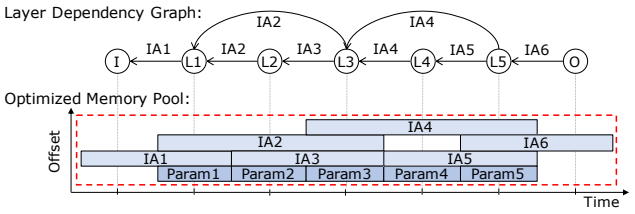


Figure 5: An example memory usage plan on an optimized memory pool, derived from a layer dependency graph (I: input layer, 0: output layer, L1-L5: hidden layers).

First, it goes through each node (i.e., layer) in the layer dependency graph sequentially from the front to the back layers in order to use the layer and dependency information (lines 3-7). For each hidden layer, the optimized lifespan of the model parameter buffer is constantly one layer since the parameters are exclusively assigned to one corresponding layer (line 5). On the other hand, the optimized lifespan of a IA buffer is determined by the dependent layers. For example, in Figure 5 IA2 is accessed by the layers L2–L3 and thus it must be present in the memory until L3 finishes using it. Similarly, the optimized lifespan of IA4 is L3–L5, from the beginning of the layer that produces the IA to the end of the last dependent layer (lines 6-7). After the optimized lifespans (B) are identified, the memory buffers are re-arranged to have the ones with the shortest lifespan first and the longest lifespan last on the list (B').

Given the optimized lifespans of the memory buffers, allocation is planned on each buffer (lines 13-17). The offset of a new buffer in the memory pool (M) is determined by the current size of the committed space in the optimized timespan (line 15). The new buffer will be allocated at the lowest offset above the previously committed space. In this way, the allocation plans (when to allocate, the offset and size) for all memory buffers are made (line 17). In addition, the size of the optimized memory pool is determined (line 18).

Sub-layers. Due to massive model parameters, certain layers often consume excessively large memory in reality, inducing a large optimized memory pool. To address this, our framework partitions the entire computations in a large layer into multiple sub-layer computations. The rationale is since a layer computation is nothing but a dot-product operation, vector operations can be divided into sub-layer partitions, making it possible to partially load model parameters. Considering the algorithm above, while estimating the size of parameters (line 5), it makes sure that the estimated amount of model parameters exceeds a threshold which is taken from an input for this purpose. If satisfied, it only allocates the

maximum of the partitioned parameters, rather than whole parameters assigned to the current layer. As such, the size of the optimized memory pool can be kept small and the utilization of the optimized memory pool space can be high throughout the execution.

6.3 On-demand Parameter Loading

In contrast to dynamically generated IAs, model parameters are in a form of pre-generated values (by training) stored in a model file. Since the current memory management of a DL program has a bulk allocation of memory buffers, loads all model parameters from the file at once, and keeps the buffers committed for a long time (§5), it does not fit into our optimized memory usage plan. In our system, we address this by loading the parameters in an *on-demand* fashion. Because model parameters are assigned to a specific layer and they are only used once during the computation of the corresponding layer (and never used again), layers can time-share a small memory region in the EPC and avoid occupying a large EPC space for all parameters. Using the parameter information of each layer in the layer dependency graph, a vessel identifies the location of the parameters assigned to each layer in the file and loads them at the beginning of the corresponding layer in an OCALL. The parameters are loaded into the memory buffer in the optimized memory pool, and thus only present in the EPC for an essential period of time. Our approach requires an OCALL for each layer thereby triggers a more number of enclave exits than the *all-at-once* approach. However, we discovered through experiments that the all-at-once approach causes a much higher overhead for parameter loading than our on-demand parameter loading. Specifically, the secure paging overhead caused by the large parameter buffers overwhelms the overhead caused by the OCALLs.

Transposed parameters. In some models (e.g., VGG16), model parameters require a transpose before they are used by the layers. A transpose requires an additional memory space to store the outcome of the computation, in addition to the parameter buffer, resulting in a 2x memory commitment for the parameters. In our system, the memory usage planning identifies whether each layer requires a transpose on the model parameters (using the information in the layer dependency graph), and generates an additional plan to allocate a memory buffer for the transposed parameters in the optimized memory pool. The optimized lifespan and size of this buffer is equal to that of the original parameters.

