BRAUM: Analyzing and Protecting Autonomous Machine Software Stack

Yiming Gan
University of Rochester
ygan10@ur.rochester.edu

Paul Whatmough
Arm Research
paul.whatmough@arm.com

Jingwen Leng
Shanghai Jiaotong University
leng-jw@cs.sjtu.edu.cn

Bo Yu
PerceptIn
bo.yu@perceptin.io

Shaoshan Liu
PerceptIn
shaoshan.liu@perceptin.io

Yuhao Zhu
University of Rochester
yzhu@rochester.edu

***Abstract*—Autonomous machines, such as Autonomous Vehicles (AV), are vulnerable to a variety of different faults such as radiation-induced soft/transient errors, adversarial attacks, and software bugs, which all jeopardize the reliability of autonomous machines. *How* vulnerable the AV software stack is to different error sources, however, remains an open question.**

This paper performs comprehensively fault injections to study how the AV software stack behaves under different error sources. We show that algorithms in an AV software stack inherently possess different forms of masking mechanisms. Based on the characteristic of the inherent fault tolerance mechanisms, we formalize the notion of Fault Tolerance Level (FTL), which quantifies how faults in an algorithm can be masked and/or attenuated without affecting the actuator commands, providing opportunities to relax fault protection.

Leveraging the FTL formulation, we propose a dynamic protection system, which, at the high level, spends the limited protection budget (e.g., spatial/temporal redundancy) on the most vulnerable parts of the AV software (i.e., with the lowest FTL). Using Autoware as a case study, we show that our system reduces the error rate of AV software stack by more than 90% with negligible performance overhead.

I. INTRODUCTION

Autonomous machines are not reliable. With the rapid growth of the autonomy inside drones, robotics and vehicles [1]–[3], the unreliability of hardware and software in autonomous machines have also resulted in many crashes of the systems. Among them, the reliability of autonomous vehicles, which are vehicles executing driving tasks autonomously, is crucial, as faults that happen in autonomous vehicle systems could lead to severe consequences [4], [5].

Despite numerous efforts in improving the safety of AV products [6]–[8], a myriad of sources threatening AV safety still exist. A single bit-flip of the transistors inside the hardware caused by thermal irregularities or cosmic rays [9]–[11] can result in a silent data corruption (SDC) of an arbitrary algorithm in the AV software stack. The SDC can propagate to the following software and influence the behavior of the vehicle. Carefully designed adversarial attacks on traffic signs and traffic lights are proved to be able to mislead the perception module of the AV [12]–[14] easily. Software bugs [15], [16] in the AV software stack can also be the source of unreliability.

Existing fault-tolerance techniques are expensive. Traditional protection mechanism such as modular redundancy [17]–[19], anomaly detection and recovery [20]–[23], and re-execution [24], [25] introduce spatial and/or temporal overhead, challenging the real-time nature of AV.

We propose BRAUM, a dynamic protection system rooted in the understanding of the inherent fault tolerance mechanisms of autonomous software. Using the popular Autoware AV software as a case-study [26], we observe that many algorithms in an AV software have inherently error-masking and/or error-attenuation capabilities through operations such as operator union and low-pass filtering. BRAUM systematically identifies these error-masking mechanisms and leverages these mechanisms to selectively provide error protection to AV software with minimal overhead.

To identify the error-masking capability of AV algorithms, we conduct a large-scale fault injection into the AV software stack and use program analysis techniques to trace how faults propagate and are masked by each algorithm in the AV software. Our fault injection framework synthesizes and injects faults that mimic different forms of potential faults an AV software could encounter, including transient errors, adversarial attacks, and software bugs. The fault injection framework is lightweight so as to minimize the impact on the behavior of the original software stack.

Our fault injection and analysis reveal many mechanisms for error masking and/or error attenuation possessed by different algorithms in the AV software stack. Building on top of these inherent mechanisms, we propose the notion of Fault Tolerance Level (FTL), which describes whether and how the output error of an individual algorithm will be masked before reaching the actuator. We show an iterative algorithm that calculates the FTL of an algorithm in an AV software stack.

Leveraging the FTL calculation, we propose a dynamic

Copyright and Reprint Permission: Abstracting is permitted with credit to the source. Libraries are permitted to photocopy beyond the limit of U.S. copyright law for private use of patrons those articles in this volume that carry a code at the bottom of the first page, provided the per-copy fee indicated in the code is paid through Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923. For reprint or republication permission, email to IEEE Copyrights Manager at pubs-permissions@ieee.org. All rights reserved. Copyright ©2022 by IEEE.

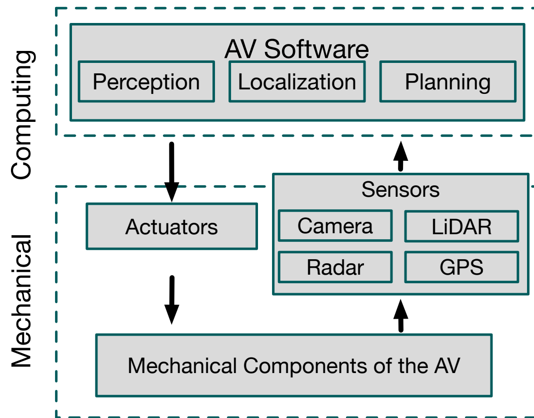


Fig. 1: An overview of AV system.

protection system that selectively elides and/or relaxes protection of certain algorithms in the software. Intuitively, if an algorithm has a "high FTL", e.g., its output errors can always be masked before reaching the actuator, there is no need to, for instance, re-execute that node to avoid soft errors.

