# Latency Attack Resilience in Object Detectors: Insights from Computing Architecture

Erh-Chung Chen[1], Pin-Yu Chen[2], I-Hsin Chung[2], Che-Rung Lee[1], and

[1] National Tsing Hua University, Hsinchu 300044, Taiwan
s107062802@m107.nthu.edu.tw, crlee@cs.nthu.edu.tw
[2] IBM Research, 1101 Kitchawan Rd Yorktown Heights NY 10598, USA
pin-yu.chen@ibm.com, ihchung@us.ibm.com

**Abstract.** Image-based object detectors are increasingly being used in surveillance and autonomous driving systems in real-time. However, those systems are threatened by latency attacks which inflate the elapsed time of each query, such that the system cannot respond properly within a reasonable time interval. In this paper, we find the root cause of the vulnerability of latency attacks on object detectors is caused by the occurrences of minor page faults. We propose a decision algorithm to mitigate this problem. The decision algorithm can automatically decide the optimal implementation to be executed based on the current status of the target system. To the best of our knowledge, this is the first paper to solve the latency issue from the point of view of computing architecture. Our studies provide a useful guideline for designing real-time applications on edge devices with more efficient and resilient responses.

**Keywords:** latency attack · object detection · page fault

## 1 Introduction

Object detection has become an indispensable technology in our daily lives, finding applications in various domains like surveillance systems [5, 13] and autonomous driving systems [1, 24]. An eligible detector should not only exhibit high accuracy but also demonstrate efficiency in inference time. To reduce the inference time, several techniques have been proposed, including network pruning [11, 31], and model quantization [7].

Security and reliability are another critical concern for model design. Latency attack, a form of adversarial attack, involves injecting imperceptible perturbations into input images, leading to a noticeable increase in the execution time of victim models [25, 29]. These attacks potentially engender abnormal system behavior and impede service accessibility for users. Moreover, they may escalate costs for service providers, necessitating additional computing resources and longer inference time to process malicious examples.

Prior research has delineated that the vulnerability of object detection is caused by non-maximum-suppression (NMS), which is employed to eliminate duplicated objects predicted by the models [25, 29], because the execution time of

NMS is a quadratic function of the total number of objects fed into NMS. In this paper, our study analyzes the impact of hardware configurations and operating systems on the execution time of NMS, revealing that prolonged execution times are attributable to minor page faults.

We list our main contributions in this paper as follows:

- We identify the root cause of latency attacks for object detectors from the point of view of computing architecture.
- We propose two minor page fault-aware NMS implementations at the application level to mitigate this problem under modern CPU-GPU architecture. The potential issue of resource exhaustion can be resolved by directly applying our solutions without necessitating system privileges or intricate knowledge of the operating system.
- To further enhance overall performance, we introduce a decision algorithm capable of autonomously selecting the optimal implementation based on the current status of the target system.
- By avoiding the occurrences of page faults, the execution time can be shortened. We believe that our findings provide a universal guideline for designing applications, especially for edge devices.

The rest of this paper is organized as follows. We give an overview of object detection and latency attacks in Sec. 2. Sec. 3 presents the root cause analysis. The implementations of the proposed minor page fault-aware NMS and runtime decision algorithm are described in Sec. 4. Sec. 5 shows the experimental results and discussions of this work. The last section is our conclusion.

## 2 Background

### 2.1 Object Detection

Object detection is a classical problem in computer vision that identifies all objects and their locations in the target images. Over the years, various models have been proposed to tackle this task, including Mask SSD [6], YOLO series [15, 28], and transformer-based detectors [20, 35].

An eligible detector should not only keep the high model accuracy but also have a fast inference time. For example, YOLO series provide models of different sizes by scaling the width of specific layers [15,28]. The lightweight models satisfy the strict constraints of memory usage and limited computing power on edge devices but a little bit of accuracy will be lost.

However, while execution times for image pre-processing and post-processing are crucial components, they have not received the same level of attention as model design in object detection research. These components play a significant role in the overall execution time of an object detection system for resource-constrained devices and should be taken into serious consideration.

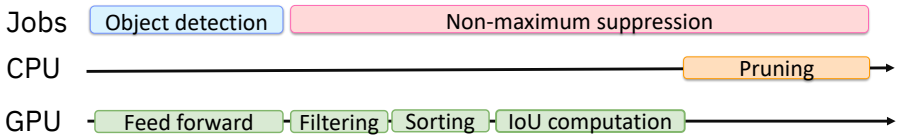



**Fig. 1:** Job scheduling for object detection on a heterogeneous architecture. The diagram is not drawn to scale.

### 2.2 Non-maximum Suppression

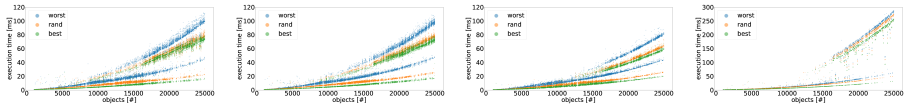
Non-maximum suppression (NMS) stands as a pivotal component in the majority of object detection models [8, 9, 28]. Its primary role lies in discerning whether multiple candidates pertain to the same objects. As shown in Fig. 1, NMS is followed by the object detection. On the modern CPU-GPU architecture, NMS consists of four distinct steps. In chronological order, the first step filters out the candidates whose confidence is lower than the threshold. The second step sorts all candidates according to their confidence in descending order. The third step, IoU computation, calculates the pair-wise scores based on the location and class information. A high score means a higher probability that the two candidates are considered to be the same object. The last step preserves the high-confidence objects and prunes redundant objects by the pair-wise scores accordingly.