6.4 EPC-aware Prediction Scheduling

A production DL system often receives a large number of prediction requests that require scalable processing. In a computing environment with multiple cores, such a workload can be scheduled into parallel processes to achieve a high scalability. However, such parallelism causes a high congestion on the EPC (i.e., EPC thrashing) and degrades the performance in a DL system with SGX, since all enclaves in the physical machine have to compete for a single small EPC region. Although our system alleviates this issue by reducing the memory footprint for an individual prediction request, it may suffer from the scalability issue when multiple vessels are launched and the EPC is filled up. We address this problem by a prediction scheduling mechanism that considers the current usage of the EPC. Algorithm 2 presents our scheduling mechanism.

Algorithm 2 EPC-aware prediction scheduling

```
1:  function predictJobSchedule(G, T)
2:      cur ← 0                                   ▷ Current memory occupation
3:      P, Size ← getOptimizedMemoryBuffers(G)    ▷ Algorithm 1
4:      Size_t ← getTotalMemorySize(Size)
5:      while True do
6:          sleep until an event e is received.
7:          if isJobReceived(e) then
8:              j ← getJobRequest
9:              if T > cur + Size_t then
10:                 e ← createEnclave(P, Size)
11:                 cur ← cur + Size_t
12:                 e.execJobInEnclave(j)
13:             else
14:                 enqueue(j)
15:         else if isCpuReleased then
16:             e ← getEnclave
17:             if isQueuedJob then
18:                 j ← dequeue
19:                 e.execJobInEnclave(j)
20:             else
21:                 destroyEnclave(e)
22:                 cur ← cur − Size_t
```

Our mechanism takes the upper bound of total committed memory to EPC as an input threshold (T). This threshold is determined by a single run of an experiment that finds the best performance point for a given model (§8.3). For every prediction request, our mechanism checks if scheduling a new prediction will violate the threshold (and thus cause EPC thrashing). It uses the memory usage estimation of the running vessels to calculate the current usage of the EPC (line 9) and launches a new vessel only if the new usage will not exceed the threshold (lines 10-12). If either the prediction will not fit into the memory or all cores are currently occupied, then it adds the request into the FIFO queue and waits until a vessel terminates after the prediction (line 14). Upon the termination of a vessel, a new vessel is launched and a request from the queue is scheduled into it (lines 17-19).

7 IMPLEMENTATION

We implemented Vessels using the Darknet neural network framework [46] as the basis, for its portability and flexibility. Note that the architecture of Vessels is agnostic to Darknet and applicable to various DL platforms. Pre-trained models by other DL frameworks can be converted into Darknet models for our system [1]. Since the baseline version of Darknet is incompatible with SGX, we ported its source code to an SGX-compatible form based on the Intel SGX SDK version 1.5 for Linux. We measured the number of secure page swaps by modifying the SGX device driver. For the ease of deployment in a cloud infrastructure, we packaged Vessels in a Docker container [39] and performed experiments using this container.

8 EVALUATION

In this section, we evaluate the efficiency and scalability of Vessels with a single enclave and multiple concurrent prediction enclaves.

8.1 Experimental Setup

Our experiments are performed on a machine with an Intel Core i7-6700 3.40GHz CPU and 32 GB RAM running Ubuntu 16.04 with Linux kernel of version 4.8.0-36-generic. We configure the BIOS to setup 128 MB for EPC region. As we present in §5, we employ 9 popular pre-trained models [26, 28, 33, 46, 47, 51, 52, 58], and ImageNet [48] as the sources of prediction dataset. For the models that Darknet framework does not support by default, we either generate them from scratch through our own training process or a model conversion from other DL frameworks (e.g., Caffe [31]).

8.2 Single Prediction Enclave

We first evaluate the effectiveness of Vessels for a single prediction in terms of three aspects: the latency of a prediction, memory consumption, and the number of occurrence of secure paging. To begin with, Vessels is deployed within an enclave, as described in §6.2, taking an image as an input, then makes a prediction for each DL model.

We compare the performance result of Vessels with a straw-man SGX system (Table 4). Overall, Vessels significantly outperforms Straw-man for all the target models, in both the memory footprint and runtime overhead. The overhead induced by our memory usage planning is 0.7s on average (included in the execution time in Table 4), which is a one-time cost per model and therefore negligible. Regarding the EPC usage, Vessels achieves much less memory footprint than Straw-man; Vessels consumes memory less ($\leq$ 59 MB) than the EPC limit (93.5 MB) for the DL models except VGG16 and YoloV3. As expected, such an optimized memory footprint causes significant reduction of paging; all memory buffers used during the prediction remain within the EPC region, thus none of paging happens in most cases, rendering Vessels to complete the prediction job much earlier than Straw-man (even at a near native speed).