In summary, we make the following contributions:

- We propose a lightweight fault injection framework to synthesize and inject representative faults that an AV could encounter.
- We analyze the fault injection results to identify inherent error masking/attenuation mechanisms in AV software.
- We propose the notion of FTL to intuitively describe how *invulnerable* an algorithm is, and provide an iterative algorithm to calculate the FTL of each algorithm in the software stack.
- We design a dynamic protection system that selectively elides/relaxes protection for high-FTL algorithms.
- We implement our protection system on Autoware, a widely-used AV software. Experimental results show that we reduce error propagation rates by 90.1% compared with a baseline without any protection and reduces the execution overhead by 47.2% compared to a baseline that provides always-on protection.

II. Background

We briefly introduce the design of AV software in Sec. II-A, the source of threats to the reliability of AV software in Sec. II-B and existing protection mechanism in Sec. II-C. We discuss common protection methods in Sec. II-C.

A. Autonomous Vehicle Software Stack

Fig. 1 shows a high-level overview of an AV system which consists of a computing system and a mechanical system. At each time frame, the computing system takes input from the sensors (e.g., cameras, LiDAR, GPS) to infer an actuation (e.g., throttle, brake, steering wheel angle) to the actuators and further control the mechanical components of the AV.

The computing system runs a complicated software system on the hardware. Typical AV software stacks such as Autoware [26] and Baidu Applo [27] perform three major tasks that enable the autonomous driving capability: perception, localization and control. The perception module detects and interprets the environments with the information provided by the cameras and LiDARs. The localization module locates the current position of the AV on the map with the input of GPS and IMUs. Planning digest both results from perception and localization module to plan for the trajectory and control the actuators. Usually each module contains multiple algorithms. Different algorithms are connected in a consumer-producer fashion, formulating a complicated computational graph.

B. Source of Threats to AV Software

AV software is not running in a safe and reliable environment. In reality, when the vehicle is driving autonomously on the road, different sources of threats exist. We briefly describe the three main error sources that this paper focuses on.

Soft errors. Soft error is a type of error that changes the states of a logic device (e.g., a SRAM cell). A soft error can be caused by cosmic rays or thermal impact. Multiple works [28]–[30] have shown that a bit-flip caused by the soft error can easily result in incorrect outputs of a program, i.e., Silent Data Corruption (SDC). SDC can take place in stage/algorithm in an AV software stack. An SDC occurring at one stage could propagate and eventually corrupt the output of the entire software stack, crashing the vehicle [31].

Adversarial attacks. Unlike soft error and software bugs that take place unintentionally, adversarial attacks are carefully crafted manipulations of the input of certain algorithms that cause an algorithm to misbehave. The most well-studied example is the adversarial attack on the input image to the deep learning-based perception module. Negligible perturbations to an input image can lead a perception DNN to mispredict, e.g., recognizing a stop sign as a green light [32]–[36]. Adversarial attacks have also been extended to other sensor input such as point cloud data [37], [38].

Software bugs. Software can be buggy. Even with experienced programmers and extensive testing techniques, AV software is vulnerable to different kinds of bugs. Previous work [15] found 499 bugs in the two widely-used AV Software Autoware [26] and Baidu Applo [27]. These bugs exist in the perception, localization, planning and actuation of the AV software. 10.6% of the bugs will lead to a crash of the AV system in the end [15], showing that software bugs can be a serious threat to the safety of AVs.

C. Common Protection Mechanisms

Redundancy. Redundancy, both temporally and spatially, serves as an effective way of countering the threats to the AV software. Temporal redundancy refers to executing a part of the code more than once [39]–[41]. Redundant executions can help alleviate the threat of SDC caused by soft errors as they are transient. Temporal redundancy introduces significant

performance overhead. Executing each software module twice effectively halves the performance.

Spatial redundancy refers to executing the same algorithm using different physical hardware instances [42], [43]. For instance, Tesla's Full Self-Driving (FSD) chip makes two copies of the entire processing logic, effectively introducing a dual modular redundancy [44]. Modular redundancy has been shown to be effective against software errors [45] and adversarial attacks [46]. Modern processors usually provide hardware support to minimize the performance overhead of executing on identical hardware copies [47], [48]. As a result, the main overhead of special redundancy comes from the added silicon area and the associated non-recurring engineering costs, which are expected to increase as AV platforms are increasingly integrating specialized accelerators [49], [50].

Anomaly detection. As an alternative to redundancy, anomaly detection shields the errors at the sources. Unlike much traditional software, Av software process temporal inputs, i.e., sequences of sensors inputs, which exhibit strong temporal consistency. For example, when a car is driving in a straight lane, it is unlikely that the path planning module will issue a sudden acceleration to the actuator. Therefore, errors in AV software sometimes are manifested as outliers that break the temporal consistency. Different techniques on anomaly detection [51], [52] are proposed to detect outliers in AV software. Anomaly detection, however, introduces overhead due to the execution of the detection algorithm.

III. Understanding Inherent Fault Tolerance in Autonomous Machine Software Stack

We first describe our error injection methods (Sec. III-A). We then analyze how the fault propagates (Sec. III-B) to identify different masking mechanisms (Sec. III-C). From individual nodes' masking mechanisms, we describe how the fault-tolerance level of each node is derived (Sec. III-D).

A. Error Definitions and Injection

Autoware is a widely used AV software stack. We apply Autoware into the simulation platform of CARLA [53] to form realistic AV driving scenarios. Autoware is built with the support of Robotic Operating System (ROS), where different algorithms in Autoware is represented by a separate process or ROS node. Different ROS nodes communicate through ROS messages which is the output of each algorithms. All the ROS nodes and ROS messages formulate a large directed graph which we will refer to ROS graph in the following context. The ROS graph is a strict equivalence of Autoware. The error injection happens on our server with 8 Intel(R) Xeon(R) W-2123 CPUs and a Nvidia Quadro RTX 4000 GPU and is tested on the Ubuntu 18.04.5 system.

