Multi-version Machine Learning and Rejuvenation for Resilient Perception in Safety-critical Systems*

Qiang Wen*‡, Júlio Mendonça†‡, Fumio Machida*, Marcus Völp†
*Department of Computer Science, University of Tsukuba, Japan
†Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg
wen.qiang@sd.cs.tsukuba.ac.jp, julio.mendonca@uni.lu, machida@cs.tsukuba.ac.jp, marcus.voelp@uni.lu

***Abstract*—Machine learning (ML) has become a crucial component in safety-critical systems, such as those used in autonomous vehicle perception. However, the correctness and, therefore, the safety of these systems can be compromised by incorrect input data, accidental faults, and security breaches. This paper investigates using a replicated ML architecture to mitigate the risks associated with complex single-points-of-failure. Additionally, it explores the application of rejuvenation to sustain healthy majorities when facing persistent threats. We evaluate the output reliability of the proposed architecture in two case studies: traffic sign detection and perception for autonomous driving. We adopt models and reliability functions, validating our findings using realistic data sets and fault injection experiments. Using CARLA simulator, we also evaluate the driving safety when using the proposed architecture. Our results show our models can present a good generalization and multi-version ML with proactive rejuvenation can improve correctness and, thus, safety despite faults and cyberattacks.**

***Index Terms*—Multi-version system, Machine learning, Perception, Rejuvenation, Reliability, Safety**

I. Introduction

Recent advances in Machine Learning (ML) have revolutionized application domains, such as text, speech, image and video processing, control, and even arts. The success of ML has also led to widespread adoption in safety-critical application domains, such as autonomous driving or health, with in-car perception systems being among the most popular use cases. Perception identifies the road, traffic signs, other vehicles, and pedestrians to obtain an accurate map of the vehicle's surroundings, classifying potential mobile and stationary obstacles to predict possible future behaviors of the former [1]. The goal is to identify a safe route for the vehicle to approach its destination.

ML-based systems, including perception systems, cannot guarantee output correctness due to inherent uncertainty related to the probabilistic decisions of ML models [2], [3]. Furthermore, transient faults (e.g., bit flips in the weight matrix [4]) and malicious cyber-attacks (e.g., adversarial inputs [5]) can degrade the perception performance, potentially causing misclassification with possibly severe consequences, leading to accidents or even loss of human life. Therefore, reliability of perception is still a significant concern and challenge to be solved, in particular for safety-critical applications such as autonomous driving.

N-version programming (NVP) [6] is a promising approach to ensure diversity, leading to independence in the failure characteristics of versions performing a task. This diversity ultimately enhances safety by enabling the survival of a healthy majority, which can outvote arbitrarily faulty or compromised modules (i.e., Byzantine modules). Previous studies have analyzed the adoption of N-version ML systems and presented their benefits for improving output reliability [7], [8], [9], [10]. However, they only considered classification errors due to the uncertainty of ML models and not adversarial circumstances such as transient faults or cyber-attacks. Also, NVP only provides an initial diversity to secure a healthy majority, which may be lost as persistent adversaries continue to learn how initial versions may be compromised. Proactive rejuvenation and automatic diversification are needed [11], which have not yet been adequately investigated for NVP [12].

This paper proposes a novel architecture that integrates multi-version ML and reactive/proactive rejuvenation to enhance the relability of safety critical ML systems. To quantitatively evaluate the effectiveness of the proposed architecture and study its parameters, we develop Deterministic and Stochastic Petri Net (DSPN) models and reliability functions that assess how trustworthy the resulting system will be. The DSPN models capture accidental faults and intentionally malicious threats and their effect on vehicular perception systems. Since the architecture uses multiple-version ML, decision rules are defined to provide the final output of the system. These rules not only support full fault masking in situations where this is still possible but also degraded reliability modes of operation after an increasing number of ML modules cease to operate. We conduct experiments for two-version and three-version ML modules (e.g., dual or triple modular redundant perception systems) through a real-world traffic sign dataset and reliability models. Our results show that the reliability of the two-version system can be higher than that of the three-version system. The results also show scenarios where adopting a rejuvenation mechanism would be beneficial or not. We analyze the optimum values for the choice of the rejuvenation time interval and other relevant input parameters regarding system output reliability. Lastly, we implement a case study where we use a multi-version

‡Equal contribution
*This is a preprint version. The final version will appear in the DSN'25 proceedings.

perception system with time-triggered proactive rejuvenation in an AV simulator Carla to evaluate the proposed approach in a real-world scenario. The main contributions of this work are summarized as follows:

1) develop DSPN models and reliability functions to compute the output reliability of systems, taking into account accidental faults, malicious attacks, and proactive and reactive rejuvenation;
2) perform fault injection experiments to confirm the analytical findings and to evaluate multi-version ML models;
3) perform parameter studies, analyzing the interplay of various parameters to find scenarios where output reliability is maximized; and
4) demonstrate the evaluation of the proposed work by implementation of a multi-version perception system with time-triggered proactive rejuvenation in an AV simulator to enhance driving safety.

The remainder of the paper is organized as follows. Section II presents background and related work. Section III introduces the current vulnerabilities in ML systems. Section IV presents the proposed architecture for multi-version ML systems subjected to proactive rejuvenation. Section V presents the developed DSPN models to analyze rejuvenating multi-version ML systems. Section VI presents experiments using a real-world traffic sign dataset and numerical analysis of the proposed models. Section VII presents a case study where a multi-version perception system with time-based rejuvenation is implemented in an AV. Section VIII discusses some threats to the validity and limitations of our work. Finally, Section IX concludes this paper.

II. Background and Related Work

A. N-version machine learning

N-version ML system is the architectural approach to improve the reliability of ML system output by exploiting multiple versions of ML inferences [13]. Several architecture options are possible as different ML models and input data can diversify the inferences from independent ML modules. For example, Wen and Machida [13], [10] proposed reliability models to design two- and three-version ML systems. Mendonça et al. [14] proposed the use of N-version ML and rejuvenation to improve the reliability of perception systems output. However, they only evaluated the system through models and functions. Other works, such as Wu et al. [15], Xu et al. [16] and Latifi et al. [8], experimentally evaluated the adoption of N-version ML systems, showing reliability benefits.

Two-version systems run two ML modules in parallel and compare inference results to determine the system output. If one of the two inference results is incorrect, the decision is postponed to maintain safety, trusting in re-executing the inference correctly. Three-version systems include one additional ML module and a voter to select the majority result. The voter can adopt, for instance, a 2-out-of-3 scheme, where the system failure occurs when two or three ML modules output errors coincidentally. Lyons and Vanderkulk [17] defined the failure probability of a three-version system as $F = 3(1-p)p^2 + p^3$, where p is the output error probability of a system version, and $0 \leq p \leq 1$. In this equation, it is assumed that output errors of the system versions are independent. However, this assumption is unlikely to hold, as demonstrated by Littlewood et al. [18]. Ege et al. [19] further extended the model to assume that the error probability is dependent on a factor α, where $0 \leq \alpha \leq 1$. Thus, the failure probability of the three-version system amounts to

$$F = 3\alpha p(1-\alpha) + \alpha^2 p. \quad (1)$$

The reliability of the three-version system becomes $R = 1 - F$. However, in this case, they assume the same p and α for all three system versions, which is an assumption unlikely to hold for ML-based systems. Later, Wen and Machida [10] extended this model, by allowing different values for p and α. The failure probability of the three-version ML system is given by

$$F = \alpha_{1,2} \cdot p_1 + \alpha_{1,3} \cdot p_1 + \alpha_{2,3} \cdot p_2 - 2\alpha_{1,2} \cdot \alpha_{1,3} \cdot p_1, \quad (2)$$

where p_i is the probability that ML model m_i outputs errors and $\alpha_{i,j}$ represents the intersection of error sets E_i and E_j, which are the input sets that make ML models m_i and m_j output error, respectively.

B. Fault-injection and ML

Fault-injection experimentation comprises well-developed techniques to evaluate the dependability properties of systems [20]. These techniques are essential for understanding the system's behavior under hardware and software fault situations. Primarily, these techniques aim to replicate the effects of faults that may be hard to replicate in large or complex systems [21].

With the growth of ML applications, understanding the impact of faults on hardware and software for these applications became crucial. Recent works have proposed fault-injection tools and frameworks to facilitate replicating fault effects on ML models [22], [23], [24], [25], [26]. All of these tools or frameworks facilitate the evaluation of ML models' behaviors when encountering faults or atypical situations. The PytorchFI proposed by Mahmoud et al. [22] is an open-source tool that is among the most adopted tools to perturb the models and generate faulty model versions. It allows users to perform perturbations in both weights and neurons in deep neural networks (DNNs) convolutional operations during the models' execution. These perturbations on the DNN models can represent different types of faults in ML modes, such as bit-flips, memory errors, and adversarial attacks. For instance, Piazzesi et al. [27] use PytorchFI to generate faulty ML models and evaluate how an AV would drive when the ML model that performs object detection encounters different types of faults. Wen et al. [28] utilize PyTorchFI to generate compromised models, enabling an evaluation of the impact of the multi-version perception system on AV driving safety.