In both VGG16 and YoloV3, however, their peak memory exceed the EPC limit even after the significant improvement (86% and 73%) over the original memory usage. According to our analysis, an auxiliary buffer to support a convolutional layer still takes up over 100 MB. That remains unoptimized in memory as the memory pool accommodates solely intermediate activations and model parameters in our design[3]. Note that a larger memory pool is required for YoloV3, whose convolutional layers operate upon a much larger volume of IAs according to its hyper-parameter.

We also compare the performance of Vessels with the recent SGX systems [2, 34, 43] described in §5. In our experiment, we run InceptionV3 on the three SGX systems and Vessels for a comparison – TensorSCONE and TF Trusted require a modification to DL model and have limited the number of models we could run on all four systems. In our result, Vessels shows an overhead of 1.97x compared to an unprotected run, outperforming Eleos [43] and TF Trusted [2] significantly (19-82x improvement in latency) and TensorSCONE [34] by 1.2x improvement (Table 1). It is noteworthy that Vessels offers more efficient DL prediction than TensorSCONE and TF Trusted while the systems uses a feature-reduced version of the DL model (with quantized parameters) in expense of lower accuracy. Moreover, Vessels improves prediction efficiency further when concurrent enclaves are to handle many prediction requests (§8.3), which is out of scope of these existing systems.

In summary, Vessels achieves a significant reduction of memory over the original memory usage, ranging from 73% to 91%. As a result, it reduces the prediction latency by 18%-94% and 1.2-82x compared to Straw-man and the recent SGX systems, respectively.

8.3 Concurrent Prediction Enclaves

We extend our experiment to evaluate Vessels to handle many prediction requests with multiple prediction enclaves that run in parallel. For high throughput, it would be desired to handle as many requests as possible in parallel in a multi-core computing environment. However, the size of the EPC does not scale with the number of enclaves with SGX. Launching a new enclave for every prediction request causes an excessive number of secure page swaps and leads to an EPC thrashing.

To test how Vessels improves the performance of concurrent prediction enclaves with the memory optimization and EPC-aware scheduling respectively, we examine the number

[3]We leave further improvement on this as future work.

Models	Peak Memory Performance		VESSELS Memory Pool	Execution Time Performance		Secure Paging	
	Peak Memory in VESSELS	Reduction from STRAW-MAN		Time	Improvement over STRAW-MAN	# of Swaps	Reduction from STRAW-MAN
AlexNet	29 MB	89.5%	8 MB	1.29s	94.01%	0	100%
ResNet101	38 MB	88.1%	23 MB	7.34s	69.81%	0	100%
ResNet152	39 MB	91.2%	23 MB	10.77s	66.90%	0	100%
DenseNet201	42 MB	88.8%	19 MB	4.71s	63.02%	0	100%
ResNext152	59 MB	89.6%	41 MB	11.04s	69.47%	0	100%
DarkNet53	53 MB	80.6%	30 MB	7.2s	69.37%	0	100%
InceptionV3	49 MB	85.4%	18 MB	16.44s	57.44%	0	100%
VGG16	156 MB	86.1%	34 MB	50.12s	57.44%	1.86 M	50.53%
YoloV3	225 MB	73.2%	85 MB	132.18s	18.89%	3.48 M	17.92%

Table 4: Performance of a single prediction enclave in VESSELS compared to STRAW-MAN.