The goal of BRAUM error injection framework is to mimic different kinds of errors AV software may encounter while being lightweight enough to not influence the regular execution of AV software. We achieve this by injecting three types of errors in Autoware.

Soft errors. To mimic soft error-induced SDC, we leverage architectural-level register error injection. During BRAUM error injection, the selected victim ROS node will send out its process ID (PID) to the error injection process so that the injection process can attach to the victim ROS node via *ptrace* system call. The `ptrace` system call allows us to manipulate the register files of the selected victim ROS node. First, we randomly pick a general-purpose or floating point register and randomly pick a bit to flip. Second, we use `ptrace` and the PID of the running process to pause the execution, obtain the register value, inject the fault, and resume the execution. This architecture-level register error injection has little overhead, as shown in previous error injection tools [54], [55].

Adversarial attacks. To mimic potential adversarial attacks, our strategy is carefully corrupt the *output* of relevant ROS nodes. For the perception module, which mainly performs object detection and tracking, we emulate two common types of adversarial attacks [32], [33], non-targeted attack and targeted attack. For non-targeted attacks, we randomly change the detected object class to another class that exists in the dataset. For targeted attacks, the detected object class is randomly changed to another class commonly seen in AV (e.g., person, stop signs). To emulate corruptions in bounding boxes, we either create a very large bounding box (i.e., 240×240 pixels) when the ground truth is an empty box or remove the bounding box altogether if otherwise.

Adversarial attacks on the localization and planning modules are much less common in literature. We create our best-effort localization and planning attacks by assuming that an attack on localization moves the vehicle position away from the ground truth anywhere between 5m and 300m, similar to prior work [56], [57]. Attacks on planning are similarly emulated except we move the predicted future, rather than current, position of the vehicle.

Software bugs. To emulate software bugs, a randomly generated error is applied (added) to a node's output signal. Both adversarial attacks and software bugs are implemented by using the ROS topic publish mechanism with little overhead.

Summary. In total we cover 23 ROS nodes (and 26 ROS topics) as shown in Tbl. I. A campaign of 14,196 error injections was performed over 30 days. The 26 ROS topics cover virtually all the output topics, including localization, perception, planning and control in Autoware; the only topics/nodes that are not covered are those that are specific to the simulator itself (e.g., UI, saving data, visualization).

B. Error Propagation Analysis

Goal. We analyze the results of the fault injection campaign to understand how different forms of faults in different nodes are propagated to the output of the AV software. In particular, we have three goals. First, we aim to identify, for each injected error, whether it is propagated to the end of the ROS graph and, thus, corrupts the actuator commands. Second, in case an error is masked, i.e., invisible to the actuator, we aim to locate where the error is masked. Finally, we aim to identify the fault masking mechanism used to mask that error.

TABLE I: List of ROS nodes this paper analyzes.

Module	ROS node	EPR
Sensor Preprocessing	ray_ground_filter	0%
	voxel_grid_filter	0%
Localization	can_odometry	0%
	ndt_matching	23.4%
	pose_relay	8.7%
	vel_relay	2.4%
Perception	vision_darknet_detect	0%
	vision_beyond_track	0%
	detection_lidar_detector	0%
	detection_lidar_tracker	0%
	range_vision_fusion	0%
	naive_motion_predict	0%
	costmap_generator	0%
Planning and Control	astar_avoid	0%
	velocity_set	36.3%
	decision_maker	100%
	pure_pursuit	17.8%
	lane_stop	0%
	lane_rule	26.9%
	twist_filter	69.2%
	twist_gate	80.6%
	lane_select	0%
	waypoint_planner	0.68%

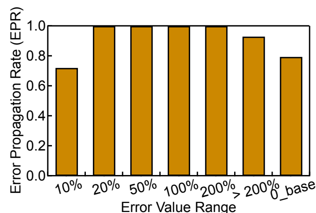


Fig. 2: Error propagation rate when injecting error into node *twist_gate* which does not have inherent masking.

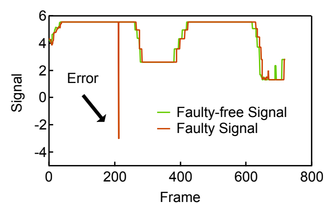


Fig. 3: One signal in the instructions to the actuators has been affected by the error.

Note that corruption in the actuator commands *might* not actually cause a safety issue, mainly because the closed-loop control in the software stack would very likely mitigate a transient command error in the current frame and re-align the vehicle back to the correct trajectory. In this sense, what is reported here is a conservative analysis of AV safety.

Establishing ground truth. To analyze whether and how errors are propagated, we must first obtain the ground-truth, i.e., fault-free results. To that end, we use the same input scenario used in fault injection to drive Autoware and record the output of *each ROS node* under analysis. To accommodate natural run-time variance, the same run is repeated 5,000 times to capture a distribution for each node's output.

Analyzing results. For each fault injection run, if the AV software stack provides no error masking mechanism, the output of the ROS graph, i.e., the actuator commands, will necessarily be corrupted. In contrast, if the actuator commands are uncorrupted (according to some metric), some form of error masking must have taken place between the node where the fault is injected and the output node. Our goal in this section is to identify the masking node. Next section describes the actual masking mechanisms we identified.

We first define that an ROS node is deemed to be corrupted by the fault, i.e., the error is propagated to this node, if the node output is "out of distribution", which is empirically defined as lying outside the bounds of the fault-free range by over three times of the mean value. This criterion is similar to what is used in prior work [58].