C. *Software rejuvenation & DSPN modeling*

Software rejuvenation [29] is a well-known maintenance technique to avoid system degradation and help increase system availability and reliability. Traditional software reliability and rejuvenation modeling are well-established in the literature, especially for virtualized and cloud environments [30]. Petri nets and their extensions, such as DSPN and Stochastic Reward Nets (SRN), have been successfully employed to model and evaluate fault-tolerant systems [31] and rejuvenation techniques [32]. Studies have adopted this modeling technique to analyze the impact of aging-related bugs in software systems [33] and how rejuvenation mechanisms help mitigate them [34]. However, a few works consider this technique as an instrument to prevent ML system degradation and improve output reliability [14].

DSPNs are excellent for modeling failures and time-based rejuvenation behaviors. They allow stochastic transitions representing events with some time variance, such as failures, following an exponential time distribution. DSPNs also allow deterministic transitions that can be applied regularly to trigger a rejuvenation operation. DSPNs follow the regular Petri nets notation. Tokens are kept in places, and transitions firing consume tokens from one place, generating new tokens in another place. We denote tokens by small black circles or numbers, places by white circles, and transitions by rectangles. Immediate transitions are thin black, stochastic transitions are white and deterministic transitions are bold black rectangles. Arcs (represented by arrows) connect places and transitions. It defines tokens' flow through the places and can have weights. Inhibitor arcs (represented by an arrow ending with a small white circle) can disable a transition when its weight is met. We refer the reader to the work [35] for further details.

III. NN Vulnerabilities and Fault Model

Similarly to any other software, neural networks (NNs) are subject to faults and threats that could compromise their output. Different types of faults can affect NNs *permantely* or *temporarily*. Two main accidental fault models have been widely explored for NNs: i) fixed logic values (i.e., permanent faults), representing situations where permanent defects in the transistor and interconnection structures cause bits to get stuck, and ii) random bit-flips (i.e., transient faults), representing situations where external perturbations alter the data representing potentials at logic, register, or memory level [36]. Both types can cause misclassification and compromise the applications that depend on the NN being correct.

Threats to NN include poisoning, adversarial, model extraction, and model inversion attacks, just to list a few from the work [37]. Threat models considering those types of attacks have been widely studied. For instance, many works have explored adversarial attacks and proposed defenses against them [38], [39], [40]. However, although less explored, the security of NNs can be further harmed through vulnerabilities in deep learning frameworks (e.g., PyTorch, TensorFlow, or Caffe). For example, Xiao et al. [41] presented vulnerabilities in such frameworks that could allow attackers to (1) launch denial-of-service attacks, (2) crash deep learning applications due to memory exhaustion, (3) generate wrong classification outputs by corrupting the classifier's memory, or (4) hijack the control flow to remote control the deep learning application hosting system. The latest CVE reports on Tensorflow (CVE-2023-27506, CVE-2023-25668), PyTorch (CVE-2022-45907), and Caffe (CVE-2021-39158) confirm the presence of such vulnerabilities.

Torres-Huitzil and Girau [36] review a large body of work that focused on passive methods to tolerate the above faults in NNs (e.g., by replicating critical neurons, noise injection during training, and by optimizing for tolerance). However, passive methods cannot tolerate framework or implementation faults, which is why, in this work, we shall focus on active methods.

We assume that ML modules can fail for arbitrary, potentially intentionally malicious reasons, and they may exhibit arbitrary failure characteristics in doing so. This includes random bit flips and memory corruption caused by high energy particles striking the microelectronic circuit [36], [42], but also cyberattacks leveraging vulnerabilities in ML frameworks [41] or attacks presenting adversarial samples. In this work, we do not address model extraction and inversion attacks, as they operate through different mechanisms. To a limited extent, majority voting and proper isolation of the ML modules hinder the exfiltration of information, even if an NN or its implementation has been compromised. Our architecture includes components — a voter and rejuvenation mechanism — that we assume will not fail or be attacked for simplicity. Rejuvenation leverages operating system mechanisms to start, isolate, and connect ML modules. Gouveia et al. [43] demonstrate the resilient design of such mechanisms.

IV. Multi-version ML Systems

Active fault-tolerance methods, such as error detection and recovery, masking and reconfiguration, bear the potential to tolerate and recover from all kinds of NN faults, the latter two even without understanding the nature and being able to detect the fault. In this section, we, therefore, introduce our active-replication-based multi-version ML architecture for tolerating arbitrary accidental and malicious faults in ML modules by exposing only the majority result while rejuvenating ML modules to retain this healthy majority.

Figure 1 shows our proposed architecture and its components. Multiple different sensors feed into the ML modules, which internally may select a subset of these inputs [44] or be prepared to detect and sideline faulty sensors. ML modules can execute a combination of NNs and other decision modules to perform the classification and prediction task at hand. For example, for autonomous driving, an ML module may include NNs for vehicle, passenger and obstacle detection, traffic sign detection, face-based intention prediction (e.g., to learn if pedestrians wish to cross the road), and they may invoke them in response to prior NN outputs (e.g., dependent on the number of pedestrians detected).

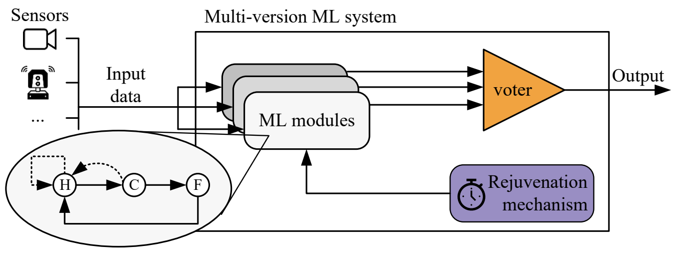

Fig. 1: Multi-version ML systems with time-triggered proactive rejuvenation. ML modules can be healthy (*H*), compromised but still functional (*C*), and non-functional (*N*). Dashed transitions (i.e., C $\rightarrow$ H and H $\rightarrow$ H) represent proactive rejuvenation.

A trusted voter collects the individual outputs from the ML modules and decides, based on pre-defined rules, what the final output should be. This way, arbitrary faults in one ML module can be tolerated and compensated by the other healthy ML modules, which meet the rules of the voter and agree to some extend with the outputs they generate, even if these outputs differ slightly. Possible voting schemes include simple majority (e.g., 2-out-of-3) [10], unanimity (e.g., 3-out-of-3) [8] or approximate versions of the above [45].

To maintain this healthy majority, the rejuvenation mechanism restarts and possibly diversifies one random ML module per time to return it to a healthy state (*H*). In this work, we focus on modeling rejuvenation (but using simple diversification of ML modules) and on time-triggered proactive rejuvenation, which also rejuvenates a module while it is still healthy, to avoid missing compromised modules in case detection fails. In particular, rejuvenation recovers an ML module from ongoing cyberattacks, for example, by reloading and redeploying an ML module from a safe memory location. Although an ML module cannot process sensor data while rejuvenating, the entire system may profit from embracing a proactive rejuvenation mechanism (in addition to a reactive one for recovering non-functional modules from N $\rightarrow$ H) as it can recover the total capacity of an ML module. Gouveia et al. [43] illustrate how to replicate the Operating system (OS) functionality required by such rejuvenation mechanisms.

For the remainder of this paper, we consider a multi-version ML system composed of three ML modules, a trusted voter, and a rejuvenation mechanism. That is, we reason about reliability under the condition that the voter and rejuvenation mechanism continues to operate correctly, while ML modules may transition from healthy (*H*) to compromised but functional (*C*) to non-functional (*N*) states. Adversaries having compromised an ML module seek to retain its responsiveness while subverting the output this module produces. This is to avoid widespread detection mechanisms that, for example, suspect a module is faulty if it does not respond by its deadline. We shall model such behavior as a possibility to remain compromised (i.e., in *C*-state) while failure to respond triggers detection and reactive recovery (of the modules in *N*-state). We apply the following voting rules:

R.1 when the three modules are operational, the voter needs two correct inputs to classify the environment correctly (i.e., 2-out-of-3 voting). Thus, output error occurs only when the voter receives two or more wrong inputs from the modules.

R.2 when only two modules are operational, the voter also needs two correct inputs to classify the environment correctly. In the case the modules diverge on the input, the voter *safely skips* the decision because it cannot give an output with confidence. Output error occurs only when the voter receives two similar wrong inputs from the modules.