As suggested in Overload [2], when the image resolution and the threshold are fixed, the execution time of NMS is dominated by the last two steps, which grows quadratically with the number of objects. This characteristic renders latency attacks practical by generating numerous ghost objects. It is crucial to note that the overhead is contingent on hardware despite their theoretical time complexity being the same. The study of latency attacks on object detectors from the perspective of computing architecture remains an underexplored area.

### 2.3 Adversarial Attack

Adversarial attacks can deceive models by intentionally adding an imperceptible perturbation to the original inputs [3, 22]. For image classification, adversarial attacks typically aim to decrease model accuracy. Conversely, for object detection, adversarial attacks can alter the location of bounding boxes [14], generate ghost objects [33] or make specific objects invisible [30].

The latency attack, a variant of adversarial attacks, aims to increase the execution time of deep learning-based applications [26]. Some dynamic vision models enable early prediction exit once certain conditions are met. These properties raise concerns about the possibility of latency attacks [12, 17]. Similarly, Overload attack [2] demonstrated that generating more objects results in increased execution time for object detection.




**(a)** Nvidia V100 **(b)** Nvidia 1080 Ti **(c)** Nvidia A10 **(d)** Nvidia Jetson NX

**Fig. 2:** Execution time of NMS on various GPUs

### 2.4 Mechanisms for Page Faults

To make the most efficient use of available memory, infrequently used data can be moved to the disk, allowing more programs to run simultaneously or to process larger data. This is accomplished through virtual memory. Physical memory, on the other hand, refers to the actual random access memory (RAM) installed on a computer. According to the definition of page fault accounting in Linux kernel, there are two types of page faults: major page faults and minor page faults. The major page fault moves the page from the disk space back to the physical memory. On the other hand, the minor page fault finds a suitable location in physical memory to map the virtual memory space without needing to retrieve the page from the disk.

The cost of page faults varies depending on the memory access pattern. For instance, two accesses on the same page that are not currently residing in physical memory trigger a page fault, whereas accessing two distinct pages leads to page faults. On the other hand, if variables are explicitly initialized in a proper time interval and all pages remain in physical memory without being swapped out, the corresponding overheads can be covered effectively.

## 3 Performance Analysis

This study delves into the impact of latency attacks on deep learning-based object detection systems, specifically focusing on task scheduling on CPU-GPU architectures. It particularly investigates edge devices, which are constrained by limited memory and lack the extensive page functionalities typical of operating systems. The research analyzes the underlying causes of these attacks and proposes preventive measures to mitigate their effects.

To gain insights into the vulnerability of NMS against latency attacks, we conducted experiments proposed in Overload [2] to assess the impact caused by the occurrences of page faults. The performance evaluation used synthetic data under three distinct cases. In the worst-case scenario, we simulated a situation in which all objects output by the object detection model survive. The best-case scenario returned only one object, while the random case randomly assigned properties to all objects. By implementing these three distinct cases, we are able to effectively evaluate the performance of the proposed methods under a range of circumstances, and make meaningful comparisons between them.

Theoretically, the execution time of NMS should scale quadratically with the total number of objects. Nevertheless, the experimental results shown in Fig. 2

**Table 1:** The occurrence of page faults.

| | page faults [#] | |
|---|---|---|
| | Baseline | Attack |
| Nvidia V100 | 2,491,556 | 13,285,029 |
| Nvidia 1080 Ti | 2,028,464 | 14,494,822 |
| Nvidia A10 | 2,497,918 | 13,456,800 |
| Nvidia Jetson NX | 1,644,311 | 10,845,541 |

demonstrate that divergent execution time curves form two distinct groups as the number of objects increases across various hardware configurations. This phenomenon indicates that there are some effects on latency that are not involved from the perspective of a high-level algorithm.

We conducted a performance analysis on CPU side by using a profiling tool called *perf*. Tab. 1 shows the numbers of page faults, which are accumulated by performing object detection on 1000 adversarial images generated by Overload attack in the column *Attack*, while *Baseline* represents the case where the images fed into NMS are the normal images. The statistical results clearly demonstrates a significant increase in the number of page faults in the presence of the latency attack. It is known that page faults create high overhead, and their occurrences depend on the memory access patterns on CPU side as well as the hardware specifications. This observation strongly suggests that the stochastic execution time is triggered by the occurrences of page faults.

The influences of occurrence of page faults can be illustrated in Fig. 3. While the diagram is not drawn to scale, it meticulously outlines the chronological order of execution. After the completion of IoU computation on GPU, the outputs are copied from GPU to CPU, which is called D2H (Device to Host), and fed into the Pruning function running on CPU. After the pruning, the data are moved from CPU to GPU, which is called H2D (Host to Device), and the used memories are freed. As depicted, page faults occur during D2H and Pruning function since whenever the CPU accesses a context that has not been loaded into physical memory. The occurrences of page faults bring unexpected overheads. Moreover, as the number of output objects increases, the demand for memory on the CPU side escalates, leading to a substantial rise in page fault occurrences and associated latency.