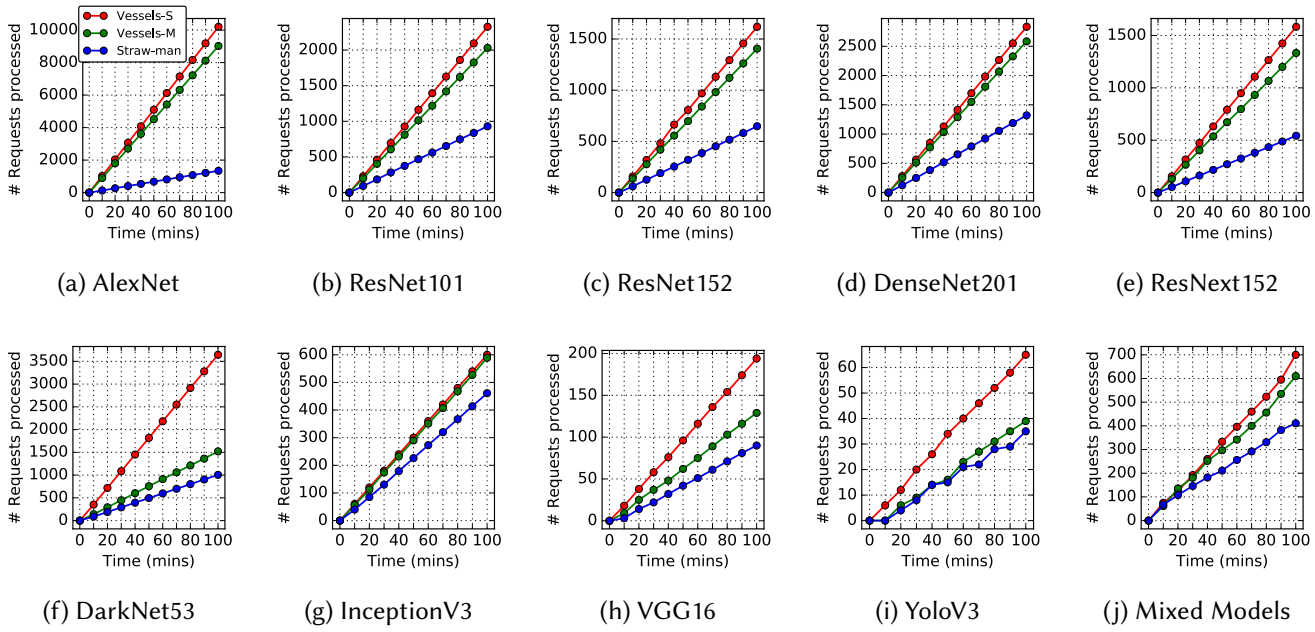


Figure 6: Number of prediction requests handled over time by concurrent enclaves. `Vessels-S` and `Vessels-M` exhibit the result for VESSELS with and without EPC-aware scheduling, respectively. `Mixed Models` is an average result of multiple runs with randomly selected DL models.

of prediction requests that three systems handle over time: `Straw-man`, `Vessels-M`, and `Vessels-S`. Figure 6 presents the result of the three systems with concurrent enclaves. Our experiment issued an unbounded number of prediction requests to the three systems for 100 minutes. As a baseline, we ran STRAW-MAN on 8 fixed number of concurrent enclaves, equal to the number of processor cores, without considering EPC usage (`Straw-man`). Similarly, `Vessels-M` is run on the fixed number of concurrent enclaves but it employs the memory optimization of VESSELS. `Vessels-S`, on the other hand, employs both the memory optimization and EPC-aware prediction scheduling. We used per-model memory thresholds ranging from 100 MB to 250 MB in our experiments.

As shown in the figure, the number of processed requests show a mostly linear growth for all models, but with a more significant growth rate with `Vessels-M` and `Vessels-S` over time. As a result, the improvement that `Vessels-M` and `Vessels-S` bring over `Straw-man` is substantial after 100 minutes of processing (an average of 131% for `Vessels-M` and 195% for `Vessels-S`) as shown in Figure 7.

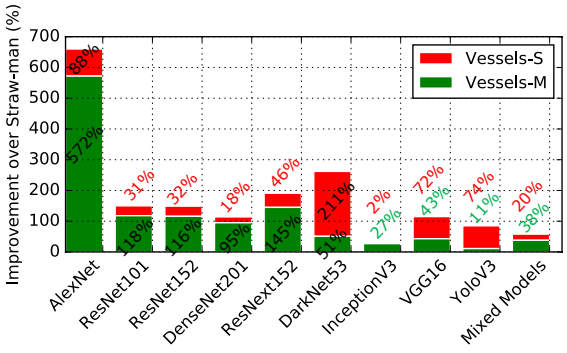


Figure 7: Performance improvement of Vessels over Straw-man. The improvement is calculated based on the number of prediction requests handled for 100 minutes.

