AegisDNN: Dependable and Timely Execution of DNN Tasks with SGX

Yecheng Xiang, Yidi Wang, Hyunjong Choi, Mohsen Karimi, and Hyoseung Kim
University of California, Riverside
yxian013@ucr.edu, ywang665@ucr.edu, hchoi036@ucr.edu, mkari007@ucr.edu, hyoseung@ucr.edu

***Abstract*—With the rising demand for emerging DNN applications in safety-critical systems, much attention has been given to the reliability and trustworthiness of DNN inference output against malicious attacks. Although prior work has been conducted to improve the privacy of DNN inference by executing the entire DNN model inside Intel SGX enclaves, existing approaches pose severe performance challenges to achieve dependable and timely execution simultaneously. In this paper, we propose AegisDNN, a DNN inference framework to address this problem. AegisDNN leverages secure SGX enclaves for protecting only the critical part of real-time DNN tasks which are vulnerable to potential fault injection attacks. To choose the right set of layers for protection while ensuring the timeliness of task execution, AegisDNN includes a dynamic-programming based algorithm that finds a layer protection configuration for each task to meet the real-time and dependability requirements based on the layer-wise DNN time and SDC (Silent Data Corruption) profiling mechanism. AegisDNN also utilizes a machine-learning based SDC prediction method to significantly reduce the time for estimating SDC rates for all possible layer protection configurations. We implemented AegisDNN on Caffe, PyTorch, and Tensorflow with Eigen BLAS ported into SGX enclaves to comprehensively demonstrate the effectiveness of AegisDNN against state-of-the-art DNN fault-injection attacks. Experiment results indicate that AegisDNN could satisfy both dependability and real-time requirements simultaneously, when none of the other compared approaches could do so.**

I. Introduction

Deep neural networks (DNNs) are becoming an essential element in safety-critical applications with intelligent autonomy. In systems like self-driving cars and smart robotics, various DNNs are used collectively to achieve intelligent features such as object detection, scene recognition, and natural language processing. As DNNs are increasingly involved in autonomous decision-making processes, a faulty or late output may lead to catastrophic behavior, which is unacceptable in safety-critical domain [19, 40, 45]. Therefore, it is an emerging challenge to ensure highly dependable real-time execution of DNN tasks with limited computing resources.

Contrary to the common belief that DNNs are inherently robust to errors, recent studies have demonstrated that DNNs, particularly those for image classification, are vulnerable to adversarial *fault injection* attacks [9, 25, 34, 35]. Since the output of DNNs is determined by input data, trained weight values, and intermediate results, manipulating some of these parameters can easily lead to misclassification. Physical laser beam [7] or software-based row hammer attacks [34, 43] are shown to be successful in flipping specific DNN parameters in memory. To ensure dependability against such attacks, it is imperative to protect the parameters critical to output correctness from malicious data modifications. The leakage of the critical parameters including data structures and location in memory should also be prevented because such information is required by fault injection attacks to analyze the weakness of DNN models. Even if attacks happen, the degree of misclassification should be contained within a permissible and predictable range.

To achieve those requirements, we leverage the Trusted Execution Environment (TEE), specifically Intel's Software Guard eXtension (SGX) [3] in this work. SGX provides *secure enclaves* to ensure the integrity and confidentiality of data and code from untrusted users even if the underlying OS or DRAM hardware is compromised. Any malicious attempt to modify data and code within enclave memory space is strictly prevented by architecture support. With these benefits, prior work on DNN cloud offloading has utilized SGX to address privacy issues by running the entire DNN model inside an enclave [10, 12, 24]. However, we cannot simply adopt the same approach to our problem due to the significant performance penalty of SGX. Up to 40× slowdown is observed compared with running on a GPU (see Sec. III for details), which is detrimental to real-time tasks. Concurrent execution of multiple DNN tasks in enclaves further deteriorates performance due to the limited memory size of SGX.

This paper presents AegisDNN to address the aforementioned *real-time* performance challenges while ensuring the *dependability* that we define as the reliability and integrity of DNN classification output. The key idea of AegisDNN is to find out critical subsets of DNN layers for the requested level of dependability and execute only them on enclaves for protected execution with a minimal performance penalty. To achieve this, AegisDNN first analyzes the fault sensitivity of each DNN layer, quantifying the degree of misclassification when the layer is not protected. Based on this information, AegisDNN determines the layer protection configurations of a given taskset to satisfy both real-time performance and dependability requirements. It is worth noting that AegisDNN does not conflict with the techniques proposed in prior work [24, 38, 40, 44] so that they can be accommodated in our framework to further speed-up DNN task execution.

We have implemented AegisDNN on Caffe [1], PyTorch [30] and TensorFlow [4] with Linux SGX SDK [3] on an Intel x86 platform. The performance of AegisDNN has been evaluated for tasksets using popular DNN models against three different state-of-the-art fault injection attacks [9, 25, 34]. Experiment results show that AegisDNN significantly improves the dependability of DNN output (e.g., as much as 88.8% lower misclassification rates than unprotected GPU execution) with minimal real-time performance loss (e.g., only 0-1.7% of deadline misses when fully-protected SGX execution causes 96.5% or higher misses).

Contributions. Below summarizes our contributions:

- To the best of our knowledge, AegisDNN is the first work to leverage SGX for protecting only the critical parts of real-time DNN tasks against fault injection attacks. It is designed amenable to formal real-time schedulability analysis, by accessing SGX as a mutually-exclusive resource and partitioning layers with real-time performance in mind.
- We propose a dynamic programming (DP) based algorithm that balances between fault rate and utilization demand to find the best protection configuration for each task. It has a polynomial time complexity but still finds solutions comparable to those of exhaustive search in our experiments.
- As the number of combinations of layer-wise protection for each DNN task is enormous, we present a machine-learning based approach to reduce the fault injection time by predicting task-level fault rates under different combinations.
- We performed a thorough evaluation of AegisDNN on a real platform under various state-of-the-art fault injection attacks [9, 25, 34]. AegisDNN could satisfy both dependability and real-time requirements when *none* of the existing approaches could do so.

II. Related Work

DNN Fault Attacks. Recently, much work has focused on the feasibility and implications of fault injection attacks on DNNs. DeepLaser [7] is a physical attack using laser beam to perturb hidden DNN layers in embedded devices. Software-based bit-flip attacks [26, 34, 43] using the rowhammer technique [20] have also been developed. These attacks aim to achieve misclassification by injecting a relatively small amount of errors. However, they commonly assume that attackers have the full knowledge of victim DNN models (e.g., model structure and parameters) and the access to critical locations, which our work prevents. TensorFI [25], BinFI [9], and Ares [35] are fault injection tools that analyze DNN models and identify critical bits, and they are useful not only to assess the impact of soft errors but also to guide attack strategies. Bit-flip attack (BFA) with progressive bit search [34] presents a powerful attack method that can completely malfunction a DNN model by very few bit flips.

SGX and Trusted Execution. SGX has been used for DNNs primarily to secure user privacy, e.g., confidentiality of input image, when DNN execution is offloaded to untrusted cloud systems. Notable prior studies are [10, 12, 24], all focusing on privacy attacks. Specifically, Occlumency [24] includes techniques to reduce memory usage during DNN inference in SGX, which is orthogonal to our work and can be applied to AegisDNN to improve enclave execution speed. Privado [12] offers enhanced privacy via model transformation which incurs additional runtime costs. They both can be used for fault-injection attacks since they execute the entire DNN in enclaves for a complete protection. However, this approach remains impractical as the inherent performance overhead of SGX is still significantly high compared to GPUs and hardware accelerators. Serdab [10] focuses on the privacy of video processing DNNs and partitions DNN models across distributed enclaves on multiple machines to reduce overall latency. While the idea of partitioning DNNs is similar to ours, Serdab cannot be used against fault-injection attacks because it executes only the first few layers in enclaves and then the rest layers in untrusted CPU/GPU, making them vulnerable to attackers. It also limits the switch from enclave to untrusted CPU/GPU to at most once per DNN, and does not consider contention on trusted/untrusted resources. On the other hand, AegisDNN chooses what layers to protect considering real-time and dependability requirements (not just initial layers), and the switch between GPU and enclave can happen multiple times while taking into account the resulting costs.

While these approaches focus on DNN inference, Chiron [15] focuses on training models within enclaves to conceal training data from cloud service providers. Slalom [38] offloads linear operations of DNNs to untrusted GPUs and then verifies the correctness of results in enclaves. This approach is applicable to speed-up the protected layers in AegisDNN. The idea of providing secure enclaves on GPUs has been studied [39], but it comes with extra overhead and is still not available on today's hardware.

Real-Time DNN Execution. In the context of real-time systems, recent work [40, 41, 44] has focused on improving DNN inference latency and throughput. DART [40] proposes a scheduling framework to achieve pipelining and data parallelism on heterogeneous CPUs and GPUs. It offers deterministic timing guarantees to real-time tasks and increased throughput to best-effort tasks. S3DNN [44] presents a supervised stream scheduling approach for DNN workloads running on GPUs, and yields enhanced response time and throughput. Although these improves real-time performance, none of them protects DNN execution from fault injection attacks. Unawareness to such attacks makes the system to use incorrect inference output, which may cause catastrophic consequences. This is the problem our paper addresses.