We analyze each fault injection run in a *backward* fashion, starting from the output node of the ROS graph and check if the fault propagates to this node. If not, the error injected must have been masked somewhere before the node. We then check whether the parent nodes are corrupted. We repeat this process until we reach the node where the fault is injected. This process help us to precisely locate the node, if any, that masks the injected fault.

Once we identify a fault-masking node, we then identify the actual masking mechanism in the node. This is done in a semi-automated way. We statically instrument the code to monitor how each input variable is used in the code. We then re-run the fault injection to capture the statement that masks the error. We then manually examine the code to understand how the error is actually masked. We discuss our findings next.

C. Masking Mechanisms

We classify the masking pattern exist in Autoware into four different categories: No Masking (NM), Attenuation (A), Unconditional Masking (UM), and Conditional Masking (CM). We characterize how often errors propagate to the actuator command using Error Propagation Rate (EPR). Tbl. I shows the EPR of all the nodes we inject.

No masking. Some nodes have no inherent masking mechanisms. Fig. 2 shows the EPR of the *twist_gate* node, where the x-axis shows the amplitude of error injection. For example, "10%" on the x-axis means the value after error injection is with in the range of 90% to 110% of the original value. "0_base" means the original value is 0.

Almost all the injected errors are propagated to the output because of a lack of inherent masking in *twist_gate*. The overall EPR is 82.1%. Fig. 3 shows an example, where an error injected into *twist_gate* causes drastic changes to the actuator commands.

Attenuation by low-pass filter. Attenuation mechanisms exist commonly in the source code of Autoware. Low-pass filter is a traditional way of filtering out high-frequency signals. Autoware uses a low-pass filter at the end part of the entire compute graph to smooth the output signals and avoid sudden changes. The carefully-designed low-pass filter will degrade the signal with a sudden change in its output, which is effective to errors with low amplitude. Fig. 4 shows the error propagation rate when the low-pass filter is applied. The EPR is significantly lower (59.7% in "10%" error value range, 67.0% in "20%" error value range) when the error amplitude is low but remains 100% when the error value range increase.

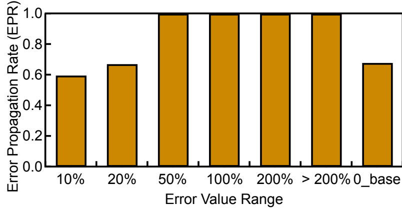
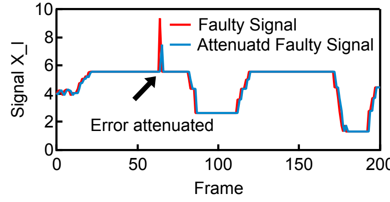
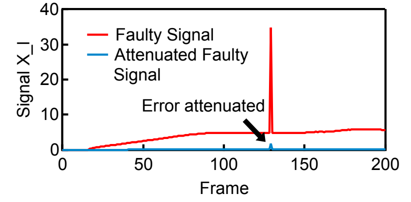


Fig. 4: Error propagation rate when inject error into node *twist_filter* which use low-pass filter to attenuate errors.



(a) Example of an error attenuated by the low-pass filter.



(b) Example of an error attenuated by the integral computation.

Fig. 5: Examples of signals with error attenuation mechanism such as low-pass filter and integral computation.

TABLE II: All the errors injected into the nodes that preprocess LIDAR point cloud data are masked.

ROS node	# of input points	# of faulty points injected	EPR
Ray_ground_filter	[1682,4019]	[0,80]	0%
Voxel_grid_filter	[915,1349]	[0,80]	0%

Fig. 5a shows an example of the use of the low-pass filter. The error amplitude is significantly reduced from 62.1% to 35.4%.

Attenuation by integral computation. Another typical way of attenuation in Autoware is integral computation. For example, in the localization module, specifically the *ndt_matching* node, the new estimation of pose and localization is calculated by an addition of the previous pose and the difference on distance and pose. The difference is calculated using the velocity estimation and interval on time. An error occurs on the velocity estimation node is largely attenuated as the time step is very small. Fig. 5b shows an example where the error is significant on the estimated velocity but attenuated to a small value afterwards.

Unconditional masking in sensor input preprocessing. We find there exist unconditional masking inside AV software which complete mask the errors happen in certain nodes. Unconditional masking technique exists in sensor input preprocessing node. Sensor input preprocessing nodes edit sensor input and sometimes remove redundant parts. For example, raw point cloud data from LIDAR sensor will be first filtered the points representing the ground and then used in point cloud object detection node. Such preprocessing nodes will generate massive sensor data that is naturally robust to errors. We inject errors into a LIDAR point cloud preprocessing node. The point cloud filter will filter out the points that represent the ground. The errors we inject manipulate the coordinates of the points after filtering. We change up to 80 points and monitor the output of the node that consume the faulty point cloud data.

We find that with the increase of faulty points injected into the point could filter node, all the outputs of the consumer node are not impacted with a relative standard deviation (RSD) of 0.002, -0.018 and 0.09 respectively.

Tbl. II shows the effect of injecting errors into the LIDAR point cloud preprocessing nodes. For each time, we inject an error that changes the coordinates of a certain number of points with an upper bond of 80 (2.9% and 6.7% of total output point cloud respectively). We find that all of these errors are masked and the final output to the actuators are not influenced.

Unconditional masking through multi-sensor fusion. Another unconditional masking technique utilized in AV software is multi-sensor fusion. AV software usually uses more than one source of sensor input during perception tasks such as detection and tracking. Both LIDAR and camera will capture the objects' information around the vehicle and usually the multi-sensor information will be fused together for higher accuracy and robustness [59], [60].