R.3 the voter accepts only one input when there is only one operational module. In the case the input is correct, it will classify the environment correctly.

In addition to the above, Machida et al. [46] found that multi-version ML systems have a high correlation of inference errors. We shall, therefore, consider that the misclassification of samples has an error probability dependency α between the different versions of the currently active ML modules. Misclassification of ML modules (e.g., m_x, m_y) in a healthy state (*H*) is therefore considered to be dependent with a factor of $0 \leq \alpha \leq 1$.

V. Modelling Multi-Version ML Systems Reliability

This section presents the DSPN models and reliability functions we have developed to evaluate the reliability of multi-version ML systems and multi-version perception in particular. We describe the models, the time-triggered proactive rejuvenation mechanism, and the reactive rejuvenation mechanism we deploy to return them to a healthy state. We formulate reliability functions for singular, dual- and triple-modular redundant ML systems. In this paper, we focus on quantitatively studying the influence of parameters such as rejuvenation interval, duration, mean time to compromise, error probability dependency, and reliability when ML modules are in healthy or compromised states. For that reason, we limit the number of ML modules to $n = 3$. In contrast to the existing work [14], we define reliability functions for three version systems and experimentally evaluate them using a traffic sign dataset.

A. DSPN models

We leverage DSPNs to represent failure behaviors, reactive rejuvenation, and time-triggered proactive rejuvenation in our evaluation of the reliability of multi-version ML systems. According to our fault model (see Section III), we assume that the voter and the rejuvenation mechanism are always correct. We model time-triggered proactive rejuvenation to recover at most one ML module at a time and to randomly select which module to rejuvenate. Reactive rejuvenation is triggered by the voter not receiving a proposal and rejuvenates non-functional modules (in *N* state) one at a time. We model reactive rejuvenation to take precedence over proactive rejuvenation. That is, while non-functional modules remain, no healthy

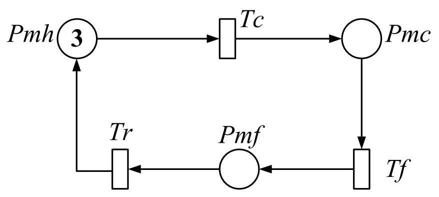

Fig. 2: DSPN for a multi-version ML system not employing a rejuvenation mechanism.

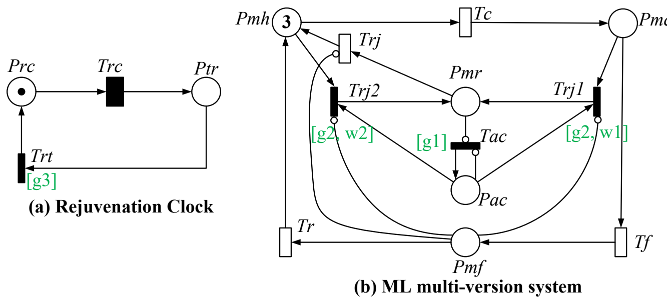

Fig. 3: DSPN for a multi-version ML system containing three ML modules and adopting a time-triggered rejuvenation mechanism.

or compromised module will be rejuvenated. DSPN models mainly adopt exponential distributions as per generalization and for a good representation of stochastic behavior of timing events. Note that other types of distribution can also be adopted.

Figure 2 presents the DSPN for a system with *3* ML modules subject to failures, attacks, and reactive rejuvenation only (i.e., reactive rejuvenation is limited to non-functional modules). The place *Pmh* denotes the number of healthy ML modules, place *Pmc* the number of compromised and place *Pmf* the number of non-functional modules. Since $n = 3$, we instantiate the model with 3 tokens in *Pmh*. The exponential transition *Tc* represents attacks or failures that compromise an ML module but leave it responsive, possibly with wrong outputs. When *Tc* fires, one token is consumed from place *Pmh*, and a new token is generated in place *Pmc*, meaning that one ML module went from state *H* to *C*. *Tc* firing corresponds to the time it took an adversary to find and utilize a vulnerability through which a module could be compromised. *Tf* is enabled if at least one token resides in *Pmc*. When *Tf* fires, a token is consumed from *Pmc* and placed into *Pmf*, representing the change from state *C* to *N* of an ML module. The fact that this module is no longer responding to inference requests. This might occur in response to accidental faults or attacks that crash the module rather than leaving it operational and will trigger the detection mechanism for rejuvenating that module. Once triggered, this mechanism will proceed with the rejuvenation, ignoring further outputs, until the module is returned to a healthy state. The transition *Tr* models this reactive rejuvenation mechanism, which moves one token from *Pmf* to *Pmh*.

Figure 3(a) shows the model of a time-trigger for proactive rejuvenation, and Figure 3(b) the integration of this mechanism into the multi-version ML system shown in Figure 2. Time-triggered proactive rejuvenation starts with a token in place *Prc*, which enables the deterministic transition *Trc*. *Trc* fires after the rejuvenation interval of length $1/\gamma$, where γ is rate at which rejuvenations should happen. Next, *Trc* consumes the token in *Prc* and places a token in *Ptr* to indicate the triggering for a rejuvenation action. Transition *Trt* models the resetting of the clock to trigger the next rejuvenation action.

TABLE I: Guard functions and arc weights for the DSPN models of Figures 3 (a) and (b).

Transition	**Guard/Weight**	**Enabling Function/Value**
Tac	g1	*#Ptr = 1*
Trj1, Trj2,	g2	*(#Pmf + #Pmr) < 1*
Trt	g3	*(#Pmr + #Pac) > 0*
Trj1	w1	*IF (#Pmc = 0): (0.00001) ELSE (#Pmc/(#Pmc + #Pmh));*
Trj2	w2	*IF (#Pmh = 0): (0.00001) ELSE (#Pmh/(#Pmc + #Pmh));*

We connect the multi-version ML model of Figure 3 with the rejuvenation clock in Figure 3 (a) by means of *guards*. Guards refer to Boolean functions over the current marking. They enable transitions to fire, provided the Boolean function returns *true*. The presence of a token in place *Ptr* satisfies the guard function *g1*, which enables the transition *Tac* (in the DSPN in Figure 3 (b)). When *g3* is satisfied, the transition *Trt* can fire, generating a token in *Prc*, which resets the clock into the state where it waits for the next rejuvenation to happen after $1/\gamma$. Table I lists all guard functions and weights used in the DSPN models.

Apart from the proactive rejuvenation mechanism, the operation of the DSPN for the three-version ML system of Figure 3 (b) is similar to the one described in Figure 2. Proactive rejuvenation works as follows. When the transition *Tac* fires, it generates a token in the place *Pac*, where it enables either *Trj1* or *Trj2*, to start the rejuvenation process. The firing of *Trj2* means a healthy ML module will be rejuvenated, while the firing of *Trj1* indicates a compromised ML module will be rejuvenated. However, the firing of *Trj1* or *Trj2*) can only happen when there is no faulty ML module (i.e., no tokens in place *Pmf*). The inhibitor arcs from *Pmf* to *Trj1* and *Trj2* model this behavior to ensure reactive rejuvenation precedes proactive rejuvenation. Since choosing what ML module to proactively rejuvenate is random, the weight functions *w1* and *w2* define the firing probability of the system choosing a healthy or compromised module to be rejuvenated. The firing of *Trj1* or *Trj2* generates a token in place *Pmr*, enabling *Trj*, symbolizing an ML module being proactively rejuvenated. The firing of *Trj* completes the rejuvenation procedure, which generates a token in place *Pmh*.

B. Reliability functions

In the following, we describe the reliability functions for our triple-modular redundant ML system with reactive and time-triggered proactive rejuvenation as well as for a single- and dual-module system, which we use as baselines.

Let $R_{i,j,k}$ be the output reliability of a multi-version ML system, where i, j, and k represent the numbers of ML modules in healthy, compromised, and non-functional states, respectively. When we assume an ML module can sustain its output reliability even under a compromised state, $R_{i,j,k}$ can be computed by the general reliability model for redundant systems, considering independent or dependent (see Eq. 1 and 2) errors to compute the failure probability. However, such an assumption is impractical as a compromised ML module should be more likely to provide a wrong output. Therefore, instead of relying on the equivalent failure probability p, we define $p'(> p)$ as the output failure probability in a compromised state.

In our proposed system, the states of ML modules are changed due to events such as attacks, failures, and rejuvenation. Thus, the system's expected reliability also depends on the current system state. Define S as the set of reachable states in the DSPN models presented in Section V-A. A state of a multi-version system can be represented as $(i, j, k) \in S$ with i, j, and k defined as above. The steady-state probability $\pi_{i,j,k}$ can be computed by solving the DSPN models.