## 4 Proposed Methods

Theoretically, the overheads of minor page faults can be effectively hidden during the computation on GPU or the data transfer from GPU to CPU (D2H) if we explicitly access all allocated pages after the memory allocation immediately. However, several challenges behind the above solution may have resulted in GPU stalls. Specifically, the demanded pages depend on the number of active objects and their corresponding memory addresses, which remain unpredictable and unknown until IoU calculation completion. Consequently, this implementation

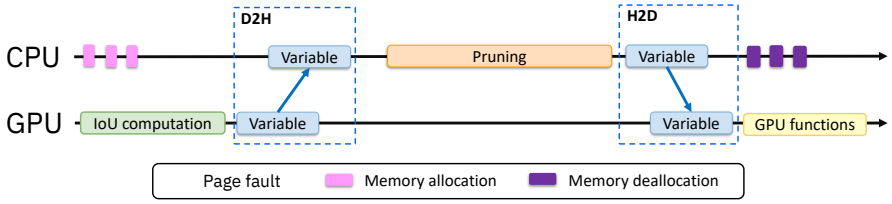



**Fig. 3:** Page faults occur during the data transformation from GPU to CPU and functions are executed on CPU. The diagram is not drawn to scale.

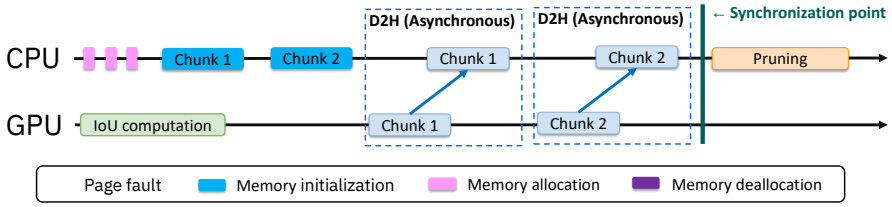



**Fig. 4:** Timeline for NMS with asynchronous data transfer. The diagram is not drawn to scale.

deepens minor page fault occurrences compared to the original approach, leading to increased execution times rather than improvements. To reduce the occurrence of minor page faults, we propose two minor page fault-aware implementations for NMS and an automatic decision algorithm that can decide the optimal NMS based on the current status of the target system.

### 4.1 Minor Page Fault-aware Implementations

The first solution is called *Async NMS*, which optimizes the performance by careful consideration of execution ordering on the CPU-GPU architecture and seeking a suitable interval to hide the overheads of minor page faults. By default, data transfer operates synchronously, which implies that both the CPU and GPU are engaged during this process. In order to offset these overheads, it becomes crucial to promptly schedule memory allocation and initialization after GPU functions are issued and to implement asynchronous data transfer. This ensures that the overheads for memory initialization can overlap with the data transfer process.

A write-after-write hazard may occur since the data from the GPU may be accidentally overwritten by the CPU initialization during asynchronous data transfer. To circumvent this, we adopt a strategy where the allocated memory is partitioned into multiple chunks. Each chunk is initialized by CPU and receives data from the GPU sequentially. This implementation is illustrated in Figure 4. Besides, a synchronization barrier is set before launching the CPU function. This step guarantees that all data have been transferred and completed and maximizes the time interval to hide the overheads of minor page faults.

On the other hand, CPU-free NMS initiates the NMS Pruning function directly on the GPU. This proactive stance completely eliminates the occurrence of minor page faults, as no functions are executed on the CPU during the object detection pipeline. For CPU-free NMS to outperform the original NMS, a prerequisite is that the execution time of the NMS Pruning function is notably shorter than the cumulative overhead induced by data transfer and page faults. An additional benefit of this approach is releasing CPU resources, which can be utilized by other processes such as image pre-processing or communication between file systems or networks. Nevertheless, the execution speed of the Pruning function on the GPU hinges on the power of the GPU being utilized.

### 4.2 Automatic Decision Algorithm

The aforementioned two implementations may inherently possess their own limitations. Additionally, the state of the operating system may also have an impact on the overall performance. These effects are not determined at the application level. To address this concern, we propose a run-time decision algorithm that can decide the optimal implementation automatically. This algorithm is executed at the pure application level without necessitating any system privilege or extensive knowledge about page faults. Furthermore, this approach is integrated into the NMS modules, and the assessment takes place while NMS is running, thereby minimizing the overheads required for performance tuning.

The decision algorithm requires a $m \times n$ table where $m$ represents the number of candidates of NMS implementations and $n$ stands for the number of objects fed into the NMS Pruning function. In this work, $m$ is set to 3, encompassing normal NMS, Async NMS, and CPU-free NMS, while maintaining a less granular approach to $n$ due to observed data fluctuations and the resulting increase in table size, which would require more effort for context population. For each NMS execution, the execution time of NMS Pruning is recorded and the corresponding table entry is updated. This table can be stored in the file system and re-used next time. The decision process is quite straightforward. The decision process is straightforward: select the NMS implementation with the shortest execution time for the given number of objects stored in the table, as depicted in Fig. 5, where *f3* is selected.

## 5 Experiments

This section presents three sets of experiments. The first experiment evaluates the time reductions for the proposed methods. The second experiment benchmarks the efficiency of the proposed solutions against latency attacks using real-world data sets. The last experiment assesses the performance of the decision algorithm.

