Frontiers in Computer Science

TYPE Methods
PUBLISHED 11 April 2023
DOI 10.3389/fcomp.2023.1125055

OPEN ACCESS

EDITED BY
Ali Shoker,
King Abdullah University of Science and Technology, Saudi Arabia

REVIEWED BY
Fernando Alves,
Vortex-CoLab, Portugal
Savio Sciancalepore,
Eindhovenx University of Technology, Netherlands
Jorrit Olthuis,
Eindhoven University of Technology, Netherlands, in collaboration with reviewer SS

*CORRESPONDENCE
Federico Lucchetti
✉ federico.lucchetti@uni.lu

SPECIALTY SECTION
This article was submitted to Computer Security, a section of the journal Frontiers in Computer Science

RECEIVED 15 December 2022
ACCEPTED 14 March 2023
PUBLISHED 11 April 2023

CITATION
Lucchetti F, Graczyk R and Völp M (2023) Toward resilient autonomous driving—An experience report on integrating resilience mechanisms into the Apollo autonomous driving software stack.
Front. Comput. Sci. 5:1125055.
doi: 10.3389/fcomp.2023.1125055

Toward resilient autonomous driving—An experience report on integrating resilience mechanisms into the Apollo autonomous driving software stack

Federico Lucchetti*, Rafal Graczyk and Marcus Völp

Critical and Extreme Security and Dependability Group (CritiX), Interdisciplinary Centre for Security Reliability and Trust, University of Luxembourg, Luxembourg, Luxembourg

Autonomous driver assistance systems (ADAS) have been progressively pushed to extremes. Today, increasingly sophisticated algorithms, such as deep neural networks, assume responsibility for critical driving functionality, including operating the vehicle at various levels of autonomy. Elaborate obstacle detection, classification, and prediction algorithms, mostly vision-based, trajectory planning, and smooth control algorithms, take over what humans learn until they are permitted to control vehicles and beyond. And even if humans remain in the loop (e.g., to intervene in case of error, as required by autonomy levels 3 and 4), it remains questionable whether distracted human drivers will react appropriately, given the high speed at which vehicles drive and the complex traffic situations they have to cope with. A further pitfall is trusting the whole autonomous driving stack not to fail due to accidental causes and to be robust against cyberattacks of increasing sophistication. In this experience report, we share our findings in retrofitting application-agnostic resilience mechanisms into an existing hardware-/software-stack for autonomous driving—Apollo—as well as where application knowledge helps improve existing resilience algorithms. Our goal is to ultimately decrease the vulnerability of autonomously driving vehicles to accidental faults and attacks, allowing them to absorb and tolerate both, as well as to come out of them at least as secure as before the attack has happened. We demonstrate replication and rejuvenation on the driving stack's Control module and indicate how this resilience can be extended both downwards to the hardware level, as well as upwards to the prediction and planning modules.

KEYWORDS

autonomous driving, resilience, fault and intrusion tolerance, Apollo, SVL simulator

1. Introduction

Over the years, autonomously driving vehicles (ADVs) have been progressively equipped with increasingly elaborate features to enhance driving experience and autonomy, ranging from high-resolution sensors to deep neural networks. Today, this increasing sophistication forms the backbone of indispensable computer vision algorithms, enabling precise obstacle perception, optimized trajectory planners, and smooth control algorithms, and has effectively been successful to asymptotically poke the level of driving automation to a higher standard. On the contrary, increasing complexity goes hand in hand with increasing vulnerability in any cyber-physical system (CPS). New pathways for malicious intrusions are opened up and the appearance of new faults becomes more probable, consequently resulting in dangerous and sometimes fatal outcomes.

Unfortunately, over the last 20 years, there has been no shortage of bad outcomes involving ADVs, many of which have been caused by unintended accelerations (UA) which killed 89 people (cbs, 2010). Erroneous behaviors such as UAs that are related specifically to the components that enable the autonomy of ADVs and independent from the human driver can potentially have two origins, either accidental due to an internal fault and/or absent fail-safe mechanism or provoked due to the presence of a malicious attacker (Lima et al., 2016).

Only by convincing the human driver of its trustworthiness can automation take over. In this regard, resilience of ADVs has to play a crucial role in triggering an effective adoption of ADVs by the general public at scale. In other words, because we expect that faults at any level will occur inevitably, infusing ADVs with mechanisms that enable tolerating those faults is of utmost importance. In the presence of faults, the notion of a responsibility gap arises naturally. This gap is characterized by situations in which it is unclear who can justifiably be held responsible for an unintended catastrophic outcome. The width of this gray zone is even more amplified by the over-reliance of modern-day ADVs' modules on artificial neural networks (NN). Not only are the safety and reliability of these modules rarely studied in cooperation with the whole ADV software stack (Peng et al., 2020) but they also inherited the connectionist bug of non-explainability.

In addition to their black-box nature, NNs are subject to usual faults which can reside in software or due to hardware issues (Torres-Huitzil and Girau, 2017). In the former, NNs can be reprogrammed (Elsayed et al., 2018), evaded (Eykholt et al., 2018), and data-poisoned (Aghakhani et al., 2021) by malicious intruders during the inference and/or the training phase, not to forget that NNs are subject to faults originating at the hardware level. Either transitory or permanent faults, such as stuck-at or bit-flip types, can alter the parameter space of the NN or lead to an erroneous computation of the hidden layer activation function (Arad and El-Amawy, 1997).

Similarly, sensors are not spared from attacks. Bad actors can modify the lane-keeping system by installing dirty road patches and ultimately causing the ADV to drift away from its lane (Sato et al., 2021). Jamming the cameras' modules (Panoff et al., 2021) or LIDAR spoofing attacks (Zhou et al., 2021) to inject false obstacle depth lead to false sensor data and hence causes the ADV data processing chain to compute erroneous control commands. In these cases, the health of the sensors is not compromised hence remain undetected by traditional fault detection schemes.

Common ADV software stacks, like Apollo Baidu (2017), are typically composed of a chain of interlocked modules that process information starting from the perception module down to the control algorithms, where the planned trajectory is transformed into ECU instructions. Because of this downstream interdependence and where the computation and safe execution of control commands are all causally interlinked, failure of an intermediary module can propagate through this information processing chain and lead to unexpected behaviors.