III. Background and Motivation

A. SGX and Enclaves

SGX is a hardware-assisted security extension built into Intel architectures. It provides a software abstraction, called *enclave*, to build an isolated and secure execution region within the virtual address space of a user process. The code and data contents of enclaves are stored encrypted in the Processor Reserved Memory (PRM), which is not accessible by even privileged software like OS and thus ensures confidentiality. The pages of PRM are decrypted only within the CPU chip and checked for integrity before use at the architecture level. This enables SGX to protect enclaves against malicious software such as fault data injection and bit-flipping attacks from other

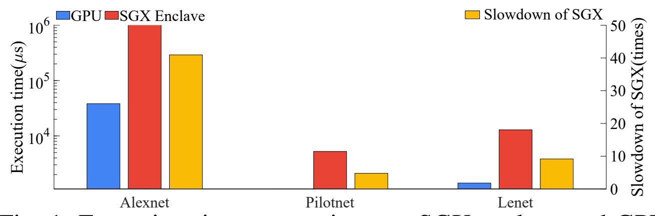


Fig. 1: Execution time comparison on SGX enclave and GPU

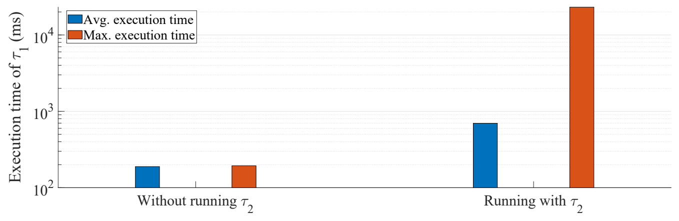


Fig. 2: Execution time of τ_1 when running in isolation vs when running concurrently with τ_2

applications, compromised OS, and low-level firmware. It can also protect against physical attacks on off-chip hardware, e.g., DRAM and memory bus, by detecting unwanted changes in enclave regions and preventing corrupted output generation. Such strong security properties allow trustworthy and dependable execution of programs even in hostile and harsh environmental conditions. We thus utilize SGX to enhance the dependability of DNN execution in this work.

B. Real-Time Performance Challenges

Although SGX provides strong protection mechanisms for DNN execution, it poses critical performance challenges. As shown in Fig. 1 with some popular DNN models, running the entire DNN model on the SGX enclave can result in a significant timing penalty (see Sec. VI for software and hardware setup). For Alexnet [21], it takes more than 40x time to execute on the enclave than on the GPU. For Pilotnet [6], which experiences the least performance slowdown among the three models shown in the figure, it is still 4.7x slower on the enclave compared to the GPU. Thus, it is practically infeasible to run real-time DNN tasks solely on enclaves.

There are three major reasons for such performance degradation. First, DNN models on the SGX enclave are executed by the CPU. The execution time of popular DNN models on CPUs is significantly slower than that on modern accelerators. Secondly, memory read and write speed is slower inside the enclave, which is critical for DNN inference requiring a large amount of memory operations. Such slowdown is mainly due to the extra steps taken by on-chip data encryption and integrity checks. Thirdly, the physical size of PRM under SGX is considerably small, e.g., 128MB for all currently supported Intel CPUs, far less than the memory needed for modern DNN models. SGX SDK on Linux supports paging to allow enclaves to have larger virtual space than the physical PRM size. However, frequent page swapping causes high overhead especially when executing large DNN models. In addition, when page swapping happens between the unprotected main memory and the protected enclave, additional encryption/decryption takes place to ensure the integrity of the swapped data.

The small SGX physical memory size is also the bottleneck when multiple DNN tasks need to execute concurrently. Memory thrashing may occur in such a scenario, further degrading the performance or even freezing the entire system. To identify the effect, we consider a simple taskset that has two memory-intensive tasks, τ_1 and τ_2, running on different cores. τ_1 allocates 30MB of memory in the enclave and repeatably reads all its memory pages in a sequential manner. τ_2 performs the same but with a larger memory size of 200MB. Fig. 2 depicts the average and maximum execution time of one iteration of τ_1 with and without the concurrent execution of τ_2, respectively. When τ_2 executes concurrently with τ_1 in the system, the execution time of τ_1 becomes 120x longer in the worst case and 3.7x longer on average. The exact moment when the task gets the worst-case slowdown is unpredictable due to the nature of the race condition between two tasks trying to page-swap at the same time. These problems need to be addressed in order for enclaves to be utilized for real-time DNNs.

IV. System Model

The system considered in this work is equipped with a GPU for accelerated DNN execution and a TEE for protected (secured) execution.[1] Intel SGX is particularly assumed for the TEE since it offers a minimum attack surface with architecture-assisted integrity check mechanisms. SGX maintains a memory region separated from the system's main memory. Hence, explicit data transmission between an enclave and main memory is required before and after a task executes DNN workload within the enclave. The GPU also has its own memory region, requiring data transmission for execution. The CPU consists of one or more identical processor cores on which tasks launch SGX and GPU operations.

To avoid unpredictable variations in execution time, we model the enclave e and the GPU g as *mutually-exclusive* shared resources, i.e., e and g are each protected by a distinct mutex lock and accessible by only one task at a time. In case of the GPU, lock-based synchronization [11, 31, 36] has been widely used to secure the worst-case execution time (WCET) which can be hampered when concurrent kernel execution is allowed [16, 29, 42]. We adopt this approach for the enclave as well to prevent the thrashing problem discussed in Sec. III. Hence, the entire SGX memory is exclusively available for the task accessing the enclave. If the task requires more memory than the SGX memory, page swapping may occur, but its performance effect can be captured as part of the WCET of that task and does not adversely affect the WCETs of other tasks.[2] Note that the limited memory size of SGX is *not* the motivation of our work. Even if a DNN can fit entirely in the SGX memory, it is not efficient to protect all layers since each layer has different vulnerability and performance characteristics, and not all layers need to be protected to achieve the required dependability.

[1]We only consider DNN inference in this paper. DNN training is not considered since it is typically done offline.

[2]This notion of temporal isolation is similar to that in memory reservation systems [18, 27].

We consider SGX as a building block in our work because it offers architecture-level support for checking the integrity of code and data stored in enclaves. However, we believe it is possible to apply our work to other TEEs, e.g., ARM TrustZone, by implementing software-based integrity checking at the entry and exit points of the TEE, but this will likely cause higher runtime overhead.

A. Task Model

We consider DNN tasks following the sporadic task model [28] with constrained deadlines. Each task uses one feed-forward DNN model, and each job of a task makes one inference request for classification. Task τ_i is characterized by:

$$\tau_i := (C_i, T_i, D_i, N_i, M_i)$$

- C_i: the WCET of a single job of τ_i in the absence of external temporal interference from other tasks
- T_i: the minimum inter-arrival time between jobs
- D_i: the relative deadline of each job
- N_i: the number of layers of the DNN model used by τ_i
- M_i: the DNN model used by τ_i

Each job of τ_i executes the layers of M_i in a sequential manner[3] Hence, C_i comprises a set of execution times corresponding to those layers. We use $L_{i,j}$ to denote the j-th layer of M_i. For a layer $L_{i,j}$, it can execute in either of the two modes, protected or unprotected, which affects the execution time of $L_{i,j}$. If $L_{i,j}$ executes in the unprotected mode, the weight parameters (if exist) and input data of the layer are first copied from unprotected CPU memory to unprotected GPU memory. Then the kernel corresponding to that layer is launched on the GPU. Finally, the output data is copied back to unprotected CPU memory when the GPU kernel finishes its execution. If $L_{i,j}$ executes in the protected mode, the corresponding program code and weight parameters are assumed to have been preloaded into the enclave's memory space with the sealing feature of SGX [3]. So only the input data needs to be explicitly copied to the enclave. Note that, since those preloaded data might have been paged out when $L_{i,j}$ starts execution in the enclave, the time to page-in them needs to be captured in the WCET of $L_{i,j}$.

To distinguish the WCET of $L_{i,j}$ in the protected and unprotected mode, we will use $C_{i,j}(d)$ where d indicates the device $L_{i,j}$ runs on, i.e., $d = e$ (enclave) in the protected mode and $d = g$ (GPU) in the unprotected mode. $C_{i,j}(d)$ is detailed with the following parameters:

$$C_{i,j}(d) := (C^{hd}_{i,j}(d), C^{e}_{i,j}(d), C^{dh}_{i,j}(d), C^{m}_{i,j}(d))$$

where d is either the enclave e or the GPU g.

- $C^{hd}_{i,j}(d)$: the maximum time for input data transmission before $L_{i,j}$'s execution on the device d
- $C^{e}_{i,j}(d)$: the WCET of $L_{i,j}$ on the device d
- $C^{dh}_{i,j}(d)$: the maximum time for output data transmission after $L_{i,j}$'s execution on the device d
- $C^{m}_{i,j}(d)$: the maximum time of miscellaneous operations required for $L_{i,j}$'s execution, e.g., device launch preparation

[3]This is valid for feed-forward DNNs which are most commonly used these days. Recurrent networks like LSTM will be considered in future work.

The WCET of L_i on the enclave e, $C^{e}_{i,j}(e)$, is assumed to include the time for reloading and swapping its own pages. Note that these parameters are also compatible with the models used in real-time GPU work [17, 31].

This work mainly considers soft real-time systems where avoiding an overload condition is important to minimize deadline misses [8]. Thus, our focus lies on how to partition DNN workloads on SGX and GPU so that dependability is achieved while preventing overloads. The design of AegisDNN, however, does not preclude its use in hard real-time systems as long as the underlying OS and device drivers provide a sufficient level of predictability. In order to check hard real-time schedulability, we assume that tasks are pre-allocated to processor cores with no runtime migration. This assumption is not necessarily required for soft real-time systems since our algorithm aims to optimize resource utilization which is bottlenecked by mutex locks for SGX and GPU.

B. Threat Model

The notion of dependability considered in this work is the capability to ensure the integrity of output generated by real-time DNN tasks in the presence of fault data injection attacks. Such attacks are often performed in a *stealthy* manner, so it is hard to detect them as long as output, despite corrupted, is generated. We do not consider other types of attacks, such as side-channel and denial of service attacks, and delayed task execution due to such attacks.[4]

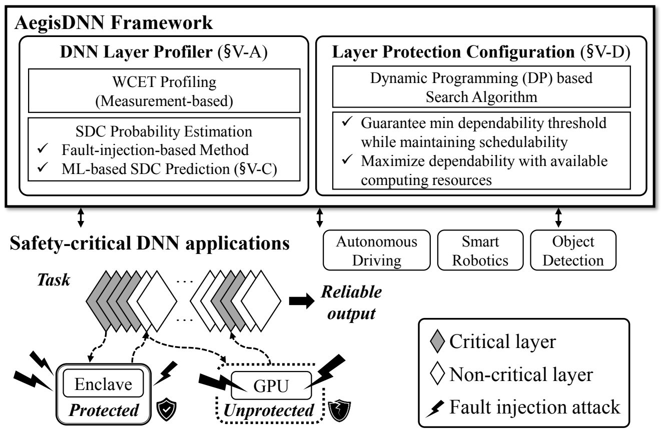
For the threat model, we follow the general model used for SGX-based systems in the literature [12, 24]. The CPU chip package, SGX features, and enclaves are trusted entities. Off-chip hardware components including DRAM, memory bus, and GPU are untrusted. Software components running outside enclaves, such as OS, device drivers, middleware, libraries, and other processes, are also untrusted. The enclaves of real-time DNN tasks are assumed to have been signed by the trusted authority. Attackers cannot launch their own enclaves due to the key management of the enclave launch procedure [3].[5] We assume that inter-enclave control-flow and data-flow integrities are ensured using existing techniques, such as Panoply [37].