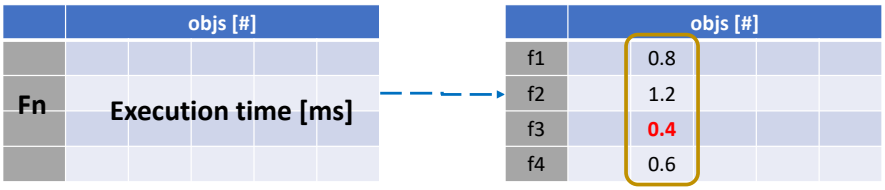



**Fig. 5:** The table records the execution times of different NMS with various numbers of input objects. *objs* stands for the number of objects fed into NMS Pruning function and *Fn* represents names of NMS implementations. In this case, *f3* is selected since it has the shortest execution time.

**Table 2:** Hardware configurations

| | CPU | GPU | OS | Architecture | CUDA | RAM |
|---|---|---|---|---|---|---|
| *Config 1* | Intel E5-2678 v3 | Nvidia V100 | Ubuntu 18.04 | amd64 | 11.6 | 128 GB |
| *Config 2* | Intel E5-2678 v3 | Nvidia 1080 Ti | Ubuntu 18.04 | amd64 | 11.6 | 128 GB |
| *Config 3* | Intel Gold 6258R | Nvidia A10 | Ubuntu 18.04 | amd64 | 11.6 | 128 GB |
| *Config 4* | NVIDIA Carmel ARMv8 | NVIDIA Volta GPU | Ubuntu 20.04 | aarch64 | 11.4 | 8 GB |

### 5.1 Setup

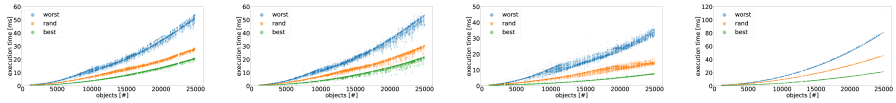
The primary goal of latency attacks is to maximize object generation by introducing imperceptible perturbations into target images and assessing the worst-case performance degradation. We operate under the assumption of unrestricted access to all information by attackers, with perturbations constrained within the $L_\infty$ norm and radii set to 16/255. The dimensions of input images are adjusted to (640, 640) and the used model is YOLOv5. Under this configuration, the maximum number of objects that can be output is 25,200.

To demonstrate the efficacy of the proposed methods, we conducted experiments using PyTorch 1.13.1 on four distinct devices, the specifications of which are detailed in Tab. 2. The first three configurations operate on *amd64* architecture. The only difference between the first two configurations is the used GPUs. The fourth configuration, on the other hand, involves Nvidia Jetson NX, a widely adopted edge device in various real-world applications. By testing on diverse hardware configurations, we can also assess the generalizability of the proposed methods to a wider range of settings.

### 5.2 Execution Time Evaluation

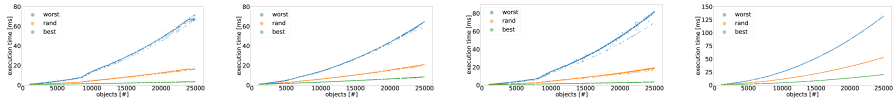
The main purpose of this experiment is to verify whether the two proposed NMS implementations can mitigate the latency issue. To make the experimental results comparable, we adopted the same experimental procedure for the root cause analysis in Sec. 3.

Fig. 6 shows the experimental results of Async NMS; Fig. 7 shows the experimental results of CPU-free NMS, where *objects* means the total number of synthetic objects. Tab. 3 summarizes the results of all implementations of NMS




**(a)** *config 1* **(b)** *config 2* **(c)** *config 3* **(d)** *config 4*

**Fig. 6:** Execution times of Async NMS




**(a)** *config 1* **(b)** *config 2* **(c)** *config 3* **(d)** *config 4*

**Fig. 7:** Execution times of CPU-free NMS

on various hardware configurations with the total object number in the interval $[23000, 24000]$, where *Attack* refers to the original NMS against latency attack, and the execution time is the average of the execution time for the total object number in the selected interval. The summarized results can fully demonstrate the potential risks of latency attacks. For *Attack*, both cases with and without minor page faults are included.

As shown in Fig. 6, it is noteworthy that the curves of execution times in all cases are observed to converge as the number of objects grows, indicating that this proposed method can defend against latency attacks from the aspect of computing architecture, particularly in scenarios where the number of objects is high. In comparison with the original NMS shown in the column *Attack* in Tab. 3, Async NMS demonstrates a remarkable ability to halve execution times across four hardware configurations, underscoring its effectiveness. These variances suggest that there may be some external factors beyond our control that can impact the performance of the NMS algorithm.

The CPU-free NMS implementation is also effective, as can be seen from the curves of execution times in Fig. 7, which converge as the number of objects increases. However, we observed that the execution times of CPU-free NMS are worse Async NMS for the worst cases. This phenomenon can be explained by the execution mode of the GPU, where multiple threads executed on GPU can cover the overheads of memory accesses and instructions loading. In contrast, CPU-free NMS executes the function Pruning with only one warp on the GPU, leading to a longer execution time.