Efforts have been made to overcome the existence of single point of failures where, for example, perception information is rendered redundant by gathering raw data from independent modalities (RGB cameras, LIDAR, and RADAR) and fused to dilute the presence of a possible faulty device (Darms et al., 2008). However, redundancy implies that additional computational cost and certain modules that comprise GPU-resource greedy NNs cannot cheaply be replicated. Geng and Liu (2020) have focused their efforts on designing a model adaptive control algorithm for robust path tracking control and equipped the sensor fusion module with fault detection capabilities to enhance the overall fault tolerance of the ADV. Validating ADV software in a real physical environment is costly and does not scale sufficiently to cover all possible driving scenarios. Moreover, deploying ADV software directly on-road can be dangerous. Hence, interfacing physics simulators, such as SVL (Rong et al., 2020), with ADV software stacks is of fundamental importance to guarantee quality assurance in the automotive sector, as required by the evolving standard ISO 21448: Safety of the Intended functionality (Iso, 2019). Relevant to the study presented herein, Ebadi et al. (2021) have stress tested the autopilot enabled by the Apollo ADV software stack inside the SVL simulation environment by generating a set of edge cases where the Perception module was unable to detect pedestrians. Similarly, Seymour et al. (2021) created 576 test cases in the SVL simulator to assess the safety of the Apollo ADV software stack and observed that the perception modules failed to detect pedestrians in 10 % of the total number of scenarios tested.

1.1. Related work

Darms et al. (2008) studied fault tolerance when fusing different sensor modalities to mask eventual faulty sensor outputs. Geng and Liu (2020) designed a model adaptive control algorithm for robust path tracking control and equipped the sensor fusion module with fault detection capabilities to enhance the overall fault tolerance of the ADV. Ebadi et al. (2021) tested the Apollo ADV software stack in conjunction with SVL and generated a set of driving scenarios where the Perception module was unable to detect pedestrians. Seymour et al. (2021) created test cases in the SVL simulator where the Perception modules failed to detect pedestrian. Abad et al. (2016) studied the set of sufficient conditions under which recovery of software-faulty modules in cyber-physical systems can be deemed safes. Abdi et al. (2018) leverage the fact that due to the inertia of certain cyber-physical systems, an intruder cannot destabilize the physical system hence they were able to implement a safe and fast system-wide restart. Chu and Wah (1990) applied redundancy in trained NNs on individual neurons. Khunasaraphan et al. (1994) developed a NN weight-shifting technique which after fault detection restores weights and subsequently recovers the entire NN in a short amount of time.

This study documents our work in designing fault and intrusion tolerant (FIT) mechanisms applied to ADVs where we demonstrate the feasibility of applying those mechanisms into the sub-components of the Apollo ADV software stack and testing them in different driving scenarios generated by the SVL simulator. In particular, we give a qualitative description of a novel recovery scheme which enables the ADV, in the presence of a detected fault at the sensor level, to maintain a stable trajectory by leveraging the availability of predicted future sensor values which upon verification are fed back to the Prediction module while the Perception module is rebooting.

The recovery scheme proposed herein positions itself in the class of shallow recovery, which entails methods that repair compromised sub-component of a CPS with minimal or no operation on the system states. For example, Abad et al. (2016) developed a technique that restarts a failed component and substitutes it with a healthy one, whereas Shin et al. (2018) propose to leverage redundancy to fuse the output of multiple replicas where upon attack detection isolate and restart the origin of the faulty contribution. Much more reminiscent of the recovery method described in this work has been evaluated by Kong et al. (2018) where a checkpointing scheme is put in place to store historical state data and its correctness is verified before restoring it for the recovery of compromised sensors.

1.2. Organization of this article

- We lay out the system architecture of a popular ADV software stack and give a description of how to embed it in a simulated physics environment using SVL.
- We highlight some vulnerabilities of Apollo and describe the threat model.
- We describe the FIT mechanisms that we implemented in three Apollo modules.
- We showcase and validate two of these mechanisms by interfacing the Apollo ADV software stack with the SVL simulation environment.

2. Apollo ADV software stack architecture

2.1. Description

Apollo is an industrial intelligent-ADV open-source software stack maintained by Baidu (2017) and is currently deployed in autonomous taxi services in different cities around the world. The code architecture follows a standard logic found in other ADV software stacks, in which various software and hardware components are integrated together following a complex logic. The architecture (see Figure 1) of Apollo can be simplified as a hierarchical processing chain of information starting from the perception module and trickling down to the CANbus as follows:

- **Perception** contains the different sensor drivers and the trained machine-learning tools for sensor fusion and obstacle detection.
- **Prediction** receives the recognized obstacles from the Perception module and predicts, *via* a collection of trained ML sub-modules, their probable trajectory with a prediction horizon of 8 s.
- **Planning**, upon gathering localization information from the localization system, routing information and the predicted obstacle trajectories from the Prediction module, computes the safest and shortest trajectory to be taken by the ego car.
- **Control** translates *via* path tracking control algorithms the received spatio-temporal trajectory coordinates of the Planning module into steering, braking, and throttling commands.
- **CAN** (or similar field busses or in-car networks) communicates the control commands from the Control module to the relevant ECUs and the actuators they control.

The Perception module is scheduled recurringly with a fixed period. Prediction, Planning, and Control are event-triggered as new data frames come in.

2.2. Simulator

The SVL simulator is a multi-robot simulator for ADVs maintained by LG Electronics America R&D Lab (Rong et al., 2020). SVL is able to generate a whole range of different maps and obstacles, such as road vehicles and pedestrians. It allows customizing driving scenarios and publishing them to an ADV software stack, such as Apollo, *via* the CyberRT bridge connection.

2.3. Implementation

Apollo leverages containers to isolate and protect its components. Containers offer a restricted execution environment with container-to-container communication possibilities and are hosted in Apollo on top of a Linux-based operating system. We assume for a deployed system that the containers of critical components (such as control) will be hosted directly on top of a real-time operating system (RTOS) that is capable of offering the required isolation. Of course, the RTOS in such architectures forms a single point of failure, which must be addressed in future [e.g., as demonstrated in the Midir architecture (Gouveia et al., 2022)].

For the above-mentioned reason, we implemented the resilience mechanisms discussed in this study at container-level, replicating, and restarting containers to tolerate faults and to rejuvenate replicas, but also to isolate voters and the trusted storage system. However, this leaves, when it comes to communicating agreed-upon trajectories, the underlying driver infrastructure as a single point of failure, which we address next.