We find such a process is utilized in Autoware and illustrated in Fig. 6. The output object sequences from the vision branch and the LIDAR branch are first fused through a fusion node and used in the prediction module. The results of the prediction module will be fused again with the raw point cloud data in the costmap generation node afterwards. The costmap generation node will use the results of the second fusion to guide the vehicle to search for a path through the obstacle objects. Two continuous fusion techniques ensure that errors in the related perception nodes can be tolerated.

Fig. 7a illustrate the first fusion — between vision detection and tracking with LIDAR detection and tracking. The fusion algorithm first check whether the bounding boxes captured by both branches agree on the object label, position, and area. If so, as in the benign case, the fusion algorithm will keep the vision branch's bounding box and those LIDAR bounding boxes that do not have a match. When the vision branch is faulty, the fusion algorithm can not find matching bounding boxes, and the vision bounding box will be *discarded*. Thus, the faulty vision bounding box is masked.

Fig. 7b illustrates the second fusion case. The perception module will produce a sequence of bounding boxes after detection, tracking and prediction. The bounding boxes will be fused with the raw point cloud data to produce a map for the planning module. In the second fusion, a union operation will be performed on the bounding boxes and point cloud raw data to create the map. Thus, even if the faulty bounding boxes are not masked by the first fusion process, the error will be masked in the second process.

The EPR results verify the unconditional masking through multi-sensor fusion. We inject three kinds of errors, which are label errors, bounding box location errors and size errors, into all the nodes related to vision perception and LIDAR

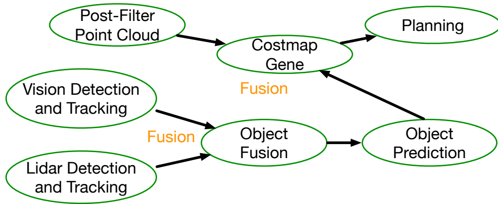
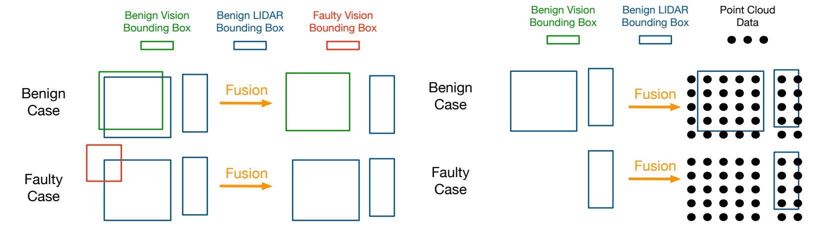


Fig. 6: Two types of fusion happen in the perception pipeline of Autoware. The first happen between vision and LIDAR perception, the second happen when creating a map for planning module.



(a) Vision detection and tracking results are fused with corresponding LIDAR branch. Mask the wrong results of vision branch.

(b) The fusion results of vision and LIDAR branch is further fused with the raw point cloud data.

Fig. 7: Detail procedure of two fusion processes.

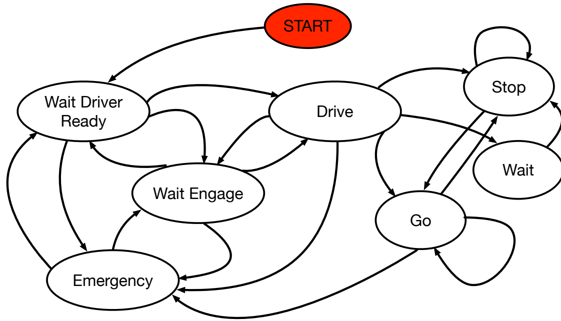


Fig. 8: Motion state machine used in Autoware.

perception branch. All of them are masked and will not influence the output signal to the actuators.

Conditional masking in state machine. Error masking under conditions also exists in Autoware. One of the most common examples is the conditional masking in the state machine of Autoware. Autoware manages the vehicle status with three complicated state machines: mission, motion and behavior state machine. Fig. 8 shows an illustration of the motion state machine. Eight different states with various transformation conditions consist of the motion state machine and provide enormous conditional masking patterns. For example, if in one frame the current state is *Wait Engage*, all the errors happen on the signals related to the *Go, Stop* and *Wait* states are masked as they are irrelevant to the *Wait Engage* state. Thus, all the nodes that produce these signals will not influence the final outputs as well.

Conditional masking in *If* statements. If statements create conditional masking patterns. Listing. 1 shows the snippet of the *velocity_set* node, which has has eight input signals and three output signals. With a driving scenario where both `condition1` and `condition2` are not satisfied, `obstacle_point` and `stopline_point` will not be read — even if they are corrupted.

To confirm this, Fig. 9 shows the EPRs when faults are injected to three nodes that generate inputs to the *velocity_set* node. In our particular runs, `condition1` and `condition2` are not satisfied; thus, faults in *pose_relay* and *velocity_relay* are naturally masked by *velocity_set*. However, `condition3` holds in certain runs; thus, the EPR under *astar_avoid* is not zero. Fig. 10 shows the execution traces. Fig. 10a compares the values of `obstacle_point`, an output of *velocity_set*, when a fault is injected to *astar_avoid*. Fig. 10b shows the value of `final_point`, another output of *velocity_set*, when a fault is injected into *pose_relay*. One can see that faults in the former are masked by *velocity_set* but faults in the latter are not.

```
Inputs: signal: current_pose, current_vel,
        points_no_ground,
        detection_range, cross_walk,
        decelerate_obstacle_point,
        obstacle_point, stopline_point, safety_point
    ;
        function: f,g, mp1, mp2, mp3
Outputs: obstacle_point, stopline_point, final_point

function velocity_set(){
  Bool condition1, condition2, condition3;
  condition1 = f(detection_range, cross_walk);
  condition2 = f(detection_range, cross_walk);
  condition3 = g(decelerate_obstacle_point);

  if (condition1 || condition2)
  {
      obstacle_point = mp1(current_pose, current_vel
    , points_no_ground);
      stopline_point = mp2(current_pose, current_vel
    , points_no_ground);
  }
  else
  {
      if (condition3)
      {
          final_point = mp3(safety_point);
      }
  }
  return obstacle_point, stopline_point, final_point
  }
```