In the best-case scenarios, CPU-free NMS achieves shorter execution times than that of Async NMS across all configurations. The reason is that Async NMS needs to communicate with CPUs but CPUs might not respond immediately because some background tasks are being processed. In a multi-user environment, there is still some interference since the resources are shared. On the other hand, Jetson NX is a fresh system, and the kernel can respond quickly. In the random cases, we find that in configurations 1 and 2, CPU-free NMS achieves shorter execution times than that of Async NMS but in configurations 3 and 4 the results

**Table 3:** The comparison of the execution time (ms) of NMS with different implementations and hardware configurations.

| | *config 1* | | | *config 2* | | | *config 3* | | | *config 4* | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | best | rand | worst | best | rand | worst | best | rand | worst | best | rand | worst |
| Attack | 62.32 | 66.96 | 82.93 | 58.05 | 64.33 | 78.25 | 46.75 | 52.34 | 63.97 | 215.30 | 225.06 | 232.93 |
| Async | 18.47 | 25.25 | **46.20** | 18.57 | 26.36 | **45.86** | 6.96 | **13.78** | **30.29** | 19.27 | **41.44** | **72.86** |
| CPU-free | **3.16** | **15.84** | 64.22 | **7.74** | **19.13** | 57.87 | **3.26** | 16.46 | 69.50 | **18.09** | 47.34 | 117.60 |

**Table 4:** The execution time of object detection with different implementations of NMS. We are highlighting the best performance among Async and CPU-free NMS against latency attacks.

| | Time [ms] | | | |
|---|---|---|---|---|
| | Baseline | Attack | Async | CPU-free |
| *Config 1* | 17.658 | 54.215 | 25.981 | **24.315** |
| *Config 2* | 8.475 | 62.559 | 20.362 | **19.464** |
| *Config 3* | 14.570 | 51.297 | 22.259 | **21.942** |
| *Config 4* | 36.558 | 263.84 | **58.893** | 60.118 |
| Overload | 23 | 230 | * | * |

**Table 5:** The execution time (ms) of the proposed NMS combined with the runtime decision algorithm.

| | *Config 1* | *Config 2* | *Config 3* | *Config 4* |
|---|---|---|---|---|
| Baseline | 17.658 | 8.475 | 14.570 | 36.558 |
| $\text{Baseline}_{\text{auto}}$ | 17.682 | 8.536 | 14.632 | 37.715 |
| Attack | 54.215 | 62.559 | 51.297 | 263.84 |
| $\text{Attack}_{\text{best}}$ | 24.315 | 19.464 | 21.942 | 58.893 |
| $\text{Attack}_{\text{auto}}$ | 24.477 | 19.475 | 21.897 | 59.125 |

are reversed. Furthermore, we notice that the variances in our experiments are relatively small for CPU-free NMS, indicating that the performance is influenced by the operating system or other programs running on the same machine.

We conclude the results shown in Tab. 3. In the best-case scenarios, CPU-free NMS exhibits superior performance. However, in the worst-case scenarios, the results are inverted. In random cases, both NMS implementations have similar execution times. Overall, the results demonstrate the two proposed methods can effectively mitigate latency attacks. We emphasize that almost all object detection models, including YOLOv7 [28], and YOVLv8 [16], can enjoy this benefit by replacing the NMS module.

### 5.3 Latency Evaluation on Real-world Data

To validate the efficacy of the proposed solutions in safeguarding against latency attacks on real-world data sets, we conducted experiments on a random subset of 1,000 images from the MS COCO data set, generating corresponding adversarial examples using the Overload attack [2]. To emulate the scenarios of surveillance and autonomous driving systems, the object detection model received an image per batch. For this experiment, we used YOLOv5 [15], the same model as used by Overload, to make direct comparisons between the results.

Tab. 4 shows the average time taken to process each image using a pipeline in milliseconds where *Baseline* means the objects fed into NMS are from normal images, and *Attack* refers to objects created by latency attacks. *Async* and *CPU-free* stand for Async NMS and CPU-free NMS respectively. To provide a basis

for comparison, we have also included the results reported by Overload at the bottom row of the table.

As can be seen, the proposed solutions maintain an execution time less than twice the execution time in *Baseline*, which is much less than the ones with latency attacks. These results suggest that the proposed solutions are robust and effective strategies for defending against latency attacks, particularly for Nvidia NX Jetson. When comparing the results of the experiments using synthetic data, as shown in Tab. 3, we notice that CPU-free NMS slightly outperforms Async NMS, except for Nvidia NX Jetson. This implies that the worst-case scenario is not practical for real-world data sets. Instead, some images in the selected set can naturally defend against latency attacks, meaning that their execution times are relatively faster and the average strength of Overload is weaker than that of the random cases using synthetic data.

On the other hand, *config 4* and Overload utilized the same hardware configuration and achieved comparable results, with latency attacks causing an approximately 7x-10x longer execution time. The performance gap may come from the differences between the selected images and the status of the operating system when running the experiments. Further analysis is required to determine the underlying causes of this discrepancy. Nevertheless, the proposed solutions have shown great promise in preventing latency attacks on object detection models.

### 5.4 Performance Analysis of Decision Algorithm

The choice between Async CPU-free NMS is contingent on specific hardware configurations and input data. Async NMS excels in worst-case scenarios with synthetic data, providing a robust defense against latency attacks. Meanwhile, CPU-free NMS establishes itself as the superior choice in best-case scenarios with synthetic data, where swift response times are paramount. In real-world scenarios, CPU-free NMS outperforms others across most configurations. The variations in performance underscore the intricate interplay of factors, necessitating a nuanced selection based on the prevailing conditions. Furthermore, CPU-free can alleviate CPU usage, potentially enhancing overall performance, especially when concurrently handling data processing or other tasks.