2.4. Vulnerabilities

A first-order analysis of the ADV architecture (see Figure 1) reveals that every module is a single point of failure. That is, a fault, triggering an error in any one module along the information processing chain can either produce an erroneous computation of subsequent modules or impede the latter from receiving timely information, which disrupts the generation of correct and timely control commands to the ECU. We address this lack of redundancy in the following sections and demonstrate how FIT designs can mitigate the risks of component failures.

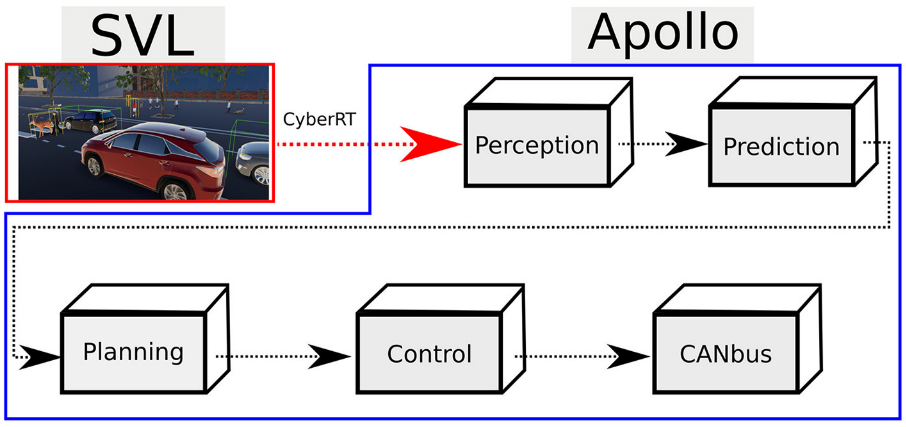

FIGURE 1
Apollo software architecture and bridge connection with SVL.

3. FIT and resilience mechanisms for autonomous vehicles

3.1. Threat model

In this work, we are primarily concerned about mitigating negative effects from accidental faults and cyberattacks on autonomous vehicle driving functions. As such, our fault model concentrates on the interior components. We consider sensor-level attacks as well as model extraction and adversarial machine-learning attacks.

We follow the fault model introduced by Sousa et al. (2009). That is, during any time window of length T_A, most f components may fail for accidental or malicious reasons. T_A thereby considers the time adversaries require to overcome the fault threshold f of simultaneous faults that the system should tolerate. In case of accidental faulty components, T_A includes as well the time needed to adjust to these accidental faults for exploiting them in their attack.

As Sousa identified, rejuvenating all n replicas faster than T_A (i.e., with a rejuvenation time $\lceil \frac{n}{k} \rceil \cdot T_R < T_A$, where k replicas are rejuvenated simultaneously) maintains the healthy majority over extended periods of time.

As usual, we assume replicated components to be properly isolated and sufficiently diverse (e.g., through obfuscation) so that they can be assumed to fail independently with high coverage. Indeed, we cannot avoid equivocation between replicated components that read from the same input buffer, we can nevertheless mitigate equivocation by letting them copy out the original input and comparing this input together with the respective processed outputs through a trusted voting mechanism Alternatively, a trusted operating system (which we assume is in place and establishes containers as fault containment domains) copies the proposed value to all replicas, avoiding equivocation in the process.

Our design is a hybrid architecture. That is, we differentiate the fault model of our trusted components: state storage and voters (see next section). While normal components can fail in an arbitrary

TABLE 1 Resource consumption per Apollo module in terms of a RAM and video RAM.

Module	RAM [GiB]	VRAM [GiB]
Perception	<0.1	6.71
Prediction	<0.1	3.21
Planning	0.40	None
Control	0.07	None

Byzantine manner, state storage and voters must not fail, which we justify through their simplicity in terms of the number of lines of code (<100) and are implemented as trivial state machines. In particular, for state storage, we assume that techniques such as ECC and scrubbing are in place to correct the effects of accidental faults in the stored data.

3.2. Control module replication

For deciding which module could benefit from a state machine replication scheme, we monitored the resource consumption during a test drive of every Apollo module in terms of memory (RAM and video VRAM) with the use of the system-monitor process viewers HTOP and NVTOP. We report the maximal amount of resource consumption in Table 1. Since the Control module is the lightest in terms of resource consumption, it lends itself optimally well for N-fold state machine replication. Control receives trajectories from planning, validates them, and forwards control commands to the ECU, by sending commands over the in-car networks. Since ECUs are, in general, not aware of the replicated nature of control, a trusted voting mechanism suggests itself to consolidate the control outputs of the individual replicas to a single command stream, masking up to f faults behind a $f+1$ majority of correct outputs (see Figure 2). For a given vote, we only consider control commands that are timestamped inside the same temporal

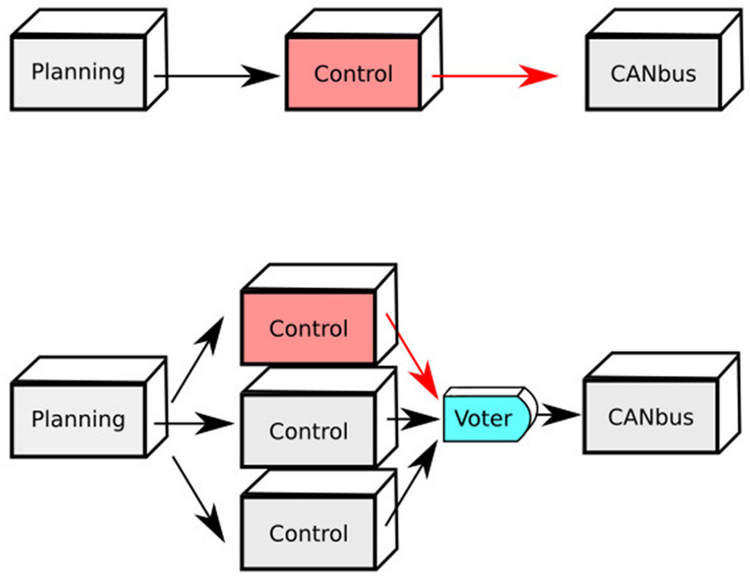

FIGURE 2
Triple control module replication with voter module.

window of size 5 ms. If no majority is reached, the time window is skipped.