Listing 1: Error masked by conditional if-statements

Summary. The understanding of different fault masking/attenuation mechanisms allow us to classify the fault tolerance level of an Autoware node. This is shown in Tbl. III. Among them, only two nodes (twist_gate and decision_maker) have no masking patterns. Both of them are at the end part of the Autoware graph and directly contributed to the output of the AV software. Two nodes have attenuation mechanisms. All the nodes in the perception pipeline are with unconditional masking mechanisms using two fusion operations. The rest

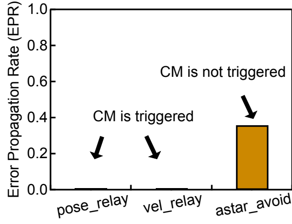
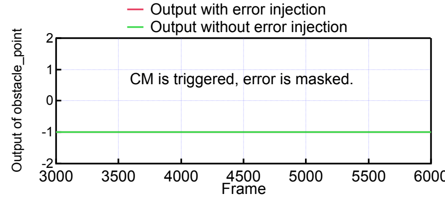
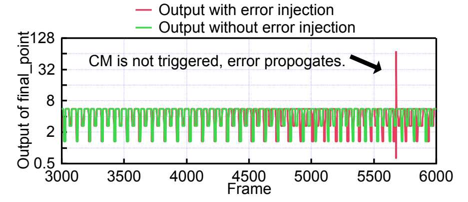


Fig. 9: EPRs when faults are injected to three producers of *velocity_set*.



(a) Faults in *pose_relay* are masked.



(b) Faults in *astar_avoid* are not masked.

Fig. 10: Outputs of the *velocity_set* node when faults are injected into its two producers *pose_relay* and *astar_avoid*.

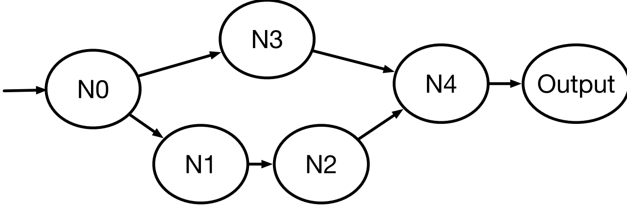


Fig. 11: A simple graph to illustrate how the dynamic FTL of a node is calculated.

nodes have conditional masking mechanisms.

We show the EPR in the last column. Not surprisingly, nodes without any masking mechanisms have the highest propagation rate, both are above 80%. Attenuation with a low pass filter can prevent a small number of errors from propagating and reduce the EPR to 69.2%. Integration shows a much stronger masking effect, only 17.8% of the errors finally propagate to the end and most of them are with large error values. Nodes with unconditional masking inherently are robust and have 0% EPR for all seven nodes. Nodes with conditional masking have various EPR, ranging from 0% to 36.3%.

D. Calculating Fault Tolerance Level

After each node is assigned with its inherent masking pattern, we can derive a node's fault tolerance level (FTL), which indicates whether/how a node's output errors can be masked/attenuated before reaching the actuator. Let us use a simple graph in Fig. 11, where we aim to calculate the FTL of node $N0$, to describe the intuition behind our algorithm.

First notice that $N0$'s sub-graph has two separate paths: $N1 \rightarrow N2 \rightarrow N4$ and $N3 \rightarrow N4$. That is, there are two paths for $N0$'s output error, if any, to be propagated to the output. Therefore, the FTL of $N0$ is the weaker of the two paths. Given a path, say, $N3 \rightarrow N4$, the error masking mechanisms of all nodes on the path are applied sequentially; therefore, the FTL of a path is the strongest FTL of all the nodes on the path. For instance, if $N3$ has inherent attenuation and $N4$ has inherent unconditional masking, the FTL of the path $N3 \rightarrow N4$ is unconditional masking, because the attenuation effort is "overwritten" by the unconditional masking. The overwriting behavior of unconditional masking, a stronger masking, takes place regardless of whether it occurs before or after attenuation, a weaker masking.

Formally, we define a partial order "weaker than", denoted $\prec$, between three masking mechanisms:

$$\mathsf{NM} \prec \mathsf{A} \prec \mathsf{UM}$$

Note that a conditional masking node, by definition, will be dynamically resolved to either no masking or unconditional masking depending on whether the node is triggered.

Given the intuition above, the FTL of $N0$ in Fig. 11, $F(N0)$, is calculated by the following equation:

$$F(N0) = min(max(F(N1), F(N2), F(N4)), \\ max(F(N3), F(N4))) \quad (1)$$

Notice how the equation is inherently iteratively defined: calculating $F(N0)$ requires first calculating $F(N1)$, $F(N2)$, $F(N3)$, and $F(N4)$. This suggests that to calculate the FTL of a node one must start from the *output* node of the graph. In practices, we reverse all the directions in the ROS DAG, start from the output node to perform a breath-first search, and calculate the FTL of all the nodes in a single traversal pass. The FTL of the output node is NM.

Tbl. III shows the FTL of all the nodes. We make two observations from Tbl. III. First, a node's FTL might be different from its inherent masking mechanism. This is because the FTL of a node, say P, depends on all *other* nodes between P and output. Second, the FTL of certain nodes vary at run time if there exists a CM node on the path to the output.