Faults can be injected by software attacks [26, 34, 43] using rowhammer [20] or even by laser beam [7].[6] We assume the target system uses virtual memory and memory protection, which limits the severity of attacks, e.g., the attacker cannot perturb unlimited size of memory contents. Even if the OS is compromised, it is not trivial to directly modify the memory contents of other processes residing in a different virtual memory space. Regardless of their origin, the degree of attacks is quantified by using Bit Error Rate (BER), which is the number of faulty bits divided by the total number of memory

[4]If the system eventually hangs and no output is generated, the attacks are easily detectable and countermeasures can take place, e.g., a human operator taking over control of self-driving cars.

[5]This prevents attackers to cause the SGX thrashing problem discussed in Sec. III although this is not a primary attack we consider.

[6]Soft errors on memory devices could also be addressed by our work. Physical attacks on the CPU chip itself are beyond the scope of the paper.



AegisDNN Framework
DNN Layer Profiler (§V-A)
WCET Profiling (Measurement-based)
SDC Probability Estimation
✓ Fault-injection-based Method
✓ ML-based SDC Prediction (§V-C)
Layer Protection Configuration (§V-D)
Dynamic Programming (DP) based Search Algorithm
✓ Guarantee min dependability threshold while maintaining schedulability
✓ Maximize dependability with available computing resources
Safety-critical DNN applications
Autonomous Driving
Smart Robotics
Object Detection
Task
Reliable output
Enclave
Protected
GPU
Unprotected
Critical layer
Non-critical layer
Fault injection attack

Fig. 3: AegisDNN framework overview

bits. A higher BER indicates a task has a higher chance to experience faults.

For a given BER, we use Silent Data Corruption (SDC) probability as a metric to evaluate the dependability of a system, and the system designer gives the minimum dependability threshold that must be satisfied. In DNNs, there is often not just a single output, but a list of ranked outputs (labels) with a confidence score for each output. This work considers the correctness of the top-ranked label as it is usually most important. Therefore, we define the SDC probability as the probability of the top-ranked element of DNN output being different from that in the fault-free case.

V. The AegisDNN Framework

This section presents the key components of AegisDNN framework. Fig. 3 illustrates the framework overview. AegisDNN has a layer-wise DNN profiling mechanism (Sec. V-A) that not only obtains the WCET of each layer but also uses a fault-injection method to estimate the SDC probability of each layer when it is exposed to fault attack. Then, it uses a machine-learning approach (Sec. V-C) to predict SDC probabilities under *all* possible layer protection configurations, which significantly reduces the fault-injection time for profiling. Based on these profiles, AegisDNN runs a dynamic-programming based algorithm (Sec. V-D) to find the best feasible layer protection configuration for each task that allows satisfying both the dependability and schedulability requirements. At runtime, AegisDNN uses the configurations found by the algorithm to control the execution of DNN tasks.

A. Profiling Layer-wise WCET and SDC Probability

As discussed earlier that protecting all the layers of a DNN is expensive and impractical, AegisDNN chooses only the critical DNN layers to protect. Meanwhile, AegisDNN takes into account the performance cost caused by such protection to mitigate the scheduling penalty. To achieve both, AegisDNN generates a layer-wise DNN profile, which is composed of a WCET profile and an SDC probability profile.

1) WCET Profile: To understand the performance cost of protecting a DNN layer by executing it on the SGX enclave e compared to on the GPU g, the WCET needs to be estimated on both. AegisDNN takes a measurement-based approach for WCET estimation. All DNN models used in the taskset are executed n_{profile} times on both the enclave e and GPU g, where n_{profile} can be chosen by the user, and the execution time records are stored in the profile DB. AegisDNN estimates the WCET of each layer by taking the maximum among the observed execution time history. Note that the WCET estimation by this procedure may be violated when longer execution time is observed. Due to this limitation, AegisDNN claims to achieve soft real-time guarantees and to minimize the occurrence of deadline misses. For firm or hard real-time systems, one may add an error margin to the observed WCET, as done in practice, or apply more robust estimation methods, e.g., statistical approaches using Extreme Value Theory [5, 13].

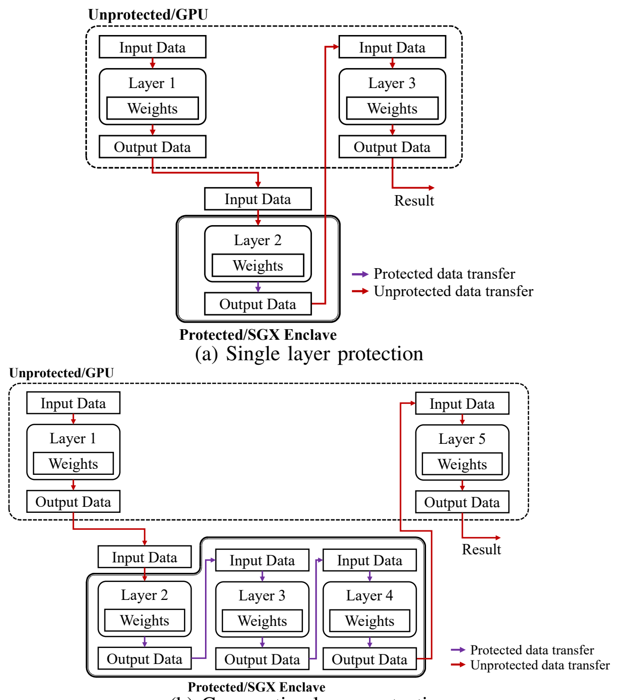
2) SDC Probability Profile: To understand the criticality for each DNN layer $L_{i,j}$ of task τ_i, AegisDNN profiles their SDC probability. The memory usage for computing each DNN layer can be classified into input data, weights parameters (if exist), and output data. As the output of the previous layer is directly used for the input of the next layer, we thus consider the SDC probability of only input and weight data. We use $sdc_{\text{in}}(i,j,b)$ to denote the SDC probability of the input data of layer $L_{i,j}$ under the BER of b, and $sdc_{\text{weight}}(i,j,b)$ to denote the SDC probability of the weight data, accordingly. The SDC probability of a layer $L_{i,j}$ is estimated by using existing fault injection attack methods [9, 25, 34, 35] onto the memory regions of input or weight data accordingly under the BER of b. The attacked bits are chosen based on the fault-injection attack method used. Next, AegisDNN executes the DNN inference and compares it with the results from fault-free condition. Finally, the $sdc_{\text{in}}(i,j,b)$ and $sdc_{\text{weight}}(i,j,b)$ are estimated by calculating the number of wrong outputs divided by the number of total fault injections when we only inject faults into the input and the weight data regions of layer $L_{i,j}$, respectively, while other layers remain fault-free.

B. Implications of Task-level Layer Protection

In this subsection, we discuss the implications of protecting different combinations of layers at the task level. We consider two cases of task-level protection: single layer protection and consecutive layer protection.

1) Single Layer Protection: As shown in Fig. 4a, when only a single layer $L_{i,j}$ of task τ_i is executed in the enclave e, only its weight data and the intermediate computation result can be protected. The input data of the layer $L_{i,j}$ is still vulnerable to fault as the output data of the previous layer $L_{i,j-1}$ is unprotected and fault can occur during its execution or memory copy. It is worth noting that we cannot protect the input data of the first layer due to this reason.

2) Consecutive Layer Protection: However, when multiple layers are consecutively protected, as shown in Fig. 4b, input data of all protected layers except the first layer (layer 2 in this example) are protected besides all weights and intermediate computation results. In other words, to protect the input data of a layer $L_{i,j}$ for task τ_i, both layer $L_{i,j-1}$ and layer $L_{i,j}$ have to be protected consecutively at the same time. Such



(a) Single layer protection

(b) Consecutive layer protection

Fig. 4: Comparison of layer protections

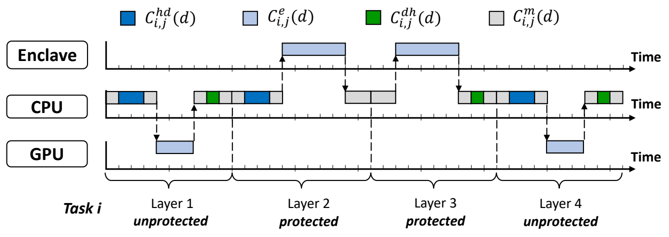


Fig. 5: Execution of consecutively protected layers

observation indicates that protecting consecutive layers enables stronger protection. Moreover, it requires less memory copy between the unprotected CPU/GPU memory and the protected enclave memory. Fig. 5 exemplifies the saved memory copy time for Task i where layers 2 and 3 are protected. The output data transmission of the layer 2 ($C^{dh}_{1,2}$) and the input data transmission of the layer 3 ($C^{hd}_{1,3}$) do not present in the timeline since those can reside within the enclave and do not need to be transmitted. AegisDNN takes into account such property in the machine-learning SDC prediction (Sec. V-C) and the configuration-finding algorithm (Sec. V-D) to improve system efficiency and dependability.

Task-level layer protection configuration. To better represent a sequence of protected and unprotected layers, we define the layer configuration of a task τ_i in binary notation as follows:

$$S_i^p = s_{i,N_i} s_{i,N_i-1} \ldots s_{i,2} s_{i,1}$$

The superscript p of S_i^p is the identifier of the configuration. $s_{i,j}$ represents the protection status of the j-th layer of τ_i, which is 1 if protected, and 0 otherwise. For example, if the 2-th and 3-th layers of a task are protected, the configuration value is equal to 0110.