3.3. Perception module rejuvenation

In this FIT scheme, we consider sensor-level attacks that are detected by Apollo leading to a shutdown of the Perception module. The complex comprised of the Prediction, and the Perception forms the most GPU-resource intensive component (see Table 1) as it heavily relies on NN-powered functions during the feed-forward stage. Therefore, a module replication would be too costly. A different route to implement an intrusion tolerant mechanism is to repair the Perception module and reboot it fast enough to remove adversaries and ensure that the Planning module is supplied with timely obstacle predictions to compute a safe ego-car trajectory. Indeed, the large NN matrices that need to be loaded into GPU memory, leading to boot-up times of up to 4 s in Apollo, make the prospects for a fast and safe Perception module reboot impossible. While the failed Perception module is restarting, it is effectively non-operational and therefore unable to supply the Prediction module with fresh processed sensor data. The Prediction module estimates the future trajectory of every detected obstacle, hence a one-time instance of the batch of future spatio-temporal trajectory coordinates can be used to temporarily substitute missing Perception output frames. This can be achieved by designing a State Storage module (see Figure 3) acting as a buffer to record and save the predicted obstacle trajectories and make them available to the Prediction module whenever the Perception module is non-responsive.

The temporal logic of this mechanism is illustrated in Figure 4. We denote h_i as the data frame produced by the Perception module at time i; $\mathbf{h}_j = h_{j,0}, ..., h_{j,i}, ..., h_{j,N}$ is the batch of predicted obstacle trajectories produced at time instance j, referred to as states. One batch is composed of N samples where one sample $h_{j,i}$ is the spatio-temporal trajectory coordinate corresponding to the future instance at time $t_i = i \cdot \Delta t$ and $h_{j,0} = h_j$. We save $\mathbf{h}_j$ in the State Storage module. We denote by t_F the onset time of the Perception module reboot. At time $t_F + t_i$ we restore $h_{F,i} = h_j$ and continue until the Perception module reboots and starts supplying the Prediction module with fresh data.

Moreover, not all states can be deemed safe for restoration, because of the two following scenarios:

1. A possible misbehavior in the Perception module leading to wrong output either due to a sensor sampling error, jamming and spoofing attack, or a processing error at the level of the neural net modules.
2. Saved states contain future-detected obstacle trajectories that have been predicted based on past events (before t_F). It would be dangerous to restore states that have been stored when the ego-car, at time t_F, transitions from a relatively predictable situation (highway with little traffic) to a chaotic unpredictable situation (intersection crossing with pedestrians).

The first concern can be mitigated by recognizing that a missing or spoofed time frame should be, to some degree, detectable by a lack of continuity in the recorded data stream. This can be mathematically formulated by means of a continuity check based on the Lipschitz' λ-continuity definition applied to a sequence of past recorded Perception module outputs. That is, if there exists a function f, which admits a finite $\lambda \in R$ such that $\|f(h_{j,0}) - f(h_{j+1,0})\| \leq \lambda\|h_{j,0} - h_{j+1,0}\| \; \forall j = 0, ..., F$ then all $h_{j,0}$ are deemed continuous.

The second concern can be avoided by adding a constraint to state restoration by estimating the entropy or the temperature of the predicted scene and setting a threshold under which a state is deemed safe to be restored. Intuitively, these statistical measures are directly related to the notion of predictability. For instance, a