Then, the expected system output reliability is given by assigning $R_{i,j,k}$ as the rewards for the individual states:

$$\mathbb{E}[R_{sys}] = \sum_{(i,j,k)\in S} \pi_{i,j,k} R_{i,j,k} \tag{3}$$

1) Reliability functions for a single-version system: A single-version system is the baseline of the considered architecture. Once the single module fails, the entire system becomes unavailable. The expected reliability of a single-version system can be computed with $R_{1,0,0}$ and $R_{0,1,0}$. When the ML module is healthy, output reliability is given by $R_{1,0,0} = 1-p$. On the other hand, when the module is compromised, the output reliability is changed to $R_{0,1,0} = 1 - p'$. Therefore, the expected reliability is $\mathbb{E}[R_{1v}] = \pi_{1,0,0}(1-p) + \pi_{0,1,0}(1-p')$.

2) Reliability functions for a two-version system: Following the assumptions defined in Section IV, a dual-version system should output an error when both ML modules input the same incorrect input into the voter. Note that the system degrades to a single-version system when either one of the modules is in a non-functional state. As the output reliability depends on the combinations of two modules' states, we define the reliability function matrix $\mathbf{R_{f2}}$ whose (i, j) element corresponds $R_{i,j,k}, i, j, k \in \{0, 2\}, k = 2 - (i + j)$ if the state is reachable, otherwise 0.

$$\mathbf{R_{f2}} = \begin{bmatrix} 0 & 1-p & 1-\alpha p \\ 1-p' & 1-[(p+p')/2]\,\alpha & 0 \\ 1-\alpha p' & 0 & 0 \end{bmatrix}. \tag{4}$$

Taking the function $R_{2,0,0}$ as an example, it represents the reliability of having a correct output when there are two modules in a healthy state. The failure probability of a healthy module is defined by p. Since there is an error output dependency of α between the two modules, the reliability for the state is computed as $R_{2,0,0} = 1 - \alpha p$. The same holds for $R_{0,2,0}$, where the reliability is given by $1 - \alpha p'$.

Note that defining the boundaries for the reliability function parameters is essential. The reliability model for dependent failure assumes a constant dependency parameter α. As a result, p and α are constrained by the total probability that should not exceed one. In a dual-version system, the probability that at least one version outputs error is given by $2p - p\alpha$. Since the probability cannot exceed one, we have $p(2-\alpha) \leq 1$ as the boundary for the parameters α and p.

3) Reliability functions for a three-version system: We employ a majority voting scheme in a three-version architecture to determine the final system output. It means the system outputs an error when two or more ML modules output errors simultaneously. The system degrades to a two-version system when one module is non-functional. Similar to the two-version system, the reliability of the three-version system in different states is represented by the reliability function matrix $\mathbf{R_{f3}}$ whose (i, j) element corresponds $R_{i,j,k}, i, j, k \in \{0, 3\}, k = 3 - (i + j)$ if the state is reachable, otherwise 0.

$$\mathbf{R_{f3}} = \begin{bmatrix} 0 & 1-p & 1-\alpha p & R_{3,0,0} \\ 1-p' & 1-[(p+p')/2]\,\alpha & R_{2,1,0} & 0 \\ 1-\alpha p' & R_{1,2,0} & 0 & 0 \\ R_{0,3,0} & 0 & 0 & 0 \end{bmatrix}, \tag{5}$$

$$\begin{aligned} R_{3,0,0} =& 1 - [3\alpha p(1-\alpha) + \alpha^2]p, \\ R_{2,1,0} =& 1 - [\alpha p + \alpha(p+p')(1-(p+p')/2)], \\ R_{1,2,0} =& 1 - [\alpha p' + \alpha(p+p')(1-(p+p')/2)], \\ R_{0,3,0} =& 1 - \left[3\alpha p'(1-\alpha) + \alpha^2\right] p'. \end{aligned}$$

Lastly, the probability that at least one module outputs an error in a three-version system is given by $3p(1-\alpha) + p\alpha^2$. Therefore, we have $p(3(1-\alpha) + \alpha^2) \leq 1$ as the boundary of the parameters α and p.

VI. Experimental Confirmation for the Models and their Feasibility for Reliability Analysis

This section presents our experimental and numerical analysis of the introduced DSPN models and their reliability functions to demonstrate their feasibility in analyzing multi-version ML module reliability. First, we fine-tune the parameters of the reliability functions and DSPN models to reflect the accuracy of different ML models on a real-world dataset for traffic sign detection. Then, we utilize these parametrized models to evaluate scenarios with one, two, or three ML modules with and without proactive rejuvenation. Finally, we discuss the insights and implementation implications of the different configurations.

A. Fine-tuning with a Real-world Traffic Sign Dataset

To find and test the input parameters of our reliability models, we trained three well-known NN models — AlexNet [47], LeNet [48] and ResNet50 [49], [50] — on the German Traffic Sign Recognition Benchmark (GTSRB) [51], a dataset with real-world images of traffic signs. The dataset is already divided into a training and test set. We used 80% of the images from the former for training and the remaining 20% for validation during training.

We implemented and trained the models using PyTorch 2.1.1 [52] for 20 epochs and with a batch size of 128 and learning rate of 0.001, fixing the *seed* of the *torch*, *numpy*, and *random* libraries to 38 for reproducibility.

After training, we used PyTorchFI [22] to inject *weight faults* with the *random_weight_inj* function using a range of (-10,30) and seeds equal to 5, 183, and 34 for AlexNet, LeNet, and ResNet50, respectively. This is to obtain reduced accuracy models to represent compromised or accidentally faulty versions. Table II shows the obtained accuracies when evaluating the models with the GTSRB test images.

TABLE II: Accuracy of healthy and compromised models when classifying the GTSRB dataset.

Model	Accuracy healthy	Accuracy compromised
1. Alexnet	0.960095012	0.755423595
2. Resnet50	0.920981789	0.772050673
3. LeNet	0.930245447	0.751306413

Based on these accuracy results, we computed the values for the parameters of the DSPN models and the reliability functions p, p', and α. p is the complement of the average accuracy of the healthy models:

$$p = 1 - \underset{i \in \{1,2,3\}}{average}(accuracy\_healthy(i)) \tag{6}$$

where i is the model number. Therefore, the computed value was $\boldsymbol{p = 0.062892584}$. Next, p' is the complement of the average accuracy of the compromised models:

$$p' = 1 - \underset{i \in \{1,2,3\}}{average}(accuracy\_compromised(i)) \tag{7}$$

The computed value in this case was $\boldsymbol{p' = 0.240406440}$. Lastly, α depends on the error set E_i of each model and on the intersection of these sets. To obtain these error sets, we analyzed which input images from the testing set were wrongly classified. For the three-version ML system, we compute α by Eq. 8 and 9.

$$\alpha_{i,j} = \frac{|E_i \cap E_j|}{\max(|E_i|, |E_j|)} \quad | \; i \neq j \tag{8}$$

$$\alpha = \frac{\alpha_{1,2} + \alpha_{1,3} + \alpha_{2,3}}{3} \tag{9}$$

The computed value for α was $\boldsymbol{\alpha = 0.444277551}$.

With α, p, and p' in place, we can compute the reliability functions in Eq. 5, representing the expected reliability when evaluating a three-version ML systems with the module variants AlexNet, LeNet, and ResNet50 against the entire GTSRB testing set (of 12630 images). To achieve this, we implemented the voting rules from Section IV to evaluate the reliability with which the ML system produces the correct outputs. We report the evaluation for each of the different states through which the model can transition based on our reliability function in Table III. Recall that such states (i, j, k) are represented by the number of healthy i, compromised j, and non-functional k ML modules.

TABLE III: Output reliability results of the reliability functions defined in Section V-B.

System state	Reliability functions
(3,0,0)	0.988626295
(2,0,1)	0.976732729
(2,1,0)	0.881542506
(1,0,2)	0.937107416
(1,1,1)	0.943896878
(1,2,0)	0.815870804
(0,3,0)	0.926682718
(0,2,1)	0.911061026
(0,1,2)	0.759593560

B. DSPN-based Reliability Analysis

Based on the expected reliabilities values for individual system states, we can estimate the reliability of a three-version system by the DSPN model. We perform parameter studies and analyze the impact of the individual parameters on the overall system reliability.