The primary objective of the decision algorithm is to streamline the process of manually selecting the optimal NMS implementation for specific scenarios. While our experiments have demonstrated the superiority of CPU-free NMS, it does not imply universal applicability. One could potentially devise a more potent attack than what we tested, and some commercial solutions might opt for more budget-friendly edge devices with lesser computing power than our tested models. Moreover, there may be scenarios we have yet to consider.

We conducted experiments to evaluate the overheads induced by the proposed decision algorithm and to analyze the execution time of the NMS chosen by the decision algorithm. All experimental configurations mirror those of the real-world data assessment, with the addition of the decision algorithm. Tab. 5 provides the execution time of the proposed NMS combined with the run-time decision algorithm where $\text{Baseline}_{\text{auto}}$ denotes the normal data in the pipeline

being processed with the decision algorithm; $\text{Attack}_{\text{best}}$ signifies that adversarial examples in the pipeline are processed using the best NMS results as presented in Tab. 4; and $\text{Attack}_{\text{auto}}$ designates that adversarial examples in the pipeline are processed by the decision algorithm.

The decision algorithm maintains an additional table to store information about each NMS implementation. However, it is evident that the overhead introduced is minimal when compared to standard processing in the baseline scenario. Similarly, for adversarial examples, only negligible overheads are observed. The only potential concern is that the introduction of the additional table by the decision algorithm may potentially lead to more significant page faults on some memory-constrained devices. However, it is worth noting that this phenomenon did not occur in our experiments. Therefore, this serves to underscore the efficiency and practicality of the decision algorithm.

### 5.5 Discussion

Nowadays, NMS is an essential component for most object detection models. Recent works [4,21] have surveyed commonly used models for underwater object detection and aerial images are YOLO [15, 18, 28], R-cnn [8, 10] series or other models that require the NMS algorithm. The proposed methods can effectively defend against latency attacks for object detection. In particular, NMS with asynchronous data transfer reduces the occurrences of page faults implicitly and halves the execution times on four configurations. We anticipate that our findings will significantly impact real-time object detection, particularly in applications like YOLO5Face [23] and YOLO-FaceV2 [34], on edge devices. However, there are some limitations or potential impacts on our work, as addressed below.

**Broader impacts for real-time applications** Although our focus is on object detection and latency attacks, the results of our experiments demonstrate that minor page faults can significantly impact the execution time. Minor page faults may occur in the time points that are irrelevant to the execution flow of deep learning models, but they can still slow down the overall performance. By avoiding the occurrences of page faults, the execution time can be shortened, leading to more efficient applications. Therefore, we believe that our findings provide a universal guideline for designing applications on edge devices. This is particularly important in applications that require high performance and low latency, such as real-time applications on edge devices.

Our proposed solutions offer the flexibility to distribute the computational load either to the CPU or GPU side, allowing for adaptable task scheduling based on current computing resource utilization. In scenarios where object detection applications are deployed on the cloud, receiving multiple requests from numerous users is common. Adopting a dynamic scheduling policy can help mitigate resource exhaustion and consequently reduce latency. We believe that our work highlights the importance of system-wide design considerations for deep learning applications to optimize overall performance.

**Defenses** We acknowledge that our proposed methods cannot directly enhance the robustness of object detectors, but they do effectively bind the overall

execution time of object detection tasks. This provides object detectors with the ability to add extra modules that can detect adversarial examples but maintain the overall execution time. For example, [19, 32] have identified that the occurrence of multiple unrelated objects within a single image is indicative of content inconsistency. This is a rare occurrence for typical examples, but it is likely to be an adversarial example. Holistically, our defense, when combined with an adversarial example detector, can improve both the security and reliability of object detection models. We believe that our contributions make significant progress in designing robust object detection tasks.

**Alternative solution** The usage of huge pages is one of the potential solutions. In theory, the occurrences of minor page faults can be reduced but it necessitates further in-depth investigations to assess its overall impact on performance. The numerous occurrences of major page faults on Jetson NX shown in our experiment indirectly indicate that the available physical memory is inadequate to accommodate intermediate results generated by the object detector. Furthermore, it is important to consider that certain edge devices may not support this feature, and even if they do, users may not have the necessary privileges to enable it. Therefore, the feasibility and effectiveness of utilizing huge pages as a solution should be carefully evaluated in the specific context of edge devices.

Pre-allocating a reusable buffer at the outset may seem theoretically sound, but the dynamic memory requirements for each image, which are only known until IoU computation, present a challenge. In memory-constrained environments, pre-allocating a larger buffer often adversely affects performance. Moreover, PyTorch lacks a global memory manager, meaning buffers are confined to the scope of the NMS function and necessitate repeated allocations. We argue that reimplementing NMS offers a more feasible and efficient solution compared to implementing a complex memory manager.

**Uncontrollable factors** The experiments shown in Fig. 6 suggest that external factors contribute about five percent fluctuation for *Async* NMS. If we consider the overall performance of object detection, it should involve image preprocessing, storing data in the file system, or reading images from the senses. These operations may interrupt the program and compete with the function Pruning for CPU resources. Moreover, CPU migrations and context switches can occur frequently on edge devices with limited resources, further affecting the optimal defense against latency attacks. Therefore, the optimal defense against latency attacks may need to be altered depending on the specific hardware and software configurations. It is crucial to consider not only the performance of NMS but also the overall performance of the object detection system when designing defenses against latency attacks.