C. Predicting Task-level SDC Probability

To find the balance between dependability and real-time performance, AegisDNN needs to choose the most critical layers to protect. Although the SDC probability has been already estimated for the case where only one layer of a DNN model is unprotected (Sec. V-A), it could not get the SDC probability when a combination of layers are simultaneously protected for a given task. To get a thorough SDC probability profile for all possible layer protection configurations, one may try to use an exhaustive-search approach by running fault injection for all configurations. However, for each DNN model M_i used in taskset, there are 2^{N_i} different configurations. It is not feasible to run exhaustive fault injections as the entire fault injection procedure required for the profiling takes enormous time to complete. To address this problem, we propose a machine-learning (ML) based approach to obtain a task-level SDC probability for a given layer protection configuration.

For a DNN model M_i used by task τ_i, AegisDNN first randomly generates n_{train} layer protection configurations $S_i^{p_1}, S_i^{p_2}, ..., S_i^{p_{n_{\text{train}}}}$, which are uniformly distributed among all configurations. AegisDNN then runs fault injection for each configuration, and collects its SDC probability. The results are used as the training data set $T = \{sdc(i, S_i^{p_1}, b), sdc(i, S_i^{p_2}, b), ..., sdc(i, S_i^{p_{n_{\text{train}}}}, b)\}$, where $sdc(i, S_i^p, b)$ denotes the SDC probability for a task τ_i with the layer protection configuration S_i^p under the BER of b. The parameter n_{train} controls the training data size for the ML algorithm and can be configured by the user. AegisDNN then converts each S_i^p into two bitmaps which represent the layer protection configuration of the input data and weights parameters, respectively. Each bit of these bitmaps is considered as an input feature for the ML algorithm. The formation of the first bitmap is $X_i^{weights} = \{x_{i,1}^{weights}, x_{i,2}^{weights}, ..., x_{i,N-1}^{weights}, x_{i,N}^{weights}\}$ in which each $x_{i,j}^{weights}$ is set if the weight data of the corresponding layer is protected. It also takes into account the effect of consecutively-protected layers by adjusting the value of bits in the second bitmap $X_i^{in} = \{x_{i,1}^{in}, x_{i,2}^{in}, ..., x_{i,N-1}^{in}, x_{i,N}^{in}\}$, where each bit indicates whether the input data of the corresponding layer is protected. Given that any two consecutively-protected layers make the input data of the second layer protected, the bit $x_{i,j+1}^{in}$ is set only if both $x_{i,j}^{weights}$ and $x_{i,j+1}^{weights}$ are 1.

For a given training data point of DNN model M_i, we first encode its layer protection configuration into the two bitmaps $X_i^{weights}$ and X_i^{in} as mentioned above. With the two bitmaps properly set, the relationship between SDC probability and layer configurations is modeled by multiple linear regression. The model we developed for SDC probability prediction can be represented in the following form:

$$\hat{y}_i = c_i + \sum_{j=1}^{N_i} \alpha_{i,j} x_{i,j}^{in} + \sum_{j=1}^{N_i} \beta_{i,j} x_{i,j}^{weights} \tag{1}$$

Here, $\hat{y}_i$ is the estimated SDC probability for a given task τ_i. c_i is a constant. $\alpha_{i,1}, \alpha_{i,2}, ..., \alpha_{i,N}$ and $\beta_{i,1}, \beta_{i,2}, ..., \beta_{i,N}$ are the coefficients of the respective predictor $x_{i,j}^{in}$ and $x_{i,j}^{weights}$.

N_i is the number of layers of τ_i. $\hat{y}_i$ equals to c_i ($c_i > 0$) if no layer is protected. Otherwise, the predicted SDC probability reduces since $\alpha_{i,j} \leq 0$ and $\beta_{i,j} \leq 0$. All data analysis are carried out using scikit-learn [32] in Python.

Our regression model can make precise estimations on SDC probabilities by capturing the interactions between protected layers as shown in Eq. (1). AegisDNN uses this model to obtain a complete set of SDC probabilities for each DNN model used by the taskset. We redefine $sdc(i, S_i^p, b)$ in the rest of the paper to denote the SDC probability predicted by the regression method for a task τ_i.

D. Finding Layer Protection Configurations

1) For Each Task: We first find a layer protection configuration for each task in the taskset with the following goals: (i) satisfy the minimum dependability threshold for each task, (ii) minimize the utilization demands of tasks, and (iii) maximize the dependability of the taskset by fully utilizing system resources. Since the dependability $\mathbf{D}$ is inversely proportional to the SDC probability p, it can be represented as $\mathbf{D} = 1 - p$.

While exhaustive search can be considered to find the protection configuration for each task, it is not a practical option. The number of layer protection configurations for a taskset Γ is $2^{\sum_{\tau_i \in \Gamma} N_i}$, which leads to an exponential time complexity for the search algorithm.

To reduce the time complexity, we first propose a dynamic-programming-based approach to find a layer protection configuration for a task τ_i that meets the given dependability threshold with the lowest utilization demand U_i. We define utilization of a task τ_i as the sum of execution time of all its layers over its period. Accordingly, the utilization $U_{i,j}$ of a layer $L_{i,j}$ of a task τ_i is given by $U_{i,j} = C_{i,j}(d)/T_i$.

We use $\mathbf{U}_z^{\mathbf{D}}[i,j,k]$ to denote the minimum utilization demand for the layers from $L_{z,i}$ to $L_{z,j}$ of a task τ_z when up to k disjoint subsequences of layers are protected and the minimum dependability threshold $\mathbf{D}$ is satisfied. So $\mathbf{U}_z^{\mathbf{D}}[i,j,1]$ means there is only one disjoint subsequence of protected layers. We store the corresponding layer protection configuration $\mathbf{S}_z^{\mathbf{D}}[i,j,k]$ for τ_z when calculating $\mathbf{U}_z^{\mathbf{D}}[i,j,k]$. Since memory copy only occurs at the boundaries of protected and unprotected layer segments, the user can configure k to limit such copy operations to improve system efficiency.

When k is equal to 1, the $\mathbf{U}_z^{\mathbf{D}}[i,j,1]$ is given by:

$$\mathbf{U}_z^{\mathbf{D}}[i,j,1] = \min \begin{cases} \mathbf{U}_z^{\mathbf{D}}[i+1,j,1] + C_{z,i}(g)/T_z, \\ \mathbf{U}_z^{\mathbf{D}}[i,j-1,1] + C_{z,j}(g)/T_z, \\ \begin{cases} \Big(\sum_{q=i}^{j} \big(C_{z,q}^{e}(e) + C_{z,q}^{m}(e) \big) + C_{z,i}^{hd}(e) + C_{z,j}^{dh}(e) \Big) / T_z \\ \qquad\qquad\qquad , \text{if } 1 - sdc(z, S_z^p, b) \geq \mathbf{D} \\ \infty \qquad\qquad\quad , \text{if } 1 - sdc(z, S_z^p, b) < \mathbf{D} \end{cases} \end{cases} \tag{2}$$

The first term in the min function, $\mathbf{U}_z^{\mathbf{D}}[i+1,j,1] + C_{z,i}(g)/T_z$, denotes the minimum utilization from layers $L_{z,i}$ to $L_{z,j}$ when the algorithm protects one section of consecutive layers among the layers from $L_{z,i+1}$ to $L_{z,j}$ ($\mathbf{U}_z^{\mathbf{D}}[i+1,j,1]$) and the layer $L_{z,i}$ is executed on the GPU g. Similarly, the second term in the min function, $\mathbf{U}_z^{\mathbf{D}}[i,j-1,1] + C_{z,j}(g)/T_z$, depicts the minimum utilization when it protects one section of consecutive layers among $L_{z,i}$ to $L_{z,j-1}$ and executes $L_{z,j}$ on the GPU g. The third term denotes the utilization under the protection configuration when the layers from $L_{z,i}$ to $L_{z,j}$ are all protected, i.e., $S_z^p = \sum_{q=i}^{j}(2^{q-1})$. We set it to infinite if S_z^p cannot satisfy the dependability requirement ($1 - sdc(z, S_z^p, b) < \mathbf{D}$). Otherwise, it represents the utilization under S_z^p. Finally, $\mathbf{U}_z^{\mathbf{D}}[i,j,1]$ is calculated as the minimum of the three terms.

When k is greater than 1, the recurrence is given as follows:

$$\mathbf{U}_z^{\mathbf{D}}[i,j,k] = \min \begin{cases} \mathbf{U}_z^{\mathbf{D}}[i,j,k-1] \\ \min\limits_{\substack{q \in [i+1, j-1] \\ r \in [1, k-1] \\ d \in [\mathbf{D}', \mathbf{D}]}} \begin{cases} \mathbf{U}_z^{d}[i,q,r] + \mathbf{U}_z^{d}[q+1,j,k-r] \\ \qquad\qquad , \text{if } 1 - sdc(z, S_z^p, b) \geq \mathbf{D} \\ \infty \qquad\quad , \text{if } 1 - sdc(z, S_z^p, b) < \mathbf{D} \end{cases} \end{cases} \tag{3}$$

In Eq. (3), S_z^p is the layer protection configuration resulted by both $\mathbf{S}_z^d[i,q,r]$ and $\mathbf{S}_z^d[q+1,j,k-r]$, i.e., $S_z^p = \mathbf{S}_z^d[i,q,r] + \mathbf{S}_z^d[q+1,j,k-r]$. Note that each of $\mathbf{S}_z^d[i,q,r]$ and $\mathbf{S}_z^d[q+1,j,k-r]$ already guarantees the dependability of the task to be higher than d. Hence, to find a configuration with the dependability higher than or equal to $\mathbf{D}$ in Eq. (3), we need to search the dependability d ranging from $\mathbf{D}'$ to $\mathbf{D}$, i.e., $\mathbf{D}' \leq d \leq \mathbf{D}$. By controlling $\mathbf{D}'$, the user can adjust the search space of the algorithm, which affects the execution time and ability of the algorithm to find the solution. For each case, AegisDNN checks $sdc(z, S_z^p, b)$ to ensure that the combined configuration S_z^p still meets the dependability requirement $\mathbf{D}$.