Compared to `Vessels-M`, the use of EPC-aware prediction scheduling by `Vessels-S` further improved the performance of Vessels by 64% on average. We analyze that the performance improvement by `Vessels-S` is smaller for InceptionV3 (2%) compared to other models because of its high-density dependencies that cause much more frequent accesses to the EPC. Due to the large memory footprint even after the memory optimization, `Vessels-M` did not improve the performance significantly with YoloV3, in comparison with other models. In `Mixed Models`, we used a randomly selected DL model among those 9 models for each request to demonstrate the performance of Vessels with diverse DL models in a shared cloud platform by multiple users/services.

Since the memory footprint differs across DL models, the corresponding memory constraint used by the EPC-aware prediction scheduling can be adjusted for the best performance. Based on our empirical study, we use per-model memory constraints, ranging from 100 to 250 MB. For `Mixed Models`, 250 MB is used to accommodate any DL model used in the experiment.

9 RELATED WORK

Deep learning in trusted execution environments. Deep learning is used in many security-sensitive applications such as face recognition and fingerprint scanning. Hardware-assisted protection techniques e.g., Intel SGX [18] and ARM TrustZone [44] have been proposed to provide a confidential computing environment for DL. Related work have been dedicated to the performance improvement due to resource restriction in the trusted execution environments [2, 34, 49, 54, 55, 55, 56]. TensorSCONE [34] and TF Trusted [2] utilize TensorFlow Lite [9] to reduce the size of a DL model and run prediction in an SGX enclave. Despite their effort to improve performance, they still suffer from massive secure paging caused by large vector computation or compromise the prediction accuracy by using quantization.

Peter et al. [56] has a similar goal as our work to improve performance by partitioning large matrix buffers, but their work targets embedded devices equipped with TrustZone. Privado [54] achieves a smaller TCB by placing partial DL specific libraries in an enclave. Unlike Vessels, however, they neither accomplish a suitable size of EPC (i.e., 128 MB) even for a single request, nor consider multiple prediction requests that would be a common prediction requirement in practice. SGX-BigMatrix [49] introduces an SGX-specific data analytic framework on which matrix computations operate in an efficient and secure way. While this work focuses on oblivious execution of general data analytic operations (e.g., sorting) with SGX, our work focuses specifically on the efficient and scalable execution of DL prediction with SGX. Slalom [55] enables an SGX-based DL system to securely offload certain operations to hardware accelerators. In comparison, Vessels addresses challenges in protecting the entire "in-house" execution of DL prediction without limiting the scope of the protection to specific (offload-able) operations.

Performance improvement of trusted execution environments. Beside DL systems running on trusted processors, there are other approaches that tackle general overhead issues caused by trusted processors. Eleos [43] provides a solution to circumvent costly enclave transitions, by deploying a virtual address space atop of existing Virtual Memory (VM) and asynchronous system calls. However, as shown in §5, extra address translations within a user-level enclave are too expensive, and becomes crucial drawbacks to DL workloads. VAULT [53] particularly focuses on improving the overhead of integrity checks during enclave memory accesses, and it is achieved through decent data structures and compression techniques. Unfortunately, such architectural approaches involve deployment issues. Glamdring [35] aims at performing source code level partitioning through various program analysis techniques. However, such a code-centric approach is rarely helpful for the improvement of data-driven programs such as DL programs.

Resource optimization of deep learning. A large body of research work in the literature focuses on the improvement of DL system performance. Model-based optimization is a common practice to achieve this. Quantization techniques [9, 30] change the original model parameters into a lightweight form. Model pruning techniques [20, 24, 25] reduce parts of a large volume of parameters that unlikely affect the intermediate activations and final classification. Although these techniques are different, both suffer from affecting classification accuracy in common because of the loss

of information in the original model. It is worth noting that these techniques can be applied to our scheme, improving Vessels' performance further (at the expense of accuracy).

Model compression [57] is another type of optimization, in which parameter data is recorded in a compressed form. In spite of their lossless nature, it does not help reduce the total runtime memory consumption, as the decompressed data should be placed in a secure memory region on their use. AWS SakeMaker Neo [10] is a recent technology by Amazon to improve DL prediction performance. Orthogonal (and thus complementary) to Vessels, it focuses on platform-specific optimization of DL prediction during compilation while Vessels focuses on memory optimization and memory budget-aware scheduling of confidential DL prediction tasks.