IV. The Braum Protection System

Based on the fault tolerance characterization, we propose a dynamic protection system based on analyzing the node vulnerability in the Autoware software stack. We first discuss a baseline protection mechanism (Sec. IV-A), followed by our protection scheme that reduces the protection overhead with little accuracy impact (Sec. IV-B).

A. Baseline Protection Mechanism

We first describe a baseline protection strategy, which is representative of those commonly found in literature [55] and provides strong protections at the cost of high overhead. Our work, however, does not fundamentally depend on this baseline scheme. Fig. 12 illustrate the baseline protection system described here.

TABLE III: Fault tolerance level (FTL) of all the Autoware ROS nodes under evaluation.

Node	Masking mechanism	FTL
twist_gate	NM	NM
decision_maker	NM	NM
twist_filter	A	A
pure_pursuit	A	A
vision_darknet_detect	NM	UM
vision_beyond_track	NM	UM
detection_lidar_detector	NM	UM
detection_lidar_tracker	NM	UM
range_vision_fusion	NM	UM
naive_motion_predict	NM	UM
costmap_generator	NM	NM/UM/A
ray_ground_filter	UM	UM
voxel_grid_filter	UM	UM
astar_avoid	CM	NM/UM
velocity_set	CM	NM/UM
lane_stop	CM	NM/UM
lane_rule	CM	NM/UM
waypoint_planner	CM	NM/UM
ndt_matching	CM	NM/UM/A
can_odometry	CM	NM/UM/A
pose_relay	CM	NM/UM/A
vel_relay	CM	NM/UM/A
lane_select	CM	NM/UM/A

The protection strategy operates at a node granularity. The basic idea is to monitor the output of a node and detect if the output is an outlier based on certain distribution. If an outlier is detected, we then re-execute the node (i.e., temporal redundancy). We call this "output outlier detection and re-execution" (OODR). In particular, we use a Gaussian-based Anomaly Detector (GAD). We maintain a mean value and a standard deviation σ in a fixed size window of 10 frames.

When the value of the output is N standard deviations away from the mean value, the recovery will be triggered. Critically, N is configurable based on the nature of the nodes. For example, we could use a larger N if a node's FTL is A, since slight fluctuation in the output is likely attenuated by the subsequent nodes. N would be higher or lower for nodes classified as NM and UM, respectively. The particular recovery scheme we consider is to re-execute the current node.

In addition to detecting the output outliers (and re-executing an abnormal node), we also detecting input outliers. In particular, we use the same GAD to detect any outlier in the inputs. If an outlier is detected, we replace the input with the an input value that is within the N sigma distance. We call this "input outlier detection and resetting" (IODR).

Sharp readers might wonder why such an input outlier detection and resetting is necessary: wouldn't protecting every node's output effectively protect every node's input? The reason is three-fold. First, it is possible that not all the nodes' outputs are protected, especially when a node's implementation is provided by a third-party library or when protecting a node is simply to costly (e.g., re-execution takes too much time). Second, re-execution does not fundamentally mitigate

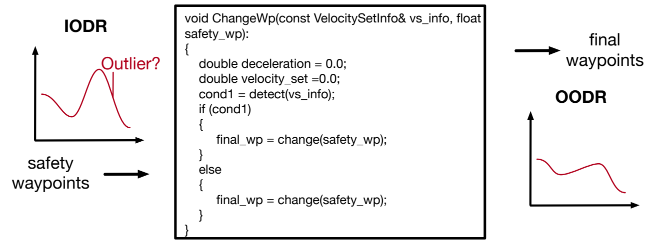


Fig. 12: Baseline protection system in BRAUM.

faults introduce by software bugs or adversarial attacks, for which input outlier detection and resetting is known to be effective [55]. Finally, a fault-free output could be corrupted during data transmission.

B. Our Protection System

They key insight behind our protection system is that if errors/faults in node's output can be inherently masked or attenuated later, one can relax the protection strength of the node and thus reduce the protection overhead. Algo. 1 shows the overall protection system.

At run time once a node finishes its execution, we first calculate the dynamic FTL of the node using the algorithm described in Sec. III-D (an example is shown in Equ. 1). Recall that this step must be executed dynamically for each frame, since a node's fault tolerance mechanism depends on how a CM is resolved at run time.

Calculating the FTL of a node requires us to resolve all the yet-to-be executed CM nodes in the sub-graph of the current node. Thus, we must predict how each downstream CM is resolved. For simplicity, we use a last-value predictor, i.e., using the resolution of CM in the last frame as the prediction of the current frame. This is inspired by recent work that shows that autonomous machine states have temporal consistency [61], where sudden state changes are rare.

If a node's dynamic FTL is NM, the outlier detection and temporal re-execution is triggered as usual. However, if a node's FTL is stronger than NM, we could potentially elide the re-execution. Specifically, if a node's dynamic FTL is UM, we can skip outlier detection and temporal re-execution altogether, since any output error is expected to be masked down the line. If a node's FTL is A, we relax the outlier detection threshold (use a larger N in Sec. IV-A) in that slight change in output could be attenuated later in the execution. Relaxing the outlier detection threshold could reduce the frequency of node re-execution, improving performance.

Handling Mis-predictions. Just like a mis-prediction in a processor must be dealt with to avoid incorrect pipeline execution, the mis-prediction of a CM node's resolution must be taken care of as well. In particular, mis-predicting the resolution of a CM node P could alter the FTL of a node that P depends on. For instance in Fig. 11, if $N2$ is a CM node that should have been resolved to NM but, instead, is predicted as UM, both $N0$ and $N1$'s FTL should be re-calculated, resulting in different protection schemes for both nodes.