2) For All Tasks in a Taskset: We now propose our search algorithm in Alg. 1 to find the layer protection configuration for all tasks in a taskset. The goals of the algorithm are to: (i) satisfy the minimum dependability threshold for all tasks, (ii) ensure the schedulability of the taskset, and (ii) maximize the dependability of all tasks while maintaining the schedulability. The algorithm takes as input the taskset Γ, the minimum dependability threshold $\mathbf{D}$, the set of candidate k values $\mathbf{K_s}$, and the set of candidate dependability values $\mathbf{D_s}$. The values in $\mathbf{D_s}$ are greater than or equal to the requirement $\mathbf{D}$, and used to maximize the dependability of all tasks.

For each $\tau_i \in \Gamma$, the algorithm computes the minimum required utilization $\mathbf{U}_i^d[1, N_i, k]$ for all $d \in \mathbf{D_s}$ and $k \in \mathbf{K_s}$ values according to Eqs. (2) and (3) (line 7). It stores the corresponding configuration in $\mathbf{S}_i^d[1, N_i, k]$. Then, the algorithm checks the feasibility of the taskset under the configuration $\mathbf{S}^{\text{sol}}$ for the minimum dependability threshold $\mathbf{D}$ (lines 9-10). Here, feasibility means all tasks in the taskset satisfy both dependability and schedulability. If any of the tasks has $\mathbf{U}_i^{\mathbf{D}}[1, N_i, k_{max}] = \infty$, dependability is said to be unmet. Schedulability will be discussed in the later of this subsection.

If the taskset is not feasible, the algorithm returns an

Algorithm 1 Finding layer protection configuration of all tasks

```
Input: $\Gamma = \{\tau_1, \tau_2, \tau_3, ..., \tau_n\}$: taskset
Input: $\mathbf{D}$: minimum dependability threshold
Input: $\mathbf{D_s}$: a set of search dependability values including $\mathbf{D}$
Input: $\mathbf{K_s}$: a set of candidate $k$ values used in Eqs. (2) and (3)
Output: $\mathbf{S}^{\text{sol}} = \{S_1^{\text{sol}}, S_2^{\text{sol}}, ..., S_n^{\text{sol}}\}$: Solution layer protection configuration for each task; $\mathbf{S}^{\text{sol}} = \emptyset$, if failed.
1:  function FIND_SOLUTION($\Gamma$, $\mathbf{D}$,$\mathbf{D_s}$, $\mathbf{K_s}$)
2:      $\mathbf{S}^{\text{sol}} = \emptyset$ /* initialization */
3:      $k_{max} = \max(\mathbf{K_s})$
4:      for all $\tau_i \in \Gamma$ do
5:          for all $d \in \mathbf{D_s}$ do
6:              for all $k \in \mathbf{K_s}$ do
7:                  Compute $\mathbf{U}_i^d[1, N_i, k]$ by Eqs. (2) and (3)
8:                  Store $\mathbf{S}_i^d[1, N_i, k]$ accordingly
9:      $\mathbf{S}^{\text{sol}} = \{\mathbf{S}_1^{\mathbf{D}}[1, N_1, k_{max}], ..., \mathbf{S}_n^{\mathbf{D}}[1, N_n, k_{max}]\}$
10:     if Taskset $\Gamma$ is feasible under $\mathbf{S}^{\text{sol}}$ then
11:         for all $d \in \mathbf{D_s}$ in descending order do
12:             for all $\tau_i \in \Gamma$ do
13:                 Replace the $i$-th term in $\mathbf{S}^{\text{sol}}$ with $\mathbf{S}_i^d[1, N_i, k_{max}]$
14:             if Taskset $\Gamma$ is feasible under $\mathbf{S}^{\text{sol}}$ then
15:                 break /* The best solution is found for $\tau_i$*/
16:             else
17:                 for all $\tau_i \in \Gamma$ do
18:                     Restore the old $i$-th config in $\mathbf{S}^{\text{sol}}$
19:     else
20:         return $\mathbf{S}^{\text{sol}} = \emptyset$ /* no solution*/
21: end function
```

empty solution (line 20) as no feasible configuration is found. Otherwise, it tries to maximize dependability for all tasks. It iterates over $d \in \mathbf{D_s}$ in descending order (line 11), and replaces the protection configuration for each task τ_i in the solution configuration set $\mathbf{S}^{\text{sol}}$ with $\mathbf{S}_i^d[1, N_i, k_{max}]$. Then the algorithm checks the feasibility again (line 14). If the taskset is feasible, the algorithm returns the found solution $\mathbf{S}^{\text{sol}}$. Otherwise, it restores $\mathbf{S}^{\text{sol}}$ and then tries the next highest dependability value until the taskset is feasible.

The time complexity of our DP-based algorithm excluding schedulability test and SDC probability lookup is $O(\sum_{\tau_i \in \Gamma} \omega \cdot N_i^2)$, where N_i is the number of layers of τ_i and ω is determined by the size of user input, i.e., $\omega = |\mathbf{K_s}| \cdot |\mathbf{D_s}|$. Since our algorithm has a polynomial complexity w.r.t. N_i, it can handle complex models with a large number of layers.

Schedulability conditions. The schedulability test depends on the degree of real-time guarantees sought and the scheduling algorithm implemented. For soft real-time systems, AegisDNN implements the classical Least Slack Time (LST) scheduler to assign task priorities dynamically based on available slacks at runtime. LST is known to be optimal if tasks are preemptive with no shared resource and the system is not overloaded. Although GPU and SGX segments in our task model make it different from the optimal condition, we focus on whether the system will be overloaded when the found configuration is applied. Hence, the schedulability test for soft real-time systems checks if the cumulative utilization demand of all tasks does not exceed 100%, i.e., $\sum_{\tau_i \in \Gamma} \mathbf{U}_i^{\mathbf{D}}[1, N_i, k_{max}] \leq 1$. We found this simple admission control sufficient to make a good trade-off between dependability and soft real-time performance.

Although we primarily consider soft real-time systems in this work, the design of the AegisDNN framework and the way it controls access to SGX and GPU allow us to find a theoretical upper bound on task response time, which can be used by our algorithm to check hard real-time schedulability. The schedulability analysis developed for tasks with mutual-exclusive shared resources [11, 31, 36] can be used in our algorithm. Here, we briefly introduce an analysis tailored to our framework under partitioned fixed-priority scheduling. We define the priority of task τ_i as π_i, the total number of disjoint subsequences of protected layers of τ_i as K_i, the w-th execution segment of τ_i as $\tau_{i,w}$, the CPU core that τ_i is allocated to as $\mathbb{P}_i$, and the execution time of $\tau_{i,w}$ as $C^*_{i,w}(d)$, where $d \in \{g, e\}$ is the device that $\tau_{i,w}$ executes on. As discussed in the system, each of GPU and SGX segments is modeled as a critical section protected by a mutex lock. We consider the priority boosting mechanism of the well-known multiprocessor priority ceiling protocol (MPCP) [33] for bounded remote blocking. In addition, we assume that tasks are busy-waiting while waiting for a shared resource for simplicity. The worst-case response time (WCRT) R_i of τ_i is given by the following recurrence:

$$R_i = C_i + B_i + \sum_{\substack{\pi_h > \pi_i \\ \mathbb{P}_h = \mathbb{P}_i}} \left\lceil \frac{R_i}{T_h} \right\rceil (C_h + B_h) + \sum_{d \in \{g,e\}} \max_{\substack{\pi_l < \pi_i \\ \mathbb{P}_l = \mathbb{P}_i \\ 1 \leq j \leq K_l}} C^*_{l,j}(d)$$

where B_i is the remote blocking time. The third term captures preemption from higher-priority tasks. The last term represents interference from lower-priority tasks executing SGX and GPU segments with boosted priority, which happens only once per resource if tasks do not self-suspend. B_i is given by:

$$B_i = \sum_{1 \leq j \leq K_i} B_{i,j}(type(\tau_{i,j}))$$

where $B_{i,j}$ is the remote blocking time of $\tau_{i,j}$, $type(\tau_{i,j})$ is the device that $\tau_{i,j}$ executes on, i.e., g or e. The remote blocking time $B_{i,j}(d)$ of $\tau_{i,j}$ accessing the device d is given by [22]:

$$B_{i,j}(d) = \max_{\substack{1 \leq w \leq K_l \\ \pi_l < \pi_i}} C^*_{l,w}(d) + \sum_{\substack{d = type(\tau_{h,x}) \\ 1 \leq x \leq K_h \\ \pi_h > \pi_i}} \left(\left\lceil \frac{B_{i,j}(d)}{T_h} \right\rceil + 1 \right) C^*_{h,x}(d)$$

VI. Evaluation

This section presents experimental evaluation of AegisDNN. We first discuss the WCET and SDC profile results, and evaluate the accuracy and speed improvement of the proposed ML-based SDC prediction method. We then conduct integrated system experiments of AegisDNN with soft real-time tasksets. Lastly, we present the application of AegisDNN to hard real-time tasksets and discuss our findings.

A. Implementation

We have implemented AegisDNN in Caffe [1], PyTorch [30], and TensorFlow [4] for an Intel platform with SGX enabled running Ubuntu 18.04, in order to evaluate our work against various state-of-the-art fault-injection techniques. We used Eigen [2] as a basis for a light-weight linear-algebra library within enclaves since it does not have any dependency other than the C++ standard library. We have used C++11 `mutex` locks for both SGX and GPU resource access control, and `conditional_variable` for event synchronization.

We have enabled paging on SGX to support the execution of large DNN models. Upon initialization, AegisDNN pre-loads all the protected weights into the SGX enclave. This design is aligned with our system model in Sec. IV, allowing protection for weight parameters at the cost of page swapping. However, due to the current SGX physical memory size limitation, we argue that we cannot avoid such performance overhead in order to provide dependability to large DNNs.