We employ TimeNET [53] to run and analyze the DSPN models and configure it with the parameters in Table IV. We follow Oboril et al. [54] and their estimation of the mean time to compromise an AV module ($1/\lambda_c = 1523s$) and mean time to failure ($1/\lambda = 1523s$) for the transitions *Tc* and *Tf*. These parameters are naturally hard to estimate and highly scenario-dependent. Our interpretation of these parameters is in terms of the distributions of time that an adversary needs to exploit a vulnerability and harm the ML module without crashing it. Automated diversification techniques, such as address space layout randomization, justify why this distribution cannot degenerate to a fixed small amount, since the adversary will always have to adjust its exploit to the current layout. We estimate a recovery time of $0.5s$ and shall use this as the mean time to repair a module during reactive or proactive rejuvenation. We adopt a default value of $300s$ for the rejuvenation interval ($1/\gamma$), but will study the impact of different values for this parameter.

TABLE IV: Default input parameters for the DSPN models.

Param	Description	Transition	Value
α	Error probability dependency	-	0.444277
p	Output failure probability (healthy)	-	0.062892
p'	Output failure probability (compromised)	-	0.240406
$1/\lambda_c$	Mean time to compromise a module	*Tc*	1523 s
$1/\lambda$	Module's mean time to failure	*Tf*	1523 s
$1/\mu$	Mean time to reactive rejuvenate	*Tr*	0.5 s
$1/\mu_r$	Mean time to proactive rejuvenate	*Trj*	0.5 s
$1/\gamma$	Rejuvenation interval	*Trc*	300 s

Table V presents the reliability result for the different configurations obtained by instantiating the models of Figure 2 and 3 with the above parameters and the reliability functions $\mathbb{E}[R_{1v}]$, $\mathbb{E}[R_{2v}]$, and $\mathbb{E}[R_{3v}]$ as reward metrics. As can be seen, the two-version system with proactive rejuvenation outperforms all other configurations. This is due to the voter being allowed to skip producing an output if the functional networks disagree. Proactive rejuvenation proves beneficial in any of the configurations.

TABLE V: Reliability results through DSPN simulation for a single-, two- or three-version with and without proactive rejuvenation.

	Reliability	
Configuration	*w/o rej.*	*w/ rej.*
Baseline (single-version)	0.848211	0.920217
two-version	0.943875	0.967152
three-version	0.903190	0.952998

C. Parameter Study

Next, we analyze the impact different values of the parameters have on the reliability of the different configurations. Figure 4 shows the sensitivity of the reliability of the individual configurations to variations of the rejuvenation interval, duration, mean time to compromise, error dependency, inaccuracy in the healthy state, and inaccuracy in the compromised state.

Proactive Rejuvenation is characterized by the rejuvenation interval $1/\gamma$ and by the time $1/\mu_r$ it takes to rejuvenate a module. Figures 4 (a) and (b) present the reliability behavior when varying these parameters, while keeping all other parameters as in Table IV. As can be seen, more frequent rejuvenation helps the system to keep a high reliability, whereby the influence on the three-version and single-version configuration are most significant. In the latter two, reliability quickly decreases as rejuvenation periods become longer. In contrast, variations of the rejuvenation time $1/\mu_r$ have only a minimal effect on reliability (see Figure 4 (b)). Another insight is that even when varying the parameters $1/\gamma$ and $1/\mu_r$, the two-version system outperforms a three-version system.

Mean time to compromise/degrade a module ($1/\lambda_c$). The impact of strenghtening or weakening the adversary by varying the mean-time-to-compromise in the range of [100s, 7000s] is shown in Figure 4 (c). As expected, higher values for $1/\lambda_c$ imply higher reliability for the system, as the models remain in a healthy state for a longer time. Particularly, the single-version systems are positively impacted when $1/\lambda_c$ grows, having the reliability improved by more than 12% and 14% for the version with proactive rejuvenation compared to reactively-only rejuvenated systems. The three-version system without rejuvenation presented a different behavior, with its reliability decreasing at the beginning (from the 100s - 1000s) and starting to increase again after the 1000s. This behavior may be explained by the fast transition of healthy ML modules to a compromised state, and at the same time, these modules spend extended time in the compromised state before transitioning to the non-functional state to be recovered.

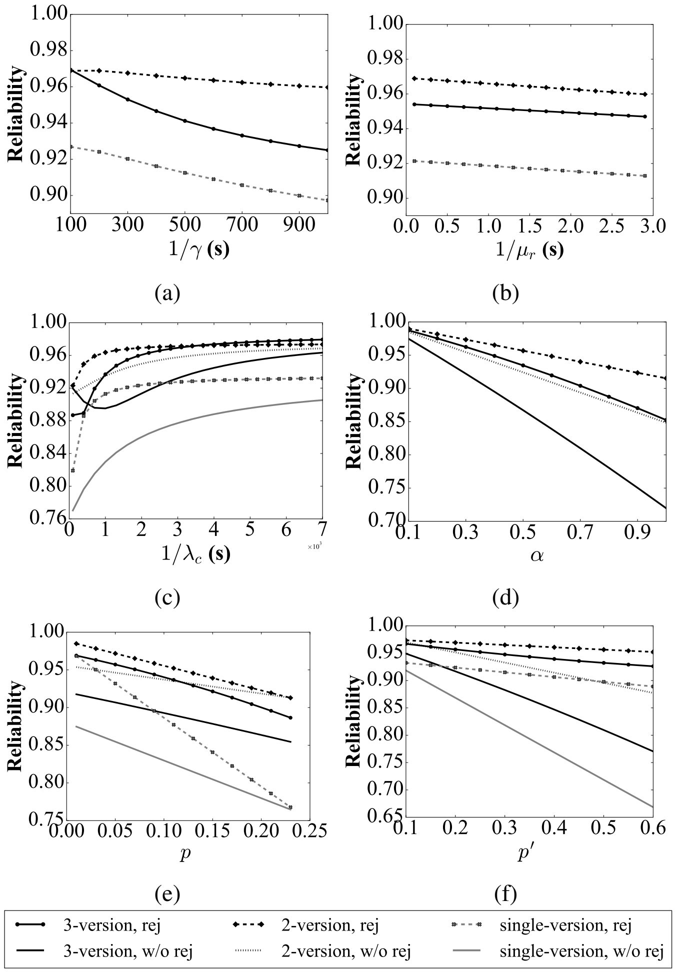

Fig. 4: Influence of (a) rejuvenation interval, (b) rejuvenation duration, (c) mean time to compromise/degrade a module, (d) error probability dependency between modules, (e) ML modules inaccuracy when in a healthy state, and (f) ML modules inaccuracy when in a compromised state over the expect output reliability.

Error probability dependency between modules (α). Figure 4 (d) shows the sensitivity of the configurations to variations in the error probability dependency between modules. We varied α in the range of [0.1, 1.0], where $\alpha = 0.1$ means lower error dependency and $\alpha = 1.0$ means higher error dependency. Note that a single-version system does not suffer any influence from this parameter since all outputs originate from a single module, which processes all inputs. As expected, less error dependency positively impacts the system's reliability, especially when adopting time-based proactive rejuvenation. In all scenarios, systems adopting such a rejuvenation mechanism outperform those with reactive rejuvenation only. The latter is seriously impacted as α grows. The reliability of the two-version and three-version without rejuvenation drops by about 13% and 26% when varying α from 0.1 to 1.

ML modules inaccuracy when in a healthy state (p). We analyzed this parameter for a range of [0.01, 0.23] since one of the constraints for the reliability functions is $p < p'$. Figure 4 (e) shows the results, which confirm that employing proactive rejuvenation is beneficial in all scenarios. However, if the module inaccuracy is high (e.g., > 0.20, the reliability improvement is marginal, and one may examine the trade-offs of reliability improvement and the complexity of adding such a mechanism to the system. Through the plot, it is also possible to visualize a threshold where rejuvenation would be beneficial, what happens when the value of p is below the intersection of lines. Note that this threshold differs according to the other input parameters, but using the proposed modeling approach, it is possible to investigate it. According to the results and considering the input parameters adopted, a single-version system adopting rejuvenation performs better than a three-version system without rejuvenation when $p < 0.10$. A similar situation occurs when $p > 0.13$, the 2-version system without rejuvenation outperforms the three-version system adopting rejuvenation. Current state-of-the-art ML models usually have an accuracy greater than 0.9 (i.e., $p < 0.10$) for most use cases. Thus, implementing a time-based rejuvenation mechanism would be attractive based on these results. In any case, system designers could leverage this analysis to define a more suitable system architecture.

ML modules inaccuracy when in a compromised state (p'). We performed the same analysis by varying the inaccuracy of compromised models in the range of [0.1, 0.6]. Figure 4 (f) shows the sensitivity of reliability to variations of this parameter. The plot shows that reliability in all system configurations drops as p' grows. However, adopting proactive rejuvenation helps mitigate some of this reliability degradation. While the reliability of systems adopting proactive rejuvenation dropped less than 4%, the negative impact on the reliability of systems with reactive rejuvenation was only more than 10%. The most harmed configuration by this change of p' from 0.1 to 0.6 was the single-version ML system without proactive rejuvenation. Here reliability dropped by 27%.