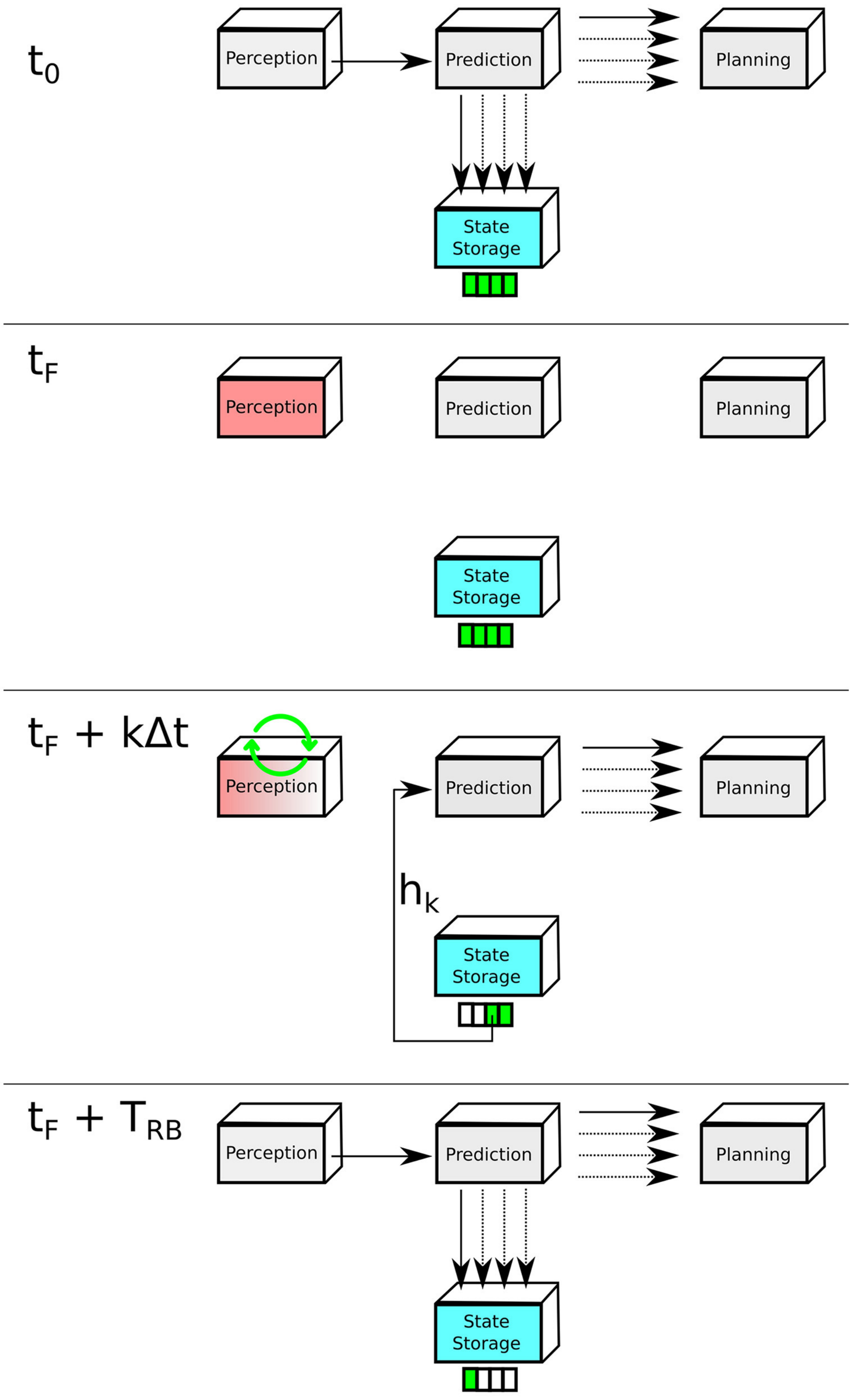

FIGURE 3
Perception module recovery. Parameters t_F and T_{RB} denote the onset of the reboot and reboot time of the prediction module, respectively. h_k is the stored state corresponding to the future time instance k and Δt is the sampling period of the prediction module output.

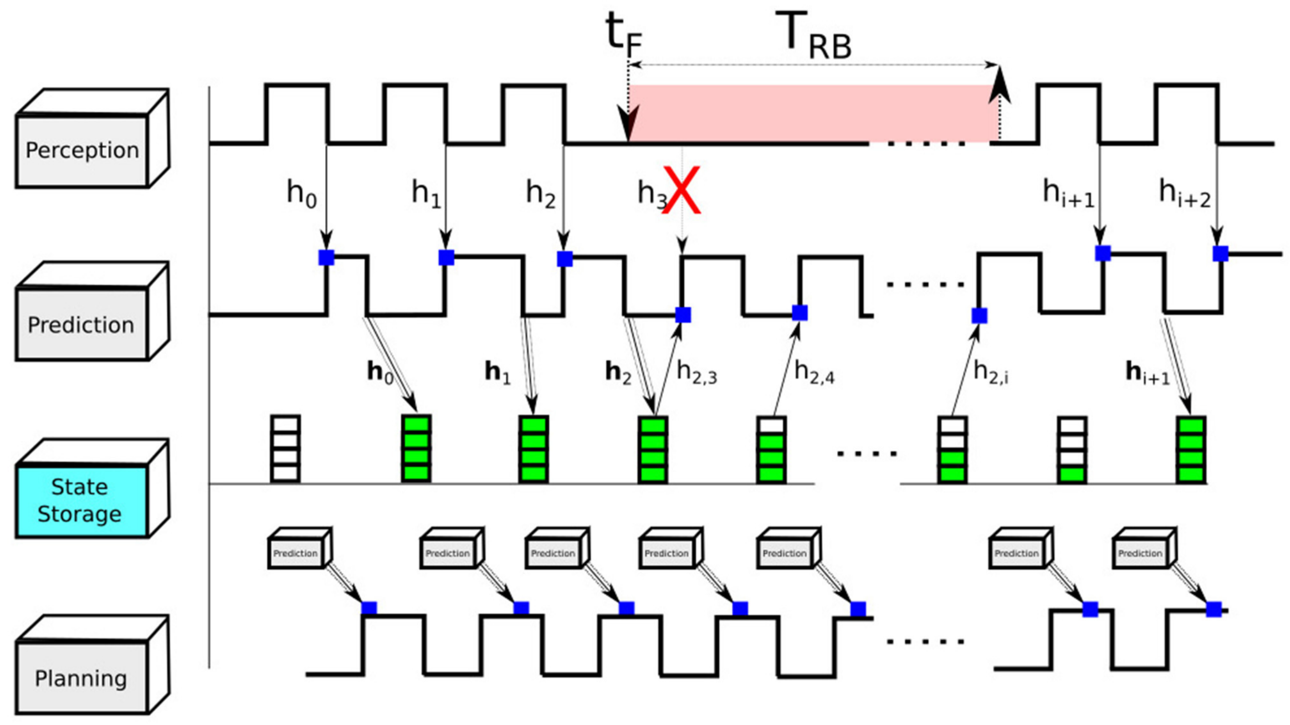

FIGURE 4
Temporal logic schematic of Perception module rejuvenation.

proximal fast-moving truck is considered to be a greater concern than a distant slow-moving pedestrian. Hence, we can intuitively sketch the general trend for such a measure. Predictability $H(t)$ should:

- decrease as a function of the prediction time,
- decrease with the ego-car speed because in a fast varying environment, new obstacles enter and/or leave the detection range,
- decrease with the relative speed of the ego-car with respect to the detected obstacles as the latter are more likely to pose a collision threat,
- increases with the relative distance between the ego-car and the detected obstacles, as the likelihood of a collision, is reduced.

Finally, the design of a fault recovery scheme that repairs a compromised Perception module and allows the ADV to continue to operate seemingly in the presence of an intrusion is bounded by the inequalities $(T_{RB} \leq N\Delta t)$ & $\left(T_{RB} \leq \max\left(\arg_t\{H(t) \leq H_{th}\}\right)\right)$, where H_{th} is the threshold value which controls how many potentially unsafe states are tolerated for restoration. That is, the prediction horizon needs to be long enough to give the Perception module enough margin to recover but at the same time the latter has to reboot faster than the last and sufficiently predict internal state to avoid feeding back samples to the Prediction module with low confidence scores.

3.4. Toward device-driver replication

In addition to protecting the high-level components of the software stack (perception, prediction, and control), we must also address faults at lower-level software components, which interact with ECUs (e.g., by sending messages over the CAN bus) or which otherwise interact with hardware.

Operating-system code interacts with devices through memory-mapped registers (MMIO) and interrupts, triggered by the device on a CPU. Writes to certain device registers may have side effects, such as the sending of a packet over the network. Therefore, to interact with devices in a fault-tolerant manner, we ultimately need to systematically forward interrupts to all replicas for interrupt handling and vote on all critical register writes.

We approach consensual OS-to-device interaction by replicating the SPI driver, leveraging Linux's user-level driver support, which in our setup communicates with the CAN-bus controllers on the PICAN Raspberry-PI hat. That is, before commands are sent to the CAN hat, consensus must be reached and all direct MMIO writes be redirected through voters.