B. Experiment Setup

We consider three popular DNN models in the evaluation: Lenet [23], Alexnet [21], ResNet-18 [14] for image classification, and Pilotnet [6] for self-driving. Three state-of-the-art fault-injection attacks are used: (i) random fault injection (RANFI) from TensorFI [25] and Ares [35], (ii) targeted fault injection (TFI) from BinFI [9], and (iii) bit-flip attack (BFA) with progressive bit search [34].[7] We consider both floating-point and INT8-quantized models. The hardware platform used features an Intel 7700K CPU with 16GB RAM, and an NVIDIA RTX 2080 Super graphics card with 8GB VRAM. We configured SGX with the maximum 128MB of encrypted memory in BIOS. Both CPU and GPU are configured to run at their maximum frequency. We disabled unrelated system services, e.g., WiFi and lightDM, to avoid potential interference.

C. DNN Execution Time and SDC Probability Profile

For all the DNN models considered in the evaluation, we first profile the worst-case execution time (WCET) of each layer when executed either on the GPU g or on the enclave e. We also profile the SDC probability of each layer $L_{i,j}$, $sdc_{\text{in}}(i,j,b)$ and $sdc_{\text{weight}}(i,j,b)$, for all DNN models considered. Figures 6, 7, 8 and 9 depict the WCET and SDC probability of DNN layers under the BER of 10^{-6} for AlexNet, LeNet, PilotNet, and ResNet, respectively. RANFI was used for AlexNet, LeNet, and PilotNet, and BFA for the quantized ResNet-18. Since BFA only attacks weight parameters, only sdc_{weight} values are displayed in Figure 9.

Observation 1. We generally observe an increasing trend in WCET when layers are executed on the enclave compared to on the GPU. However, the slowdown varies significantly layer by layer. For example, the layer 17 of Alexnet experiences a slowdown of over 50x when executed on the enclave, whereas surprisingly on the layers 11-13 of Pilotnet, the WCET on the enclave is shorter than that on the GPU. Such execution variation is mainly due to various memory copy size. The layer 17 of Alexnet is a fully-connected (FC) layer which requires over 200 MB of memory to run. When executing it on the enclave, page swapping happens frequently as a result of the limited physical memory of SGX, which significantly slows down the execution. Moreover, FC layers usually execute significantly faster on GPUs, which also contributes to the 50x slowdown of the enclave. The layers 11-13 of Pilotnet, however, are very small layers, which may not cause page swapping when executed on the enclave. The CPU-side overhead for executing on the GPU, such as memory copy between CPU-GPU memory and other miscellaneous operations, nullifies the GPU performance advantage.

[7]BFA and BinFI experiments were conducted on PyTorch and TensorFlow, respectively, by using the original authors' code.

TABLE I: Evaluation of ML prediction accuracy

DNN model	Cross-validation MAE%	Ground-truth MAE%
Pilotnet	2.14	1.03
Lenet	4.55	4.32
Alexnet	1.21	-
Resnet-18	4.80	-

Observation 2. We also observe significant variation for SDC probabilities. For Alexnet, some layers (e.g., layers 1,3-5,7-9,15-16,18,20-22) have close to zero SDC probability for both input (sdc_{in}) and weight data (sdc_{weight}). Thus protecting those layers does not contribute to lowering the task-level SDC probability. Layer 19 has the highest SDC probability for weight data (0.39). It also experiences very high slowdown (262 ms) when executed on the enclave. In other words, although protecting layer 19 can significantly improve dependability, the system might experience high performance penalty. Layer 17, however, has even higher slowdown (1184ms) compared to layer 19, but it features a much lower SDC probability for weight data (0.09). If layer 17 is protected instead of 19, the dependability gain is lower but the slowdown is larger. These results well justify AegisDNN's approach that utilizes a DP-based approach to choose only critical layers by balancing their SDC probabilities and performance cost based on the DNN profiles generated.

D. ML-based SDC Prediction

We evaluate the ML-based SDC prediction of AegisDNN in this section. We first randomly generate 512 uniformly distributed configurations for Pilotnet, 128 for Lenet, 1024 for Alexnet, and 512 for Resnet-18, which are decided based on their number of layers. Then we run fault injection for each generated configuration to obtain SDC probability samples. To evaluate the model, We use 80% of the obtained samples for the training data and 20% for cross-validation. The final model is trained by using all the samples. We also manage to run exhaustive fault injection for Pilotnet and Lenet to get all the ground truth data to better evaluate the accuracy of our ML prediction. Table I depicts the accuracy of ML prediction by cross-validation and ground-truth comparison. Our ML approach manages to achieve very low mean-absolute-error (MAE) in both cross-validation and ground-truth comparisons. We also measure the total time needed for AgeisDNN's ML approach and the exhaustive fault injection method to generate the SDC profile for all configurations, as shown in Table II. The "Training" column depicts the fault injection time needed to collect training data and to train the ML model. "Pred. All Config." denotes the time needed to predict the SDC probabilities under all configurations. "Est. FI All Config" shows the time needed to run exhaustive fault injection for all configurations. AegisDNN's ML approach significantly reduces (up to 99.9% for Alexnet) the time needed to generate

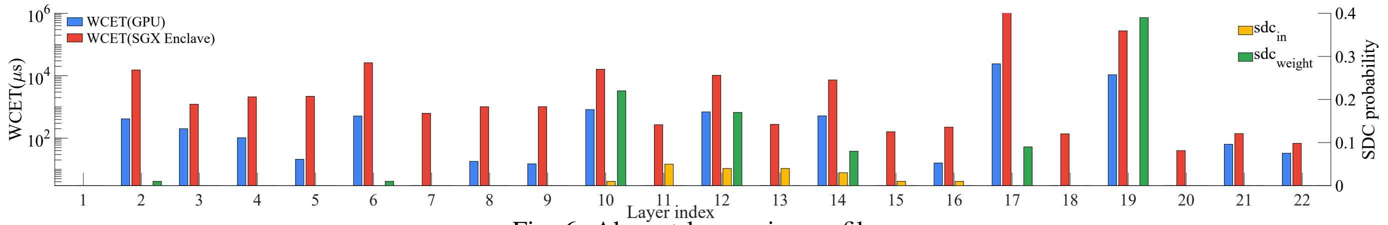


Fig. 6: Alexnet layer-wise profile

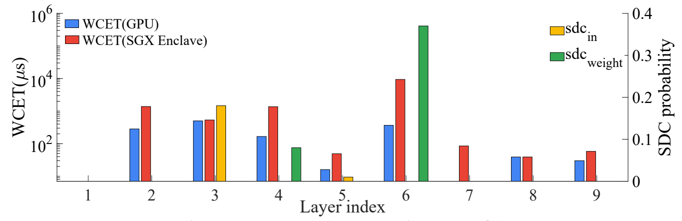


Fig. 7: Lenet layer-wise profile

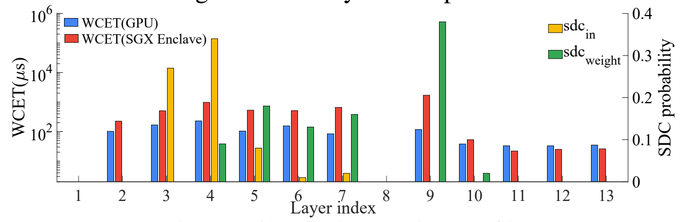


Fig. 8: Pilotnet layer-wise profile

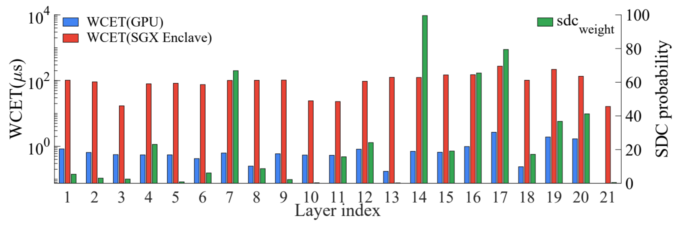


Fig. 9: INT8-quantized ResNet-18 layer-wise profile

TABLE II: Execution time comparison between AegisDNN ML approach and exhaustive fault injection appraoch

DNN model	Training	Pred. All Config.	Est. FI All Config.
Pilotnet	3.75h	1.27s	59.84h
Lenet	0.56	0.2s	2.25h
Alexnet	72hr	0.5h	33yr[8]
Resnet-18	28hr	0.4h	17yr[8]

the complete SDC profile needed for our algorithm. The experiment results indicate that AegisDNN's ML approach is able to generate acceptably accurate SDC profiles in significantly shorter time.

E. Integrated System Evaluation

In this section, we evaluate AegisDNN against AegisDNN-Simple and two baseline approaches. AegisDNN-Simple is a simplified version based on the layer-wise SDC profile of AegisDNN. It looks through $sdc_{\text{weight}}(i, j, b)$ and $sdc_{\text{in}}(i, j, b)$ (if exists) of each layer $L_{i,j}$, and picks the layers with the top-n SDC probabilities to protect. The two baselines we consider are SGX-only and GPU-only approaches, where the entire DNN model inference is executed only on the SGX enclave and the GPU, respectively. The GPU-only approach represents the conventional DNN inference frameworks such as Caffe, Tensorflow, and Torch as they do not provide any protection against fault attacks. The SGX-only approach represents the state-of-the-art such as Occlumency [24], where all DNN models are executed in the enclave and thus all protected. Two BERs are considered in the evaluation: 10^{-6} and 10^{-7}. All experiments are conducted on the real platform. We target *soft* real-time systems and use LST scheduling for this case study. In order to evaluate the soft real-time performance of our work from various aspects, we introduce the following metrics:

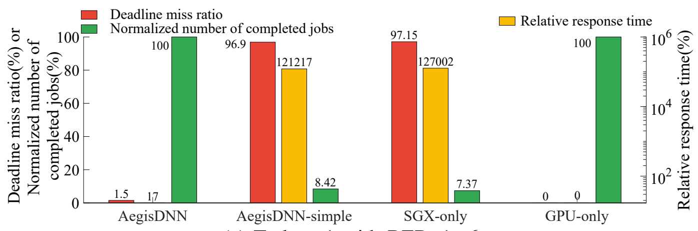
- *Deadline miss ratio*: the ratio of jobs that missed their deadlines during 10 minutes of test period. This metric can show the practical impact of the difference between our task model and LST's optimal condition.
- *Average relative response time*: the average of the observed response time normalized to the deadline for all jobs executed during our test period.
- *Normalized number of completed jobs*: the number of jobs completed by their deadlines, normalized to the number of jobs executed under the GPU-only approach.
- *Dependability*: the percentage of the jobs that successfully return the same output as in the fault-free condition.
- *Quality of Service (QoS)*: the percentage of the jobs that successfully return the same output as in the fault-free condition within their deadlines.