Parameter studies like the above allow us to configure different multi-version ML systems based on the achieved accuracy in a healthy state and based on the expected deterioration when compromised. In the following we apply these findings in an actual implementation of a multi-version ML system for the perception in an AV.

VII. Multi-version Perception System with Rejuvenation: A Case Study

To evaluate the practicality of our approach, we implemented a multi-version perception system with rejuvenation for AVs and conducted the analysis using the CARLA AV simulator [55]. Specifically, we focus on assessing the impact of rejuvenation of a multi-version perception system for *driving safety*. Through our empirical study, we address the following research questions:

RQ1: How effectively can a multi-version perception system with time-triggered rejuvenation tolerate compromised and non-functional ML models?

RQ2: How does the rejuvenation interval impact the safety of a multi-version perception system?

To address *RQ1*, we simulate a three-version perception system incorporating a time-triggered rejuvenation mechanism. Compromised ML models are generated using PyTorchFI, by introducing artificial faults into the ML models, which simulate the effects of an attack or transient fault. We then compare the AV driving behavior across eight scenarios with rejuvenation to those without rejuvenation to assess its impact on AV driving safety. To investigate *RQ2*, we vary the rejuvenation intervals in the simulation and analyze the resulting differences in AV driving behavior to evaluate the influence of rejuvenation frequency on system safety.

A. Experiments Setup

We utilize the CARLA AV simulator and the cooperative driving co-simulation framework, OpenCDA [56], to simulate various driving scenarios. In the OpenCDA simulation, each AV is equipped with sensors that capture data from both the surrounding environment and the ego vehicle, such as 3D LiDAR points and Global Navigation Satellite System (GNSS) data. This collected data is processed by the perception and localization systems, enabling the detection and localization of objects. The resulting perception and localization outputs are passed to the planning system. The planning system uses this data to compute the AV's trajectory, adjusting the vehicle's acceleration, speed, and steering based on the generated path. The final trajectory and control instructions are then forwarded to the control system, which executes the necessary commands to control the vehicle's movements. In this study, we focus on the perception system. We select eight routes, with two routes chosen from each of the maps Town02, Town03, Town04, and Town05 in CARLA, as shown in Figure 5. The starting points of the routes are marked with ovals, while the endpoints are indicated by stars. During these simulation runs, the ego AV must navigate based on its perception system, which is responsible for accurately detecting other vehicles and road obstacles. This setup allows us to evaluate the performance of multi-version perception systems with time-triggered rejuvenation under diverse scenarios and road configurations.

We construct *healthy models* by deploying variants of the YOLOv5 model [57], including YOLOv5s6, YOLOv5m6, and YOLOv5l6. The different variants enable us to simulate the multi-version perception frameworks. We then *compromise* these models with PyTorchFI [22], leveraging PyTorchFI's runtime perturbation feature for weights and neurons in DNNs. This functionality is crucial for simulating real-world scenarios where models may encounter unexpected disruptions. Like above, we employed PyTorchFI's *random_weight_inj* function with a weight range of (-100, 300) to mimic the conditions compromised models encounter, before assessing the multi-version perception system under these perturbed states.

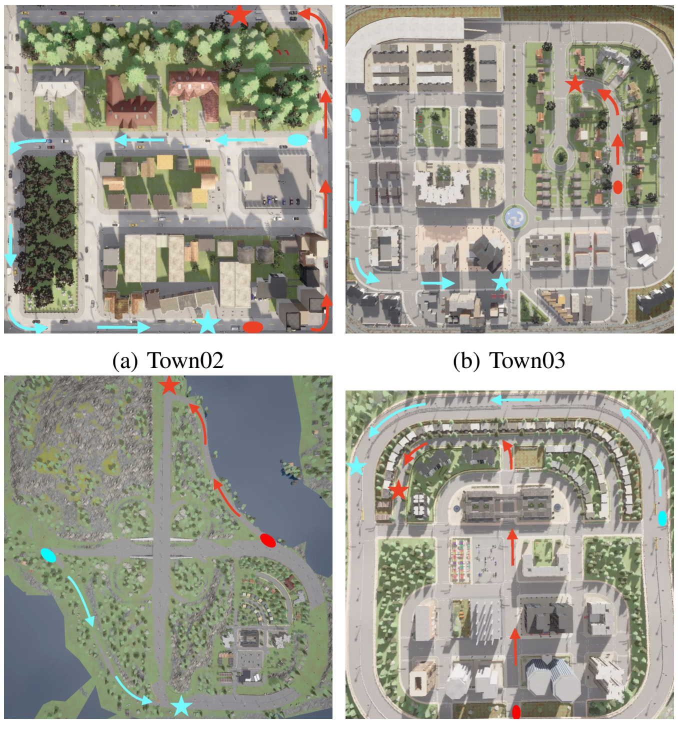
(a) Town02 (b) Town03 (c) Town04 (d) Town05

Fig. 5: Adopted maps and routes in Town02, Town03, Town04, and Town05 of CARLA simulator.

Parameters: In the simulation, each run is executed in one of the routes and lasts approximately 30 seconds from the starting point to the endpoint. Based on Oboril et al. [54], where the MTBF is 0.4 hours in 1.4 hours of recording, we set the mean time to compromise a module ($1/\lambda_c$) to 8 seconds. The module's mean time to failure ($1/\lambda$) is defined as twice the meantime to compromise, set to 16 seconds. Both the mean times to reactive ($1/\mu$) and proactive rejuvenation ($1/\mu_r$) are configured to 0.5 seconds. The default rejuvenation interval ($1/\gamma$) is set to 3 seconds, with further experiments conducted to investigate the effects of varying this parameter. The perception system begins with three healthy models (i.e., H state), which become compromised sequentially after the defined time to compromise. Subsequently, the models enter the non-functional (N) state after the specified time to failure and are recovered following the time for reactive rejuvenation. Note that we also adopted here exponentially distributed time for the parameters $1/\lambda_c$, $1/\lambda$, $1/\mu$, and $1/\mu_r$. During these states, time-based rejuvenation is randomly triggered at deterministic intervals defined by the parameter $1/\gamma$, with a 2/3 probability of prioritizing compromised models for rejuvenation. The 2/3 probability was selected to balance the trade-off between prioritizing compromised models and ensuring adequate rejuvenation opportunities for non-compromised models.

As per our fault model (see Section III), we employ *the majority voting rule* in the multi-version perception system. When all three models are operational (i.e., in either the H or C state), the voter produces a perception output if at least two models agree on the result. In the absence of agreement, the voter does not generate any perception output, which prevents the AV from updating its driving properties such as speed, acceleration, or steering. If one model becomes completely non-functional (i.e., enters the N state), the system operates in a two-version configuration. In this scenario, the voting rule requires both remaining models to output the same result for a valid perception output. If the models disagree, no perception output is produced, and its driving properties will remain unchanged.

Evaluation Metric: To compute driving safety, we measure the collision rate as the ratio of collision frames to the total frames. Additionally, we report the frame at which the first collision occurs and the total number of frames as part of our evaluation metrics. Each route is evaluated over five runs, both with and without rejuvenation, and the average values of the metrics, along with the total number of collisions observed across these runs, are recorded. These metrics offer an evaluation of the safety of the three-version perception system with rejuvenation in different scenarios. In this study, our focus is primarily on vehicle-to-vehicle collision scenarios.

B. Evaluation Results

In our study, we investigate the effects of a time-triggered rejuvenation mechanism within multi-version perception systems implemented across eight distinct route scenarios for AVs. We compare the collision data between systems with rejuvenation (*w/*) and those without (*w/o*) to measure driving safety. The experimental results are presented in Table VI. We observe that the system with rejuvenation consistently avoid collisions across all 40 runs in all tested routes, achieving a collision rate of 0%. However, significant collision rates are observed without rejuvenation, ranging from 9.70% to 54.13% across different routes. The number of collisions also increases, with some scenarios recording collisions in all runs. On average, the collision rate without adopting rejuvenations is 33.54%, having a standard deviation of 18.53%, and with the first collision frame happening on average at frame 287. The results demonstrate the effectiveness of the rejuvenation mechanism in enhancing driving safety when adopting multi-version perception systems against compromised and non-functional ML models, thereby significantly reducing the risk of collisions.

Answer to RQ1. A multi-version perception system with time-triggered rejuvenation can efficiently tolerate compromised and faulty ML models, achieving 0% collision rates across all tested routes.