Of course, this is only a partial solution, since some devices may have side effects on reads and delicate timing requirements in their interface, which need a more elaborate investigation of the voter interface, which interposes device access. We will investigate such interfaces as part of our future work.

In a pure simulation environment without the necessary hardware, it is impossible to validate the behavior of a CANbus replication. We can nevertheless put forward a few ways on how this could be practically implemented. Indeed a

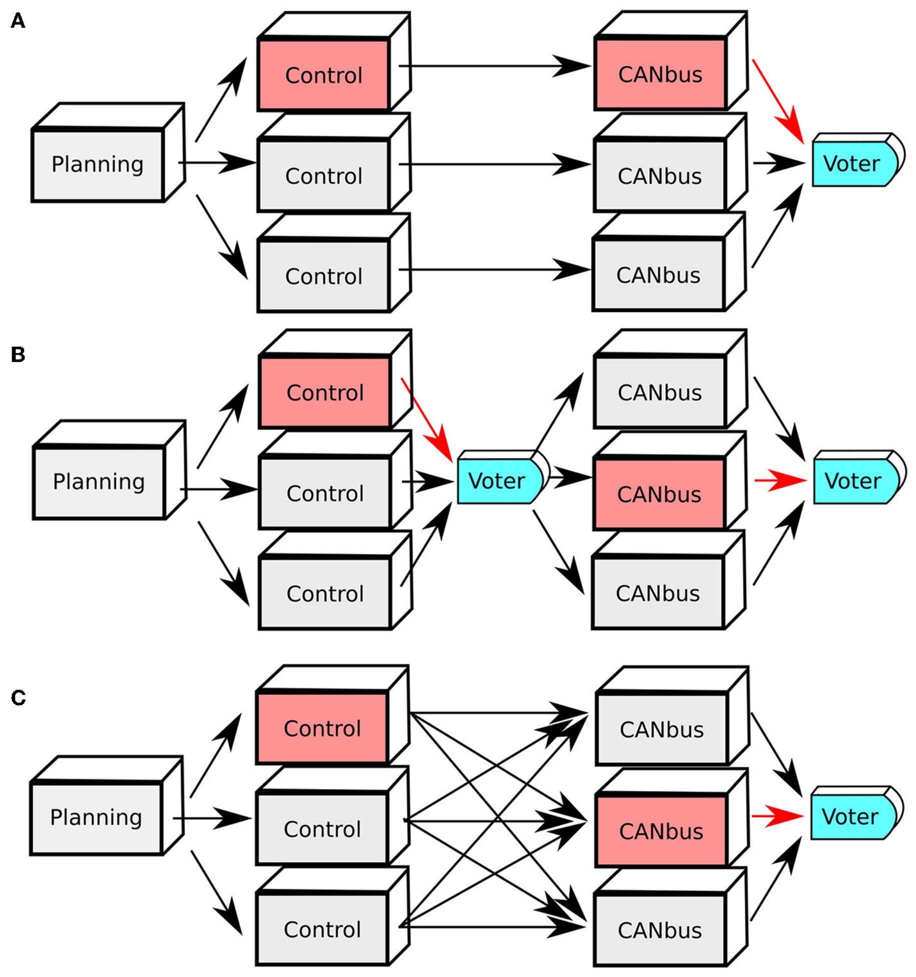

FIGURE 5
(A–C) Three different CANbus replication schemes in conjunction with a Control module replication.

TABLE 2 CANbus replication FIT properties.

Scheme	N_{Rounds}	Scalability	Fault tolerance
A	2	$\mathcal{O}(n)$	CANbus *i* state depends on control *i* state
B	3	$\mathcal{O}(n)$	All nodes can fail independently
C	3	$\mathcal{O}(n^2)$	All nodes can fail independently

CANbus replication could feasibly be integrated inside the Apollo software stack architecture in conjunction with a Control module replication. Three obvious schemes are depicted on Figure 5, each varying by number of communication rounds N_{Rounds}, scalability of the number of exchanged messages, and their capacity to tolerate faults summarized in Table 2.

4. Evaluations

4.1. Setup

We evaluate the FIT schemes by running simulations on a desktop PC with the following specification:

- Ubuntu 20.04
- AMD Ryzen 7 3700X 8-Core Processor @ 3.6GHz X 16
- 62,7 GiB of RAM
- NVIDIA GeForce RTX3090
- Baidu Apollo 6.0
- SVL simulator 2021.1 interfaced with Apollo *via* CyberRT bridge
- Cuda 11.4, Nvidia Docker 20.10.8

4.2. Control replication

We duplicated the original Control module resulting in three instances and monitored the three main control commands; steering angle, brake, and throttling intensities. We simulated a worst-case instance of one faulty replica by adding white noise to its output stretching over the whole range of values that each of the control commands can take. Each of the replicas published the computed control commands to the Voter Control module *via* the CyberRT channel which were subsequently submitted to a vote if they belonged to the same temporal window (5 ms). Whenever a majority of 2 was reached, the result of the

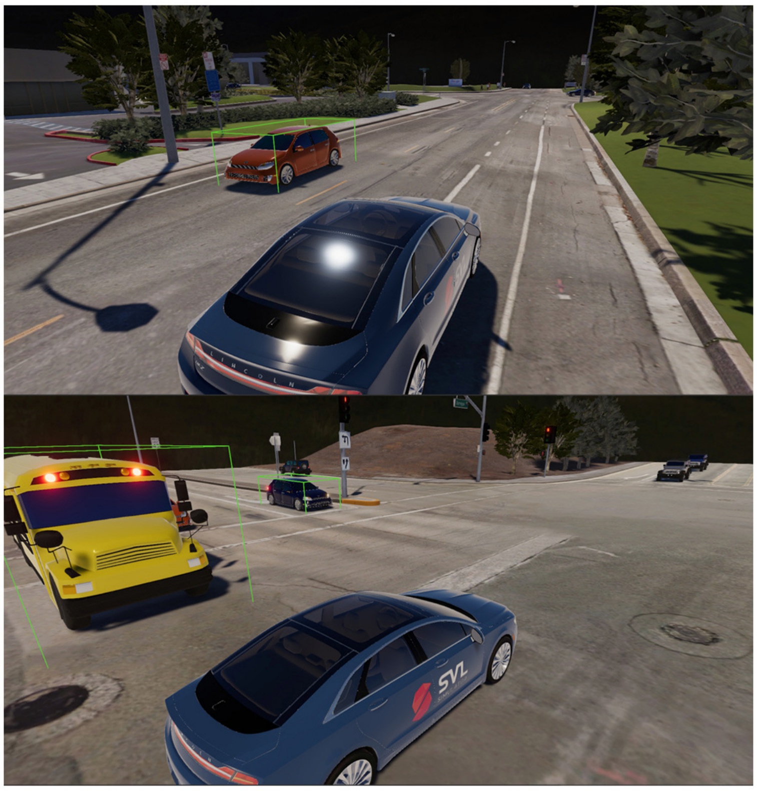

FIGURE 6
Top: High predictability, low risk for accident scenario. **Bottom**: Low predictability high risk for accident scenario.