We consider two tasksets shown in Table III. Taskset 1 represents DNN execution scenarios in a self-driving car with multiple cameras: LeNet for traffic sign text, AlexNet for road object classification, and PilotNet for steering wheel actuation. Taskset 2 represents a DNN image classification server using heavy models: AlexNet and ResNet-18. We use floating-point models for taskset 1, and INT8-quantized models for taskset 2. Tasks are assigned different periods as the required update rate may vary depending on camera locations or service types.

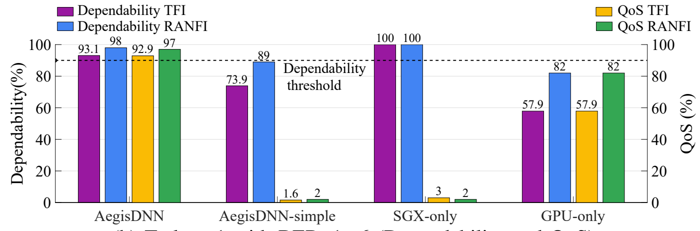
We require the system to achieve a dependability threshold of 90% when the BER is 10^{-6} as the system is running under a high degree of attacks, and 98% when the BER is 10^{-7} due to a relatively lower error rate. For RANFI, the attacker randomly injects faults across all the unprotected layers. Whereas for TFI and BFA, the attacker targets on only attacking the critical bits found by the corresponding fault injection tool. It is worth noting that finding the critical bits of DNNs not only requires a significant amount of time, e.g., 150 hours for one input image on AlexNet, but also needs visibility to the entire network parameters. Since AegisDNN prevents such visibility to the protected layers, such TFI and BFA attacks cannot be launched in real-life scenarios. We apply both RANFI and TFI to taskset 1. We deploy the most powerful BFA attack[9] to taskset 2 since it contains quantized DNN models that are more robust to

[8]This is an estimate based on the speed of progress on our tested platform.

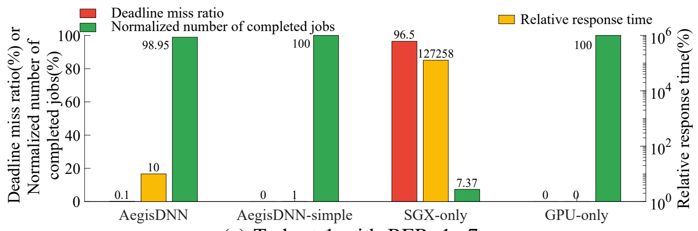
[9]BFA can successfully attack a ResNet-18 to fully malfunction (i.e., top-1 accuracy degrade from 69.8% to 0.1% on Imagenet 2012 dataset) with only 13 bit-flips out of 93 million bits.



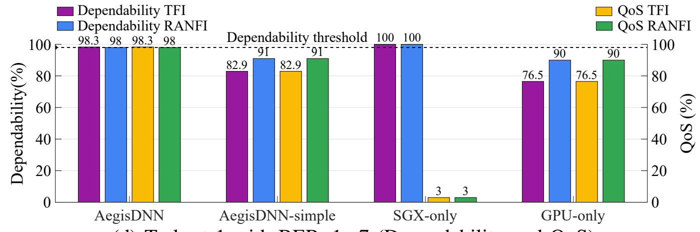
(a) Taskset 1 with BER=1e-6



(b) Taskset 1 with BER=1e-6 (Dependability and QoS)

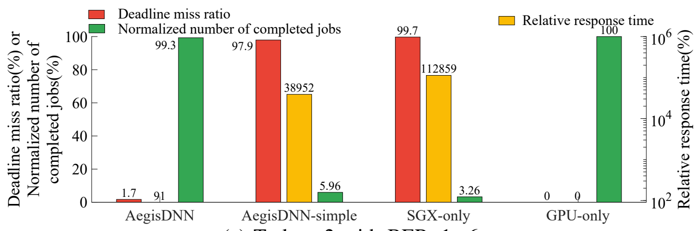


(c) Taskset 1 with BER=1e-7

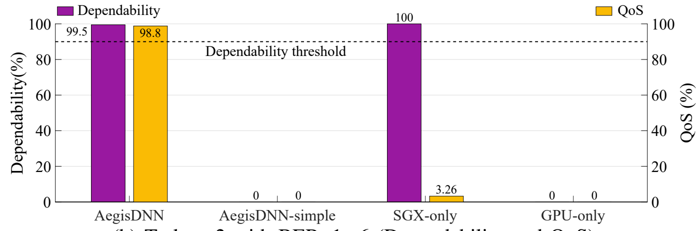


(d) Taskset 1 with BER=1e-7 (Dependability and QoS)

Fig. 10: Integrated system evaluation for taskset 1 under RANFI and TFI



(a) Taskset 2 with BER=1e-6



(b) Taskset 2 with BER=1e-6 (Dependability and QoS)

Fig. 11: Integrated system evaluation for taskset 2 under BFA

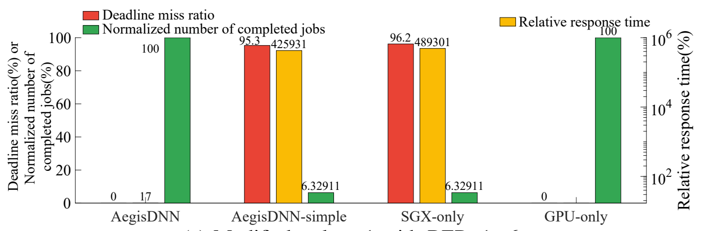
TABLE III: Taskset information

Taskset 1			Taskset 2 (INT8-Quantized)		
Task	**DNN model**	**Deadline**	**Task**	**DNN model**	**Deadline**
1	LeNet	30 ms	1	ResNet-18	100 ms
2	LeNet	50 ms	2	ResNet-18	200 ms
3	PilotNet	50 ms	3	ResNet-18	200 ms
4	PilotNet	80 ms	4	ResNet-18	400 ms
5	AlexNet	200 ms	5	AlexNet	500 ms
6	AlexNet	250 ms	6	AlexNet	500 ms
7	AlexNet	300 ms			

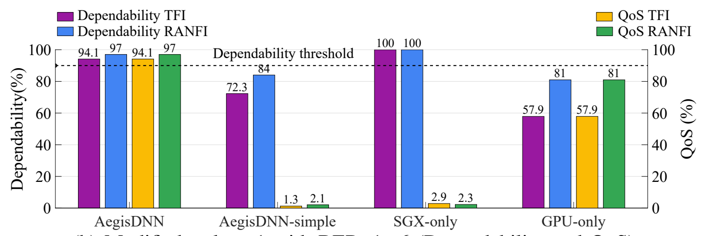
attacks [34]. For the AegisDNN algorithm, we choose $\mathbf{K_s}$ from $\{1, 2, 3, 4, 5\}$, allowing a maximum of 5 partitions between SGX and GPU. We choose $\mathbf{D_s}$ from $\{85, 86, 87..., 90\}$ when the dependability threshold is 90%, and $\{90, 91, ..., 100\}$ when it is 98%.

Figure 10 depicts the deadline miss ratio, relative response time, normalized number of completed jobs (normalized to the GPU-only approach), dependability, and QoS metrics for the taskset 1 under the two BERs and under RANFI and TFI attacks. Figure 11 shows the results of taskset 2 under BFA attack. For both tasksets under all three different attack methods, AegisDNN successfully finds the layer protection configurations under both BERs and meets the dependability threshold requirement. For taskset 1, the dependability of all approaches are lower under TFI compared to RANFI, indicating that TFI is a more powerful attack. AegisDNN has the highest QoS among all the approaches evaluated, and close to 0% of deadline miss ratio even under the high BER of 10^{-6}. It manages to complete the same number of jobs as the GPU-only approach during our test period. Such results demonstrate AegisDNN's ability to find the highest possible protection without sacrificing schedulability and number of completed jobs. It achieves up to 88.8% less failure (misclassification) rates than the unprotected GPU-only approach, and 99.9% shorter relative response time compared to the SGX-only approach. AegisDNN-Simple achieves the second best QoS under the BER of 10^{-7}, though it does not meet the dependability threshold requirement, as it only protects a certain number of layers without having the knowledge of the system overall SDC. It also suffers from drastic QoS loss when the BER changes to 10^{-6}, because it simply picks the layer with the highest SDC probability without considering execution time penalty or schedulability, which leads to 97% deadline miss ratio. For taskset 2, AegisDNN manages to achieve 99.5% dependability with a QoS of 98.3%, significantly better than any other approaches. AegisDNN-simple fails to protect the layers and the BFA attack makes the entire system a random generator. It also fails to find a reasonable layer protection configuration which considers the schedulability of the system, thereby scoring 0% QoS.

Overall, AegisDNN-Simple sometimes yields good results in some test cases despite a very simple algorithm but it lacks guarantees and stability due to its unawareness to schedulability and overall system SDC. SGX-only achieves the highest dependability but the lowest QoS (close to 0%) in all the test results because almost all the tasks have missed their deadlines. It also has the highest average relative response time. GPU-only achieves the highest number of completed jobs as well as the lowest deadline miss ratio and relative response time. However, due to its ignorance to attacks,



(a) Modified taskset 1 with BER=1e-6



(b) Modified taskset 1 with BER=1e-6 (Dependability and QoS)

Fig. 12: Modified taskset 1 with hard real-time constraints under RANFI and TFI

even under a low BER, it fails to maintain the dependability threshold requirement. In particular, it suffers significantly from TFI attack for taskset 1, e.g., almost half of the input is misclassified under BER=10^{-6}, and completely fails from BFA attack for taskset 2 (0% dependability and QoS).