**Overhead of minor page faults** We have addressed how the occurrences of minor page faults influence the performance of NMS in this paper, but the exact overheads of minor page faults have not been presented in this paper. The major reason is that the mechanism of page faults is more sophisticated. Moreover, modern CPUs have a hierarchy of multiple cache layers with different data allocation policies. The exact overheads of minor page faults may be influenced

**Table 6:** The occurrences of page faults on Nvidia Jetson NX.

| | major page faults [#] | minor page faults[#] |
|---|---|---|
| Baseline | 1,199,167 | 448,970 |
| Attack | 1,200,856 | 10,860,747 |
| Async | 1,199,251 | 501,256 |
| CPU-free | 1,199,000 | 466,891 |

by several factors. Recent work proposed a hardware implementation to reduce the cost of minor page faults [27].

Additionally, we found some unexpected behaviors. Firstly, although typical computers have around 8 GB of memory, the algorithm requires only about 100 MB. However, as shown in Tab. 6, we note major page faults are triggered so frequently on Jetson NX. Secondly, the occurrences of major page faults for Async NMS are reduced when data are transferred asynchronously. However, the total sizes of allocated memory remain constant. Those two contradictory observations implicitly suggests the interaction mechanisms between CPU and GPU beyond our explanation. Currently, we cannot give it a further analysis due to the opaque nature of the implementation regarding GPU firmware.

## 6 Conclusion

In this paper, we have analyzed the execution performance of NMS and found that the occurrence of minor page faults is the root cause of the vulnerability. Based on the analysis, we suggested a solution to schedule the data transfer asynchronously to cover the overheads of minor page faults. Another solution is to execute all functions on GPU such that the occurrences of minor page faults can be eliminated entirely. We further proposed a decision algorithm that can decide the optimization in the inference time. Besides, our methods do not require system privileges or intricate knowledge of the operating system. We conducted experiments across various hardware configurations to demonstrate the efficacy of our proposed methods. The results showed that our approaches can halve the execution time. We believe that our findings provide a universal guideline for designing real-time applications on edge devices and mitigate resource exhaustion for cloud applications.

## Acknowledgments

This material is based upon work supported by the Chief Digital and Artificial Intelligence Office under Contract No. W519TC-23-9-2037 for Pin-Yu Chen. Additionally, this material is partially supported by NTSC, Taiwan, R.O.C. under Grant No. 113-2221-E-007-140. The views and conclusions contained herein are those of the author(s) and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government.

## References


1. Arnold, E., Al-Jarrah, O.Y., Dianati, M., Fallah, S., Oxtoby, D., Mouzakitis, A.: A survey on 3d object detection methods for autonomous driving applications. IEEE Transactions on Intelligent Transportation Systems **20**(10), 3782–3795 (2019) 1
2. Chen, E.C., Chen, P.Y., Chung, I., Lee, C.R., et al.: Overload: Latency attacks on object detection for edge devices. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 24716–24725 (2024) 3, 4, 10
3. Chen, E.C., Lee, C.R.: Towards fast and robust adversarial training for image classification. In: Proceedings of the Asian Conference on Computer Vision (ACCV) (November 2020) 3
4. Ding, J., Xue, N., Xia, G.S., Bai, X., Yang, W., Yang, M.Y., Belongie, S., Luo, J., Datcu, M., Pelillo, M., et al.: Object detection in aerial images: A large-scale benchmark and challenges. IEEE transactions on pattern analysis and machine intelligence **44**(11), 7778–7796 (2021) 12
5. Elhoseny, M.: Multi-object detection and tracking (modt) machine learning model for real-time video surveillance systems. Circuits, Systems, and Signal Processing **39**, 611–630 (2020) 1
6. Fu, C.Y., Liu, W., Ranga, A., Tyagi, A., Berg, A.C.: Dssd: Deconvolutional single shot detector. arXiv preprint arXiv:1701.06659 (2017) 2
7. Gholami, A., Kim, S., Dong, Z., Yao, Z., Mahoney, M.W., Keutzer, K.: A survey of quantization methods for efficient neural network inference. In: Low-Power Computer Vision, pp. 291–326. Chapman and Hall/CRC (2022) 1
8. Girshick, R.: Fast r-cnn. In: International Conference on Computer Vision (ICCV) (2015) 3, 12
9. Girshick, R., Donahue, J., Darrell, T., Malik, J.: Rich feature hierarchies for accurate object detection and semantic segmentation. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 580–587 (2014) 3
10. He, K., Gkioxari, G., Dollár, P., Girshick, R.: Mask r-cnn. In: Proceedings of the IEEE international conference on computer vision. pp. 2961–2969 (2017) 12
11. Hoefler, T., Alistarh, D., Ben-Nun, T., Dryden, N., Peste, A.: Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. Journal of Machine Learning Research **22**(241), 1–124 (2021) 1
12. Hong, S., Kaya, Y., Modoranu, I.V., Dumitraş, T.: A panda? no, it's a sloth: Slowdown attacks on adaptive multi-exit neural network inference. arXiv preprint arXiv:2010.02432 (2020) 3
13. Jha, S., Seo, C., Yang, E., Joshi, G.P.: Real time object detection and trackingsystem for video surveillance system. Multimedia Tools and Applications **80**, 3981–3996 (2021) 1
14. Jia, Y.J., Lu, Y., Shen, J., Chen, Q.A., Chen, H., Zhong, Z., Wei, T.W.: Fooling detection alone is not enough: Adversarial attack against multiple object tracking. In: International Conference on Learning Representations (ICLR'20) (2020) 3
15. Jocher, G.: YOLOv5 by Ultralytics (May 2020). https://doi.org/10.5281/zenodo.3908559, https://github.com/ultralytics/yolov5 2, 10, 12
16. Jocher, G., Chaurasia, A., Qiu, J.: YOLO by Ultralytics (Jan 2023), https://github.com/ultralytics/ultralytics 10
17. Kung, C.H., Lee, C.R.: Add: A fine-grained dynamic inference architecture for semantic image segmentation. In: 2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 4792–4799. IEEE (2021) 3