voting was published back to the SVL simulator for actuation, otherwise the vote was skipped. We simulated in SVL a 2-min driving scenario involving crossing 10 intersections with high traffic and pedestrians. We report a stable trajectory execution despite the presence of one faulty Control module replica, with an overhead of 70 MiB RAM per additional replica, 16 Mib RAM for the added Voter module, and an added 2 ms latency. Less than 0.05 % of votes have been skipped due to a delayed replica response. We measured the relative error between the output of a healthy replica and the output of the voter. We observed an error rate of less than 0.1% for the three monitored control commands.

4.3. Perception module rejuvenation

We deployed the Perception module rejuvenation scheme in multiple simulation runs, in which we triggered a reboot of the Perception module. The prediction horizon for every obstacle was 8 s, the reboot time was consistently 3.7± 0.1 s. In a high predictability scenario (see the top part of Figure 6), during the reboot phase, the State Storage module could reliably supply the Prediction module with enough verified stored internal states before depletion (8 s) and we observed no disruption in the ego-car planning behavior. We evaluated the internal state verification procedure laid out in Section 3.4 by creating a driving scenario in SVL where the ego car Perception module is rebooted 1 s before arriving at a busy intersection crossing (see the bottom part of Figure 6). Through its state verification feature, the State Storage was able to predict the high entropy (low predictability) of the situations. Through fine-tuning the H_{th} value to a relatively conservative level, the ego car effectively avoided replaying invalid internal states to the Prediction module hence bringing the car to a complete stop before waiting for a fresh instance of the Perception module to reboot and continue its course.

5. Discussion

In this study, we reported on the results and findings of a case study, retrofitting an autonomous driving software stack with fault and intrusion tolerance mechanisms. We laid out a powerful methodology to design, test, and validate those mechanisms in interaction with the complex logic of the Apollo ADV software stack, which we embedded inside the SVL simulator. We were able to not only study the feasibility of the developed schemes but could also showcase their efficacy by measuring relevant metrics. We hereby stretch the importance of validation through simulation which is fundamental to prove quality assurance before deploying software in conventional on-road testing.

In Apollo, we already found a rudimentary preparation for retrofitting resilience, by encapsulating large subsystems in containers. This way, some of the resilience mechanisms could be provided in an application agnostic manner, at container level, whereas others (in particular, for perception and planning) required considering application-specific knowledge. Ideal designs should allow for flexible, fine-grain decomposition of components, with clear interfaces and the possibility to isolate components individually, if necessary, and to combine them into a single fault-containment domain, if not.

5.1. Limitations and future work

Our Control module replication scheme inherits the same limitation as the generic state machine replication technique. First, creating n exact copies of the original Apollo Control module, calls for enhancing diversity and the implied costs to generate this diversity, e.g., initially through n-version programming and over time through obfuscation. Second, for a total of f faulty replicas, maintaining FIT properties comes at a $2f+1$-replication cost which is exacerbated by our reliance on containers for isolation combined with our dependency on the OS.

Regarding the Perception module recovery scheme, we did not perform a thorough evaluation. Nevertheless, we were able to retrofit it into Apollo and demonstrate it qualitatively inside a simplistic simulation environment. Given the current design, we can only trigger a full perception recovery in relatively predictable driving scenarios such as straight lanes with low traffic and high visibility. Future work would include investigating the effect of different threshold predictability values H_{th} on the recovery of the ADV in diverse unpredictable and more complex driving scenarios.

Additional resilience mechanisms could be envisioned for the other Apollo modules such as the Prediction and Planning modules. As for the latter, we could easily and economically apply the same state machine replication technique as it is relatively low overhead. Being succeeded by an already replicated Control module, planning outputs should be consolidated by the same voting scheme that we discussed for the Control-Canbus complex where the same trade-offs apply among scalability, communication rounds, and capacity to tolerate faults.

Due to its high GPU resource requirement, the Prediction module could benefit from a similar recovery scheme as the Perception module. Less obvious, however, the Planning module requires the full batch of Prediction outputs to compute a safe trajectory, i.e., buying time by buffering prediction outputs and supplying them one-by-one to the the Planning module while the Prediction module is restarting is bound to fail.

A better scheme would involve triggering a fail-safe mechanism that upon detecting a failed Prediction module would trigger a separate and much simpler Planning module. The purpose of the latter would be to compute a pull-over trajectory steering the car into a proximal safe spot where a fresh Prediction module can be re-instantiated.

Going further, through the advent of cooperative autonomous driving, the pull-over scenario could be triggered by a nearby trusted car or a group of cars acting as external replicas. By relying on a distributed communication protocol, these replicas would compute and vote on a safe pull-over trajectory in a consensual fashion and communicate it to the failed car for execution. Simulating this scenario would entail running multiple Apollo instances in parallel, where an additional custom Apollo network module would ensure the vehicle-to-vehicle communication.

In addition, we plan to investigate more application-specific solutions to secure autonomous driving against accidental faults and cyberattacks to eliminate further single points of failures (like the RTOS and complex drivers). Finally, another direction of future work includes reducing the dependency on learning-based components and in turn mitigate the attack vectors they are exposed to.

Data availability statement

The original contributions presented in the study are included in the article/supplementary material, further inquiries can be directed to the corresponding author.

Author contributions

FL, RG, and MV contributed to conception, design of the study, and wrote sections of the manuscript. FL wrote the first draft of the manuscript. All authors contributed to manuscript revision, read, and approved the submitted version.