Comparison with exhaustive search. We conduct another experiment to evaluate AegisDNN against the exhaustive search approach. We first compare the execution time of both algorithms. AegisDNN is able to find the configuration for taskset 1 in 243 ms. However, the exhaustive search could not find the configuration within an acceptable time[10] We thus create a small taskset for comparison: one task running LeNet with 30ms period, and another task running AlexNet with 100ms period. We consider the BER of 10^{-6} and the dependability requirement of 90%. The exhaustive search takes 508 seconds, whereas AegisDNN finishes its DP-based algorithm in only 80 ms. Both AegisDNN and exhaustive search manage to find the configuration meeting the dependability requirement for this taskset. They both achieve the same QoS of 97%. Exhaustive search has a slightly lower deadline miss ratio by 0.01%, but other performance metrics are almost identical and better than the other three approaches.

F. AegisDNN with Hard Real-Time Constraints

Lastly, we have applied AegisDNN to a hard real-time taskset to check if our analysis given in Sec. V-D has a meaningful effect in practice. There are several additional implementation efforts to consider. First, each DNN task is assigned a static real-time priority with `SCHED_FIFO`, and we determine task priorities based on Rate Monotonic (RM). Second, given that our hard real-time analysis assumes MPCP [33] for GPU and SGX, priority boosting is implemented on top of the standard mutex locks. Third, all tasks are allocated to the same CPU core in order to exclude the effect of task partitioning methods. In fact, we found that task allocation does not give discernible performance changes in our setup because (i) the system does not have any non-DNN task with real-time priority, the presence of which may cause preemption delay to DNN tasks, and (ii) all DNN tasks have an alternating sequence of SGX and GPU segments, both of which are critical sections. Thus, scheduling performance is dominated by the contention on GPU and SGX, but not on CPU cores.

We found that the taskset 1 in Table III cannot be used as-is under hard real-time requirements and any reasonably high dependability thresholds (≥80%), presumably due to the pessimism of the analysis. Hence, we increased the deadline of task 1 to 40 ms, task 3 to 100 ms, and task 5 to 250 ms, respectively, and then AegisDNN successfully found a configuration that is expected to meet the given dependability and hard real-time requirements. We used this modified version of the taskset 1 for hard real-time experiments.

Figure 12 shows the runtime results of the modified taskset 1 with hard real-time requirements and the BER of 10^{-6}. As expected from the offline analysis, AegisDNN satisfied the dependability requirement, with the highest QoS among all the approaches and no deadline miss during 10 minutes of experiments. It is worth noting that the modified taskset is still difficult to schedule on this hardware platform. If we revert the deadline relaxation of this taskset, it can no longer pass the schedulability test and we start to observe deadline misses at runtime although the number of misses is small. Therefore, these results indicate that our hard real-time schedulability analysis can reject unsafe tasksets and AegisDNN adheres to the scheduling behavior assumed by the analysis.

VII. Conclusions

In this paper, we presented AegisDNN, a DNN inference framework for timely and dependable execution with SGX. It is motivated by the limitations of recent work on SGX-only DNN protection that executes entire DNN model inside the enclave and does not consider real-time schedulability. We address such limitations by proposing layer-wise WCET and SDC profiling mechanisms, ML-based SDC prediction method, and DP-based configuration-finding algorithm. We have implemented AegisDNN based on Caffe and Eigen on a real platform. We conducted a thorough evaluation by comparing AegisDNN with a total of four different approaches under three state-of-the-art fault-injection attacks. In our experiments, AegisDNN was able to find layer protection solutions close to the optimal ones of exhaustive search, but in a considerably shorter time. It also outperformed the other approaches in many aspects, including response time, throughput, dependability, and QoS.

AegisDNN offers several interesting directions for future work. First, while this paper focuses on one GPU and one TEE, AegisDNN can be extended to multiple TEEs and multiple unprotected computing resources (e.g., distributed TEEs and Tensor Cores) to further improve protection level and throughput. Second, other TEE technologies are worth considering so that AegisDNN can be supported in more hardware platforms. We plan to explore those topics in the future.

[10]The estimated search time for the taskset 1 is $2.3397697 * 10^{12}$ years.

Acknowledgment

We appreciate the anonymous reviewers for their valuable comments and suggestions. This work was sponsored by the U.S. National Science Foundation (NSF) grants 1943265 and 1955650, and by the U.S. Office of Naval Research (ONR) grant N00014-19-1-2496. This work was also supported by the Institute of Information and Communications Technology Planning and Evaluation (IITP) Grant (No.2017-0-00067, Development of ICT Core Technologies for Safe Unmanned Vehicles) funded by the Korea Government (MSIT).

References

[1] Caffe. http://caffe.berkeleyvision.org. Accessed: 2019-03-30.

[2] Eigen C++ library for linear algebra. http://eigen.tuxfamily.org/. Accessed: 2020-06-30.

[3] Intel Software Guard Extensions (Intel SGX) SDK for Linux OS. http://intel.com. Accessed: 2020-06-30.

[4] M. Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[5] J. Abella, M. Padilla, J. D. Castillo, and F. J. Cazorla. Measurement-based worst-case execution time estimation using the coefficient of variation. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 22(4):1–29, 2017.

[6] M. Bojarski et al. End to end learning for self-driving cars. *CoRR*, abs/1604.07316, 2016.

[7] J. Breier, X. Hou, D. Jap, L. Ma, S. Bhasin, and Y. Liu. Deeplaser: Practical fault attack on deep neural networks. *arXiv preprint arXiv:1806.05859*, 2018.

[8] G. Buttazzo, G. Lipari, L. Abeni, and M. Caccamo. *Soft Real-Time Systems*. Springer, 2005.

[9] Z. Chen, G. Li, K. Pattabiraman, and N. DeBardeleben. BinFI: an efficient fault injector for safety-critical machine learning systems. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2019.

[10] T. Elgamal and K. Nahrstedt. Serdab: An IoT framework for partitioning neural networks computation across multiple enclaves. *arXiv preprint arXiv:2005.06043*, 2020.

[11] G. Elliott et al. GPUSync: A framework for real-time GPU management. In *IEEE Real-Time Systems Symposium (RTSS)*, 2013.

[12] K. Grover, S. Tople, S. Shinde, R. Bhagwan, and R. Ramjee. Privado: Practical and secure DNN inference with enclaves. *arXiv preprint arXiv:1810.00602*, 2018.

[13] J. Hansen, S. Hissam, and G. A. Moreno. Statistical-based WCET estimation and validation. In *9th international workshop on worst-case execution time analysis (WCET'09)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

[14] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

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

[16] S. Jain, I. Baek, S. Wang, and R. Rajkumar. Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[17] H. Kim, P. Patel, S. Wang, and R. R. Rajkumar. A server-based approach for predictable GPU access with improved analysis. *Journal of Systems Architecture*, 88:97–109, 2018.

[18] H. Kim and R. Rajkumar. Shared-page management for improving the temporal isolation of memory reservations in resource kernels. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2012.

[19] J. Kim, H. Kim, K. Lakshmanan, and R. Rajkumar. Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In *IEEE/ACM International Conference on Cyber-Physical Systems (ICCPS)*, 2013.

[20] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[21] A. Krizhevsky et al. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 2012.

[22] K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *IEEE Real-Time Systems Symposium*, 2009.

[23] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[24] T. Lee, Z. Lin, S. Pushp, C. Li, Y. Liu, Y. Lee, F. Xu, C. Xu, L. Zhang, and J. Song. Occlumency: Privacy-preserving remote deep-learning inference using SGX. In *ACM Conference on Mobile Computing and Networking (MobiCom)*, 2019.

[25] G. Li, K. Pattabiraman, and N. DeBardeleben. Tensorfi: A configurable fault injector for tensorflow applications. In *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2018.

[26] Y. Liu, L. Wei, B. Luo, and Q. Xu. Fault injection attack on deep neural network. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017.

[27] A. Marchand, P. Balbastre, I. Ripoll, M. Masmano, and A. Crespo. Memory resource management for real-time systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2007.

[28] A. K. Mok. Fundamental design problems of distributed systems for the hard real-time environment. *PhD Thesis, Massachusetts Institute of Technology*, 1983.

[29] N. Otterness et al. An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.

[30] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*, 2019.

[31] P. Patel et al. Analytical enhancements and practical insights for MPCP with self-suspensions. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018.

[32] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[33] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *IEEE Real-TimeSystems Symposium (RTSS)*, 1988.

[34] A. S. Rakin, Z. He, and D. Fan. Bit-flip attack: Crushing neural network with progressive bit search. In *IEEE/CVF International Conference on Computer Vision*, 2019.

[35] B. Reagen, U. Gupta, L. Pentecost, P. Whatmough, S. K. Lee, N. Mulholland, D. Brooks, and G.-Y. Wei. Ares: A framework for quantifying the resilience of deep neural networks. In *ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018.

[36] S. K. Saha, Y. Xiang, and H. Kim. STGM : Spatio-Temporal GPU Management for Real-Time Tasks. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2019.

[37] S. Shinde, D. Le Tien, S. Tople, and P. Saxena. Panoply: Low-TCB linux applications with SGX enclaves. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[38] F. Tramer and D. Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. *arXiv preprint arXiv:1806.03287*, 2018.

[39] S. Volos, K. Vaswani, and R. Bruno. Graviton: Trusted execution environments on gpus. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[40] Y. Xiang and H. Kim. Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference. In *IEEE Real-Time Systems Symposium (RTSS)*, 2019.

[41] M. Yang et al. Re-thinking CNN frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[42] M. Yang, N. Otterness, T. Amert, J. Bakita, J. H. Anderson, and F. D. Smith. Avoiding pitfalls when using NVIDIA GPUs for real-time tasks in autonomous systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2018.

[43] P. Zhao, S. Wang, C. Gongye, Y. Wang, Y. Fei, and X. Lin. Fault sneaking attack: A stealthy framework for misleading deep neural networks. In *ACM/IEEE Design Automation Conference (DAC)*, 2019.

[44] H. Zhou, S. Bateni, and C. Liu. S3DNN: Supervised streaming and scheduling for GPU-accelerated real-time DNN workloads. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018.

[45] Q. Zhu, W. Li, H. Kim, Y. Xiang, K. Wardega, Z. Wang, Y. Wang, H. Liang, C. Huang, and J. Fan. Know the unknowns: Addressing disturbances and uncertainties in autonomous systems. In *International Conference on Computer-Aided Design (ICCAD)*, 2020.