10 DISCUSSION

Fragmentation of the memory pool. Although Vessels keeps the size of the memory pool small for most DL models in our experiments, the amount of memory required to execute each layer varies across different layers in a DL model. As such, the size of the shared memory pool is determined by the layer with the largest memory consumption in our system, leading to fragmentation of the memory space for layers with smaller memory consumption. One way to address this is to partition each layer into small sub-layers such that the size variations across all sub-layers become negligible. We leave this as our future work.

Multi-process vs multi-thread prediction. In our implementation and evaluation, we use multiple separated enclaves based on the multi-process design, rather than multi-threading mainly for two reasons. First, in a cloud environment, multiple requests may belong to different users. Second, from the security perspective, it can offer better isolation with separated memory address space. Nevertheless, multi-thread scheduling may offer better performance, although it has potential race conditions to be addressed.

GPU acceleration. By default, SGX does not support GPU-assisted acceleration, and our work focuses on CPU-only computation environment accordingly. Nonetheless, there are continuing efforts to take advantage of GPUs to speed up SGX execution [55], which are complementary to our technique.

Expanded enclave memory. Intel has recently announced a newer version of SGX (SGX2 [38]) which allows a larger EPC size. The Intel SGX Card [15] has also been announced that it allows additional EPC (up to 128 MB per card) and trusted processors by plugging it into a PCIe slot of a server machine. Although these technologies alleviate to a certain extent the memory problem of DL prediction, we argue that they do not fundamentally solve this problem as the EPC must be limited to a certain size regardless (e.g., multiple GB). For a production DL prediction system with SGX that receives many requests for a short time period, multiple GB of EPC may still be insufficient and require secure paging to host many concurrent enclaves due to their large memory footprints. In this regard, the design of Vessels is applicable to future SGX technologies with a larger EPC capacity to improve the performance and scalability of a confidential DL prediction system.

11 CONCLUSION

Despite the strong security that Intel SGX provides, current DL prediction systems with SGX suffer from significant performance overhead and scalability issues due to memory-intensive DL computation. In light of this problem, we have conducted a systematic study on the current DL prediction systems and discovered a paradigm of inefficiency that causes the issues. Our findings in turn enabled us to design a novel system, Vessels, that overcomes the limitation and provides highly efficient DL prediction with full SGX protection. In our experiments, Vessels eliminated around 90% of the memory footprint and reduced the prediction latency by an average of 58% compared to a baseline SGX system which has no functionality and accuracy loss. Our evaluation with multiple concurrent enclaves showed that Vessels can handle practical workload with high scalability (195% higher throughput than the baseline on average), demonstrating its usability in production DL prediction on the cloud.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their valuable feedback. This work was largely done while Kyungtae Kim was an intern under the supervision of Chung Hwan Kim at NEC Laboratories America. Chung Hwan Kim is the corresponding author of this paper. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of their employers.

REFERENCES

[1] 2017. Deep Learning Model Convertors. https://github.com/ysh329/deep-learning-model-convertor.
[2] 2017. TF Trusted. https://github.com/dropoutlabs/tf-trusted.
[3] 2017. Top 5 Cloud Security related Data Breaches! https://www.cybersecurity-insiders.com/top-5-cloud-security-related-data-breaches/.
[4] 2018. Asylo: An open and flexible framework for enclave applications. https://asylo.dev/.
[5] 2019. Deep Learning on AWS. https://aws.amazon.com/deep-learning/.
[6] 2019. Deep Learning VM | Google Cloud. https://cloud.google.com/deep-learning-vm/.
[7] 2019. Human Error Often the Culprit in Cloud Data Breaches. https://www.wsj.com/articles/human-error-often-the-culprit-in-cloud-data-breaches-11566898203.

[8] 2019. Machine Learning Service | Microsoft Azure. https://azure.microsoft.com/en-us/services/machine-learning-service/.

[9] 2019. TensorFlow Lite. https://www.tensorflow.org/lite.

[10] 2020. AWS SageMaker Neo. https://aws.amazon.com/sagemaker/neo/.

[11] 2020. IBM Cloud Data Shield. https://www.ibm.com/cloud/data-shield.

[12] 2020. MS Azure Confidential Computing. https://azure.microsoft.com/en-us/solutions/confidential-compute/.

[13] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. 265–283.

[14] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. Savannah, GA, 689–703.

[15] Somnath Chakrabarti, Matthew Hoekstra, Dmitrii Kuvaiskii, and Mona Vij. 2019. Scaling Intel Software Guard Extensions Applications with Intel SGX Card. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '19)*.