Next, we analyze the effect of varying rejuvenation intervals on the driving safety of multi-version perception systems in AVs, focusing on route #1 in Town02 as a case study. Collision data are compared across rejuvenation intervals ($1/\gamma$) of 3, 5, 7, and 9 seconds. The results of the experiments are depicted in Table VII. The findings reveal that shorter rejuvenation intervals enhance driving safety by facilitating the rapid recovery

TABLE VI: Collision data of the multi-version perception system w/ and w/o rejuvenation over different routes.

Route	1st coll.		Total frames		Coll. rate (%)		#Coll.	
	w/	*w/o*	*w/*	*w/o*	*w/*	*w/o*	*w/*	*w/o*
#1	NA	299	610	618	0.00	9.70	0/5	4/5
#2	NA	268	735	675	0.00	12.89	0/5	3/5
#3	NA	203	630	543	0.00	47.98	0/5	4/5
#4	NA	390	720	730	0.00	42.45	0/5	4/5
#5	NA	313	644	757	0.00	52.25	0/5	5/5
#6	NA	383	663	684	0.00	33.97	0/5	4/5
#7	NA	204	626	661	0.00	14.91	0/5	4/5
#8	NA	241	630	680	0.00	54.13	0/5	5/5
Avg/*Total*	NA	287	657	669	0.00	33.54	*0/40*	*33/40*

of compromised models, thereby preventing the accumulation of perception errors. Notably, the first collision frame occurs later as the rejuvenation interval decreases, further supporting the safety benefits of more frequent rejuvenations. In contrast, extended rejuvenation intervals correlate with a higher frequency of collisions, reflecting their negative impact on driving safety, and correlating with less reliable perception outputs. Consequently, the rejuvenation interval plays a critical role in the effectiveness of rejuvenation within multi-version perception systems, with shorter intervals improving driving safety in the case of perception systems for autonomous driving.

Moreover, the first collision frame for the system with rejuvenation also happens a bit later (on average at frame 347) in comparison with the system without rejuvenation, where the first collision happens on average at frame 287 (as shown in Table VI). This indicates the system with rejuvenation could delay the onset of erroneous outputs, providing the AV additional time to respond and potentially avoid collisions.

TABLE VII: Impact of rejuvenation interval on the driving safety.

$1/\gamma$ (s)	1st coll.	Total	Coll. rate	#Coll.
3	NA	610	0.00%	0/5
5	526	627	1.27%	1/5
7	246	574	8.93%	2/5
9	270	632	10.44%	3/5
Avg/*Total*	347	611	5.16%	*6/20*

Answer to RQ2. The rejuvenation interval directly impacts the safety of a multi-version perception system, with shorter intervals enhancing driving safety by facilitating quicker recovery of compromised and non-functional models.

C. Discussion

The observed performance gap in AV driving between the multi-version perception systems *w/* and *w/o* rejuvenation shows the effectiveness of time-triggered rejuvenation as a critical safety mechanism for AVs. The rejuvenation process mitigates the negative impact of disruptions by periodically refreshing the perception modules, thus enhancing the resilience of systems. Additionally, the system with rejuvenation shows the potential to delay erroneous perception outputs that could lead to collisions. The ability is particularly significant in real-world AV operations, where rapid decision-making is essential to avoid accidents. It provides the AV with additional time to detect and respond to potential hazards. Moreover, the rejuvenation interval, in particular, plays a critical role in determining the effectiveness of the rejuvenation mechanism. Shorter rejuvenation intervals are more effective at minimizing the accumulation of errors, ensuring the models remain accurate for longer periods, and thereby improving overall driving safety.

VIII. Threats to Validity and Limitations

Obviously the choice of ML models can significantly influence the results, in particular their inaccuracy and error correlation, as we have shown. Our work focused on single-, dual- and triple-version systems employing the defined voting scheme (see Section IV). Evaluating the reliability of multi-version ML systems considering more versions and other voting schemes, such as weighted or approximate voting, theoretically and experimentally is an important direction for future work. In our case study (see Section VII), we focused on a limited set of scenarios within a predefined layout and did not fully account for real-world variability, such as dynamic traffic conditions and environmental changes. Our goal was to demonstrate the feasibility of our methodology, but before generalizing our results to autonomous driving, more scenarios must be investigated. Furthermore, while the rejuvenation interval is shown to be critical, optimizing it for a balance between safety and computational overhead remains a challenge. In particular, we did not consider other critical system design factors, such as performance, resource consumption, energy efficiency, and costs. The costs associated with our approach and the performance overhead are inherently related to the number of models used.

IX. Conclusion

This paper proposed and investigated a multi-version architecture adopting proactive rejuvenation for ML systems. We specifically focus on how combining the two techniques can improve the output reliability of ML systems and mitigate faults and malicious attacks. We used Petri-net-based models to represent failures, possible malicious threats, and the proactive rejuvenation mechanism while integrating them with reliability functions, allowing us to derive the output reliability of different system configurations. We performed experiments using different ML models and a traffic sign dataset using real-world images to fine-tune our models. We evaluated different system configurations using the models and extensively analyzed input parameters to find scenarios where output reliability is maximized. Our results show that two-version systems can outperform three-version systems when the voter can safely skip outputs. They also confirmed that proactive rejuvenation could benefit multi-version ML systems, especially when the ML module accuracy is high.

Lastly, we presented a case study by implementing a three-version perception system with time-triggered rejuvenation in an AV simulator to experimentally evaluate the impact of rejuvenation in a real-world and safety-critical system. Our evaluation showed that a multi-version perception system with time-triggered rejuvenation can efficiently tolerate compromised and non-functional models. In future work, we aim to analyze the adoption of proactive rejuvenation mechanisms in systems with more replicas and under different voting schemes for perception and other systems.

References

[1] D. Gruyer, V. Magnier, K. Hamdi, L. Claussmann, O. Orfila, and A. Rakotonirainy, "Perception, information processing and modeling: Critical stages for autonomous driving applications," *Reviews in Control*, vol. 44, pp. 323–341, 2017.

[2] A. Toschi, M. Sanic, J. Leng, Q. Chen, C. Wang, and M. Guo, "Characterizing perception module performance and robustness in production-scale autonomous driving system," in *IFIP Int. Conf. on Network and Parallel Computing*. Springer, 2019, pp. 235–247.

[3] Apollo Documentation, "Apollo perception," 2019. [Online]. Available: http://www.fzb.me/apollo/specs/perception_apollo_5.0.html

[4] M. A. Hanif, F. Khalid, R. V. W. Putra, S. Rehman, and M. Shafique, "Robust machine learning systems: Reliability and security for deep neural networks," in *Int. Symp. on On-Line Testing And Robust System Design*, 2018.

[5] S. Qiu, Q. Liu, S. Zhou, and C. Wu, "Review of artificial intelligence adversarial attack and defense technologies," *Applied Sciences*, vol. 9, no. 5, p. 909, 2019.

[6] A. Avizienis, "The n-version approach to fault-tolerant software," *Trans. on software engineering*, no. 12, pp. 1491–1501, 1985.

[7] A. Gujarati, S. Gopalakrishnan, and K. Pattabiraman, "New wine in an old bottle: N-version programming for machine learning components." Institute of Electrical and Electronics Engineers Inc., 10 2020, pp. 283–286.

[8] S. Latifi, B. Zamirai, and S. Mahlke, "Polygraphmr: Enhancing the reliability and dependability of cnns," in *IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN)*, 2020.

[9] F. Machida, "Using Diversities to Model the Reliability of N-version Machine Learning System," 2021. [Online]. Available: https://doi.org/10.36227/techrxiv.16435656.v1

[10] Q. Wen and F. Machida, "Reliability models and analysis for triple-model with triple-input machine learning systems," in *IEEE Conf. on Dependable and Secure Computing (DSC)*, 2022.

[11] P. Sousa, N. F. Neves, and P. Veríssimo, "Proactive resilience through architectural hybridization," in *Proceedings of the 2006 ACM Symposium on Applied Computing*, ser. SAC '06, 2006, p. 686–690.

[12] M. Völp and P. Esteves-Verissimo, "Intrusion-tolerant autonomous driving," in *IEEE Int. Symp. on Real-Time Computing*, 2018.

[13] F. Machida, "N-version machine learning models for safety critical systems," in *49th IEEE/IFIP Int. Conf. on Dependable Systems and Networks Workshops (DSN-W)*, 6 2019, pp. 48–51.

[14] J. Mendonça, F. Machida, and M. Völp, "Enhancing the reliability of perception systems using n-version programming and rejuvenation," in *53rd IEEE/IFIP Int. Conf. on Dependable Systems and Networks Workshops (DSN-W)*, 2023, pp. 149–156.

[15] A. Wu, A. H. M. Rubaiyat, C. Anton, and H. Alemzadeh, "Model fusion: weighted n-version programming for resilient autonomous vehicle steering control," in *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2018, pp. 144–145.