Funding

This research was part of a partnership project funded by Huawei Technologies Co., Ltd.

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's note

References

Abad, F. A. T., Mancuso, R., Bak, S., Dantsker, O., and Caccamo, M. (2016). "Reset-based recovery for real-time cyber-physical systems with temporal safety constraints," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)* (IEEE) 1–8. doi: 10.1109/ETFA.2016.7733561

Abdi, F., Chen, C.-Y., Hasan, M., Liu, S., Mohan, S., and Caccamo, M. (2018). "Guaranteed physical security with restart-based design for cyber-physical systems," in *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)* (IEEE) 10–21. doi: 10.1109/ICCPS.2018.00010

Aghakhani, H., Meng, D., Wang, Y.-X., Kruegel, C., and Vigna, G. (2021). "Bullseye polytope: A scalable clean-label poisoning attack with improved transferability," in *2021 IEEE European Symposium on Security and Privacy (EuroS and P)* (IEEE) 159–178. doi: 10.1109/EuroSP51992.2021.00021

Arad, B. S., and El-Amawy, A. (1997). On fault tolerant training of feedforward neural networks. *Neur. Netw.* 10, 539–553. doi: 10.1016/S0893-6080(96)00089-5

Baidu (2017). *Apollo: Open source autonomous driving.*

CBS. (2010). *Toyota "unintended acceleration" has killed.* 89.

Chu, L.-C., and Wah, B. W. (1990). "Fault tolerant neural networks with hybrid redundancy," in *1990 IJCNN International Joint Conference on Neural Networks* (IEEE) 639–649. doi: 10.1109/IJCNN.1990.137773

Darms, M., Rybski, P., and Urmson, C. (2008). "Classification and tracking of dynamic objects with multiple sensors for autonomous driving in urban environments," in *2008 IEEE Intelligent Vehicles Symposium* (IEEE) 1197–1202. doi: 10.1109/IVS.2008.4621259

Ebadi, H., Moghadam, M. H., Borg, M., Gay, G., Fontes, A., and Socha, K. (2021). "Efficient and effective generation of test cases for pedestrian detection-search-based software testing of baidu apollo in svl," in *2021 IEEE International Conference on Artificial Intelligence Testing (AITest)* (IEEE) 103–110. doi: 10.1109/AITEST52744.2021.00030

Elsayed, G. F., Goodfellow, I., and Sohl-Dickstein, J. (2018). Adversarial reprogramming of neural networks. arXiv preprint arXiv:1806.11146.

Eykholt, K., Evtimov, I., Fernandes, E., Li, B., Rahmati, A., Xiao, C., et al. (2018). "Robust physical-world attacks on deep learning visual classification," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* 1625–1634. doi: 10.1109/CVPR.2018.00175

Geng, K., and Liu, S. (2020). Robust path tracking control for autonomous vehicle based on a novel fault tolerant adaptive model predictive control algorithm. *Appl. Sci.* 10, 6249. doi: 10.3390/app10186249

Gouveia, I. P., Völp, M., and Esteves-Verissimo, P. (2022). Behind the last line of defense: Surviving soc faults and intrusions. *Comput. Secur.* 123, 102920. doi: 10.1016/j.cose.2022.102920

Iso, I. (2019). "Pas 21448-road vehicles-safety of the intended functionality," in *International Organization for Standardization.*

Khunasaraphan, C., Tanprasert, T., and Lursinsap, C. (1994). "Recovering faulty self-organizing neural networks: By weight shifting technique," in *Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN'94)* (IEEE) 1513–1518.

Kong, F., Xu, M., Weimer, J., Sokolsky, O., and Lee, I. (2018). "Cyber-physical system checkpointing and recovery," in *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)* (IEEE) 22–31. doi: 10.1109/ICCPS.2018.00011

Lima, A., Rocha, F., Völp, M., and Esteves-Veríssimo, P. (2016). "Towards safe and secure autonomous and cooperative vehicle ecosystems," in *Proceedings of the 2nd ACM Workshop on Cyber-Physical Systems Security and Privacy* 59–70. doi: 10.1145/2994487.2994489

Panoff, M., Dutta, R. G., Hu, Y., Yang, K., and Jin, Y. (2021). On sensor security in the era of iot and cps. *SN Comput. Sci.* 2, 1–14. doi: 10.1007/s42979-020-00423-5

Peng, Z., Yang, J., Chen, T.-H. P., and Ma, L. (2020). "A first look at the integration of machine learning models in complex autonomous driving systems," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* 1240–1250. doi: 10.1145/3368089.3417063

Rong, G., Shin, B. H., Tabatabaee, H., Lu, Q., Lemke, S., Možeiko, M., et al. (2020). SVL simulator: a high fidelity simulator for autonomous driving. arXiv e-prints, arXiv:2005.03778. doi: 10.1109/ITSC45102.2020.9294422

Sato, T., Shen, J., Wang, N., Jia, Y., Lin, X., and Chen, Q. A. (2021). "Dirty road can attack: Security of deep learning based automated lane centering under $*Physical − World*$ attack," in *30th USENIX Security Symposium (USENIX Security 21)* 3309–3326. doi: 10.14722/autosec.2021.23026

Seymour, J., Ho, D.-T.-C., and Luu, Q.-H. (2021). "An empirical testing of autonomous vehicle simulator system for urban driving," in *2021 IEEE International Conference on Artificial Intelligence Testing (AITest)* (IEEE), 111–117. doi: 10.1109/AITEST52744.2021.00031

Shin, J., Baek, Y., Lee, J., and Lee, S. (2018). Cyber-physical attack detection and recovery based on rnn in automotive brake systems. *Appl. Sci.* 9, 82. doi: 10.3390/app9010082

Sousa, P., Bessani, A. N., Correia, M., Neves, N. F., and Verissimo, P. (2009). Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Trans. Parallel Distrib. Syst.* 21, 452–465. doi: 10.1109/TPDS.2009.83

Torres-Huitzil, C., and Girau, B. (2017). Fault and error tolerance in neural networks: A review. *IEEE Access* 5, 17322–17341. doi: 10.1109/ACCESS.2017.2742698

Zhou, C., Yan, Q., Shi, Y., and Sun, L. (2021). Doublestar: Long-range attack towards depth estimation based obstacle avoidance in autonomous systems. arXiv preprint arXiv:2110.03154.