[16] Chia che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC '17)*. Santa Clara, CA, 645–658.

[17] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2019. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 142–157.

[18] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016, 086 (2016), 1–118.

[19] Tu Dinh Ngoc, Bao Bui, Stella Bitchebe, Alain Tchana, Valerio Schiavoni, Pascal Felber, and Daniel Hagimont. 2019. Everything You Should Know About Intel SGX Performance on Virtualized Systems. In *Abstracts of the 2019 SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '19)*. 77–78.

[20] Jonathan Frankle and Michael Carbin. 2018. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635* (2018).

[21] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. 2015. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1322–1333.

[22] Matthew Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. 2014. Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing. In *23rd USENIX Security Symposium (USENIX Security '14)*. 17–32.

[23] Zhongshu Gu, Heqing Huang, Jialong Zhang, Dong Su, Hani Jamjoom, Ankita Lamba, Dimitrios Pendarakis, and Ian Molloy. 2018. YerbaBuena: Securing Deep Learning Inference Data via Enclave-based Ternary Model Partitioning. *arXiv preprint arXiv:1807.00969* (2018).

[24] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained qare two popular schemesouantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[25] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*. 1135–1143.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[27] Sanghyun Hong, Pietro Frigo, Yigitcan Kaya, Cristiano Giuffrida, and Tudor Dumitras. 2019. Terminal Brain Damage: Exposing the Graceless Degradation in Deep Neural Networks Under Hardware Fault Attacks. In *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA, 497–514.

[28] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4700–4708.

[29] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2017. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. *arXiv preprint arXiv:1712.05877v1* (2017).

[30] 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 *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2704–2713.

[31] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093* (2014).

[32] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*.

[33] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

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

[35] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. 2017. Glamdring: Automatic Application Partitioning for Intel SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC '17)*. 285–298.

[36] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. 2018. Trojaning attack on neural networks. In *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS 2018)*.

[37] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. 11.

[38] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. 2016. Intel Software Guard Extensions (Intel SGX) Support for Dynamic Memory Management Inside an Enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016 (HASP '16)*.

[39] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux journal* 2014, 239 (2014), 2.

[40] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based Fault

Injection Attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy (SP '20)*.

[41] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious multi-party machine learning on trusted processors. In *25th USENIX Security Symposium (USENIX Security '16)*. 619–636.

[42] Meni Orenbach, Andrew Baumann, and Mark Silberstein. 2020. Autarky: closing controlled channels with self-paging enclaves. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[43] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS services for SGX enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 238–253.

[44] Sandro Pinto and Nuno Santos. 2019. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Computing Surveys (CSUR)* 51, 6 (2019), 130.

[45] Minghai Qin, Chao Sun, and Dejan Vucinic. 2017. Robustness of Neural Networks against Storage Media Errors. (09 2017).

[46] Joseph Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. http://pjreddie.com/darknet/.

[47] Joseph Redmon and Ali Farhadi. 2018. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767* (2018).

[48] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252.

[49] Fahad Shaon, Murat Kantarcioglu, Zhiqiang Lin, and Latifur Khan. 2017. SGX-BigMatrix: A practical encrypted data analytic framework with trusted processors. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1211–1228.

[50] Mahmood Sharif, Sruti Bhagavatula, Lujo Bauer, and Michael K Reiter. 2016. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1528–1540.

[51] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[52] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2818–2826.

[53] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. 2018. VAULT: Reducing paging overheads in SGX with efficient integrity verification structures. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 665–678.

[54] Shruti Tople, Karan Grover, Shweta Shinde, Ranjita Bhagwan, and Ramachandran Ramjee. 2018. Privado: Practical and secure DNN inference. *arXiv preprint arXiv:1810.00602* (2018).

[55] Florian Tramer and Dan Boneh. 2018. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. *arXiv preprint arXiv:1806.03287* (2018).

[56] Peter M VanNostrand, Ioannis Kyriazis, Michelle Cheng, Tian Guo, and Robert J Walls. 2019. Confidential Deep Learning: Executing Proprietary Models on Untrusted Devices. *arXiv preprint arXiv:1908.10730* (2019).

[57] Simon Wiedemann, Klaus-Robert Müller, and Wojciech Samek. 2019. Compact and computationally efficient representation of deep neural networks. *IEEE transactions on neural networks and learning systems* (2019).

[58] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. 2017. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1492–1500.