18. Li, C., Li, L., Jiang, H., Weng, K., Geng, Y., Li, L., Ke, Z., Li, Q., Cheng, M., Nie, W., et al.: Yolov6: a single-stage object detection framework for industrial applications. arXiv preprint arXiv:2209.02976 (2022) 12
19. Li, S., Zhu, S., Paul, S., Roy-Chowdhury, A., Song, C., Krishnamurthy, S., Swami, A., Chan, K.S.: Connecting the dots: Detecting adversarial perturbations using context inconsistency. In: Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XXIII 16. pp. 396–413. Springer (2020) 13
20. Li, Y., Mao, H., Girshick, R., He, K.: Exploring plain vision transformer backbones for object detection. In: Computer Vision–ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part IX. pp. 280–296. Springer (2022) 2
21. Liu, C., Li, H., Wang, S., Zhu, M., Wang, D., Fan, X., Wang, Z.: A dataset and benchmark of underwater object detection for robot picking. In: 2021 IEEE International Conference on Multimedia & Expo Workshops (ICMEW). pp. 1–6. IEEE (2021) 12
22. Madry, A., Makelov, A., Schmidt, L., Tsipras, D., Vladu, A.: Towards deep learning models resistant to adversarial attacks (2019) 3
23. Qi, D., Tan, W., Yao, Q., Liu, J.: Yolo5face: Why reinventing a face detector. In: European Conference on Computer Vision. pp. 228–244. Springer (2022) 12
24. Qian, R., Lai, X., Li, X.: 3d object detection for autonomous driving: a survey. Pattern Recognition **130**, 108796 (2022) 1
25. Shapira, A., Zolfi, A., Demetrio, L., Biggio, B., Shabtai, A.: Phantom sponges: Exploiting non-maximum suppression to attack deep object detectors. In: Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision. pp. 4571–4580 (2023) 1
26. Shumailov, I., Zhao, Y., Bates, D., Papernot, N., Mullins, R., Anderson, R.: Sponge examples: Energy-latency attacks on neural networks. In: 2021 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 212–231. IEEE (2021) 3
27. Tirumalasetty, C., Chou, C.C., Reddy, N., Gratz, P., Abouelwafa, A.: Reducing minor page fault overheads through enhanced page walker. ACM Transactions on Architecture and Code Optimization (TACO) **19**(4), 1–26 (2022) 14
28. Wang, C.Y., Bochkovskiy, A., Liao, H.Y.M.: Yolov7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors. In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. pp. 7464–7475 (2023) 2, 3, 10, 12
29. Wang, D., Li, C., Wen, S., Han, Q.L., Nepal, S., Zhang, X., Xiang, Y.: Daedalus: Breaking nonmaximum suppression in object detection via adversarial examples. IEEE Transactions on Cybernetics (2021) 1
30. Wu, Z., Lim, S.N., Davis, L.S., Goldstein, T.: Making an invisibility cloak: Real world adversarial attacks on object detectors. In: European Conference on Computer Vision. pp. 1–17. Springer (2020) 3
31. Yeom, S.K., Seegerer, P., Lapuschkin, S., Binder, A., Wiedemann, S., Müller, K.R., Samek, W.: Pruning by explaining: A novel criterion for deep neural network pruning. Pattern Recognition **115**, 107899 (2021) 1
32. Yin, M., Li, S., Cai, Z., Song, C., Asif, M.S., Roy-Chowdhury, A.K., Krishnamurthy, S.V.: Exploiting multi-object relationships for detecting adversarial attacks in complex scenes. In: proceedings of the IEEE/CVF international conference on computer vision. pp. 7858–7867 (2021) 13

33. Yin, M., Li, S., Song, C., Asif, M.S., Roy-Chowdhury, A.K., Krishnamurthy, S.V.: Adc: Adversarial attacks against object detection that evade context consistency checks. In: Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision. pp. 3278–3287 (2022) 3
34. Yu, Z., Huang, H., Chen, W., Su, Y., Liu, Y., Wang, X.: Yolo-facev2: A scale and occlusion aware face detector. arXiv preprint arXiv:2208.02019 (2022) 12
35. Zhang, Z., Lu, X., Cao, G., Yang, Y., Jiao, L., Liu, F.: Vit-yolo: Transformer-based yolo for object detection. In: Proceedings of the IEEE/CVF international conference on computer vision. pp. 2799–2808 (2021) 2