[16] H. Xu, Z. Chen, W. Wu, Z. Jin, S.-y. Kuo, and M. Lyu, "Nv-dnn: Towards fault-tolerant dnn systems with n-version programming," in *IEEE/IFIP Int. Conf. on Dependable Systems and Networks Workshops (DSN-W)*, 2019.

[17] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, 1962.

[18] B. Littlewood, P. Popov, and L. Strigini, "Modeling software design diversity: a review," *ACM Comput. Surv.*, vol. 33, no. 2, 2001. [Online]. Available: https://doi-org.proxy.bnl.lu/10.1145/384192.384195

[19] M. Ege, M. Eyler, and M. Karakas, "Reliability analysis in n-version programming with dependent failures." IEEE Comput. Soc, 2001, pp. 174–181.

[20] J. Arlat, A. Costes, Y. Crouzet, J. Laprie, and D. Powell, "Fault injection and dependability evaluation of fault-tolerant systems," *IEEE Trans. on Computers*, vol. 42, no. 8, pp. 913–923, 1993.

[21] M.-C. Hsueh, T. Tsai, and R. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[22] A. Mahmoud, N. Aggarwal, A. Nobbe, J. R. S. Vicarte, S. V. Adve, C. W. Fletcher, I. Frosio, and S. K. S. Hari, "Pytorchfi: A runtime perturbation tool for dnns," in *50th IEEE/IFIP Int. Conf. on Dependable Systems and Networks Workshops (DSN-W)*, 2020, pp. 25–31.

[23] S. Laskar, M. H. Rahman, and G. Li, "Tensorfi+: A scalable fault injection framework for modern deep learning neural networks," in *IEEE Int. Symp. on Software Reliability Engineering Workshops (ISSREW)*, 2022, pp. 246–251.

[24] R. Gräfe, Q. S. Sha, F. Geissler, and M. Paulitsch, "Large-scale application of fault injection into pytorch models -an extension to pytorchfi for validation efficiency," in *2023 53rd IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S)*, 2023, pp. 56–62.

[25] J. Hoefer, F. Kempf, T. Hotfilter, F. Kreß, T. Harbaum, and J. Becker, "Sifi-ai: A fast and flexible rtl fault simulation framework tailored for ai models and accelerators," in *Proceedings of the Great Lakes Symposium on VLSI 2023*, ser. GLSVLSI '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 287–292.

[26] S. Pappalardo, A. Ruospo, I. O'Connor, B. Deveautour, E. Sanchez, and A. Bosio, "A fault injection framework for ai hardware accelerators," in *2023 IEEE 24th Latin American Test Symposium (LATS)*, 2023, pp. 1–6.

[27] N. Piazzesi, M. Hong, and A. Ceccarelli, "Attack and fault injection in self-driving agents on the carla simulator – experience report," in *Computer Safety, Reliability, and Security: 40th International Conference, SAFECOMP 2021, York, UK, September 8–10, 2021, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2021, p. 210–225.

[28] Q. Wen, J. Mendonça, F. Machida, and M. Völp, "Enhancing autonomous vehicle safety through n-version machine learning systems," in *IJCAI Workshop on Artificial Intelligence Safety (AISafety)*. CEUR Workshop Proceedings, 2024. [Online]. Available: https://ceur-ws.org/Vol-3856/paper_8.pdf

[29] K. S. Trivedi, *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. John Wiley & Sons, Ltd, 2016.

[30] R. Pietrantuono and S. Russo, "Software aging and rejuvenation in the cloud: a literature review," in *IEEE Int. Symp. on Software Reliability Engineering Workshops (ISSREW)*, 2018, pp. 257–263.

[31] N. Leveson and J. Stolzy, "Safety analysis using petri nets," *IEEE Trans. on Software Engineering*, vol. SE-13, no. 3, pp. 386–397, 1987.

[32] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "A survey of software aging and rejuvenation studies," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 10, no. 1, pp. 1–34, 2014.

[33] A. M. M. Paing, "Analysis of availability model based on software aging in sdn controllers with rejuvenation," in *IEEE Conf. on Computer Applications (ICCA)*, 2020.

[34] T. Thein and J. Sou Park, "Availability analysis of application servers using software rejuvenation and virtualization," *Journal of computer science and technology*, vol. 24, no. 2, pp. 339–346, 2009.

[35] M. Marsan and G. Chiola, "On petri nets with deterministic and exponentially distributed firing times," in *Advances in Petri Nets 1987*, 1987, vol. 266, pp. 132–145.

[36] C. Torres-Huitzil and B. Girau, "Fault and error tolerance in neural networks: A review," *IEEE Access*, vol. 5, pp. 17 322–17 341, 8 2017.

[37] X. Liu, L. Xie, Y. Wang, J. Zou, J. Xiong, Z. Ying, and A. V. Vasilakos, "Privacy and security issues in deep learning: A survey," *IEEE Access*, vol. 9, pp. 4566–4593, 2021.

[38] A. Chakraborty, M. Alam, V. Dey, A. Chattopadhyay, and D. Mukhopadhyay, "A survey on adversarial attacks and defences," *CAAI Transactions on Intelligence Technology*, vol. 6, no. 1, pp. 25–45, 2021.

[39] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," 2019.

[40] N. Akhtar and A. Mian, "Threat of adversarial attacks on deep learning in computer vision: A survey," *IEEE Access*, vol. 6, pp. 14 410–14 430, 2018.

[41] Q. Xiao, K. Li, D. Zhang, and W. Xu, “Security risks in deep learning implementations,” in *2018 IEEE Security and Privacy Workshops (SPW)*, 2018, pp. 123–128.

[42] P. Dodd and L. Massengill, “Basic mechanisms and modeling of single-event upset in digital microelectronics,” *IEEE Transactions on Nuclear Science*, vol. 50, no. 3, pp. 583–602, 2003.

[43] I. P. Gouveia, M. Völp, and P. Esteves-Verissimo, “Behind the last line of defense: Surviving soc faults and intrusions,” *Computers & Security*, vol. 123, p. 102920, 2022.

[44] J. Kocić, N. Jovičić, and V. Drndarević, “Sensors and sensor fusion in autonomous vehicles,” in *2018 26th Telecommunications Forum (TELFOR)*. IEEE, 2018, pp. 420–425.

[45] D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl, “Reaching approximate agreement in the presence of faults,” *Journal of the ACM (JACM)*, vol. 33, no. 3, pp. 499–516, 1986.

[46] F. Machida, “On the diversity of machine learning models for system reliability,” in *Pacific Rim Int. Symposium on Dependable Computing, PRDC*, 2019, pp. 276–285.

[47] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” *Advances in neural information processing systems*, vol. 25, 2012.

[48] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[49] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[50] Nvidia, “Resnet v1.5 for pytorch,” 2023. [Online]. Available: https://catalog.ngc.nvidia.com/orgs/nvidia/resources/resnet_50_v1_5_for_pytorch

[51] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, “The german traffic sign recognition benchmark: A multi-class classification competition,” in *Int. Joint Conference on Neural Networks*, 2011.

[52] The Linux Foundation, “Pytorch,” 2023. [Online]. Available: https://pytorch.org

[53] A. Zimmermann, “Modelling and performance evaluation with timenet 4.4,” in *Quantitative Evaluation of Systems*, N. Bertrand and L. Bortolussi, Eds. Cham: Springer, 2017, pp. 300–303.

[54] F. Oboril, C. Buerkle, A. Sussmann, S. Bitton, and S. Fabris, “MTBF Model for AVs - From Perception Errors to Vehicle-Level Failures,” in *IEEE Intelligent Vehicles Symposium (IV)*, 2022.

[55] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, “CARLA: An open urban driving simulator,” in *1st Conference on Robot Learning*, 2017, pp. 1–16.

[56] R. Xu, H. Xiang, X. Han, X. Xia, Z. Meng, C.-J. Chen, C. Correa-Jullian, and J. Ma, “The opencda open-source ecosystem for cooperative driving automation research,” *IEEE Trans. on Intelligent Vehicles*, vol. 8, no. 4, pp. 2698–2711, 2023.

[57] G. Jocher, A. Stoken, J. Borovec, NanoCode012, A. Chaurasia, TaoXie, L. Changyu, A. V, Laughing, tkianai, yxNONG, A. Hogan, lorenzomammana, AlexWang1900, J. Hajek, L. Diaconu, Marc, Y. Kwon, oleg, wanghaoyang0106, Y. Defretin, A. Lohia, ml5ah, B. Milanko, B. Fineran, D. Khromov, D. Yiwei, Doug, Durgesh, and F. Ingham, “ultralytics/yolov5: v5.0 - YOLOv5-P6 1280 models, AWS, Supervise.ly and YouTube integrations,” Apr. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4679653