Algorithm 1: BRAUM protection system.

Input: Current node T; $T(\cdot)$ represents the execution of the node; fault masking mechanism of each node in ROS graph; output outlier detection threshold N; slack in outlier detection k.

Resolve all the CM nodes in the graph;
$FTL_T \leftarrow$ Calculate the FTL of the current node T;
if FTL_T *is UM* **then**
  $T(\cdot)$;
end
if FTL_T *is NM* **then**
  Run input outlier detection and resetting with threshold N;
  $T(\cdot)$;
  Run output outlier detection and re-execution with threshold N;
end
if FTL_T *is A* **then**
  Run input outlier detection and resetting with threshold N;
  $T(\cdot)$;
  Run output outlier detection and re-execution with threshold $N + k$;
end
if *fault masking mechanism of* T *is CM* **then**
  if T*'s resolution target is mis-predicted* **then**
    Re-evaluate the FTL for all the nodes that are before T in the ROS graph;
    Re-execute the current frame from the first nodes whose actual FTLs are weaker than previous predicted;
  end
end

Our observation is that before entering a CM node we know exactly whether it would be resolved as UM and NM, from which we know whether we have mis-predicted the resolution target of this CM node. Upon a mis-prediction, we will recalculate the FTLs of all the nodes whose FTLs depend on ths CM node. If the actual FTL of a node, say P, is *weaker* than the predicted FTL, that means the protection strategy applied to P should have been *stronger*.

To deal with mis-predictions, we identify the first node in the entire ROS graph whose actual FTLs are *weaker* than what were previously predicted and re-execute the current frame from there. Note that while mis-prediction does increase the frame latency, it does *not* affect a vehicle's behavior because the output of mis-predicted execution would not have reached the vehicle actuator yet when the mis-prediction is detected.

V. EVALUATION

We first describe the evaluation setup (Sec. V-A) and demonstrate the effectiveness and overhead (Sec. V-B).

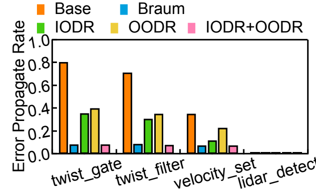


Fig. 13: Error propagation rate largely reduced when BRAUM protection is applied.

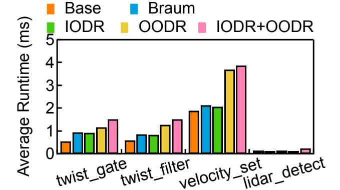


Fig. 14: Runtime overhead is minimum when BRAUM protection is applied.

A. Evaluation Setup

Nodes with protection. As a proof of concept, we pick four representative ROS nodes to implement our protection system, assuming that faults take place in only those nodes.

- *twist_gate*, whose FTL is NM.
- *twist_filter*, whose FTL is A.
- *detection_lidar_detector*, whose FTL is UM.
- *velocity_set*, whose FTL is either NM or UM, since one of its subsequent nodes has conditional masking.

Baselines. We compare with four different baselines.

- **Base**: the vanilla AV software without any fault protection.
- **IODR+OODR**: a system that performs both input outlier detection and resetting (IODR) and output outlier detection and re-execution (OODR).
- **IODR**: a system with only IODR.
- **OODR**: a system with only OODR.

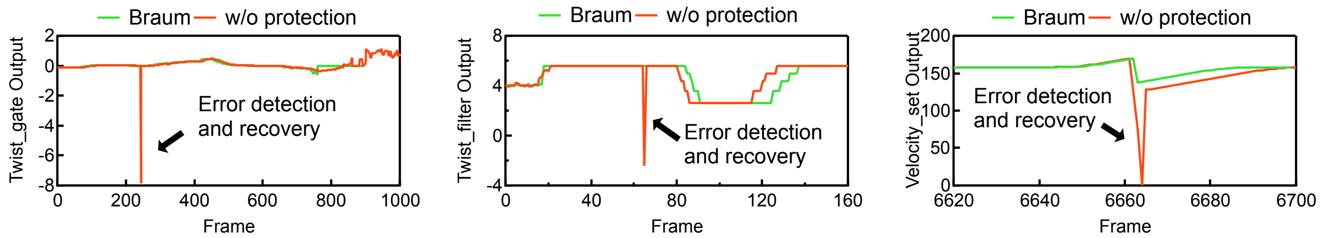
Implementation. Our protection system is implemented at the software level by modifying the Autoware source code after the analysis is performed. The source code of the four vulnerable nodes is enhanced with our protection mechanism and Autoware is then re-built from source. For evaluation, we inject the exact same errors as in Sec. III to test the effectiveness of BRAUM protector.

B. Protection Results and Overhead

Faults in *twist_gate*. Fig. 13 compares the EPR before and after the protection scheme is applied. BRAUM reduces the error propagation rate from 80.6% down to 8.3%. Fig. 15a shows a concrete example of how the protection scheme masks the error in *twist_gate*. We inject a significant error at frame 220 and BRAUM successfully detects the error and recovers the correct value. Both **IODR** and **OODR** are unable to achieve similar protection result. They reduce the EPR to 35.7% and 40%, respectively. **IODR+OODR** achieves same protection effectiveness compared to BRAUM.

The reason EPR is not further reduced is that in certain cases, the input signal does have a sudden change, yet the protector treats it as an outlier and replaces it with the average value in the previous window. An error created by the protector thus will propagate and result in a non-perfect protector. For example, in *twist_gate*, 97.3% of the protection failure cases are caused by false positives. The data is 94.5% for *twist_filter*.

Faults in *twist_filter*. The FTL of *twist_filter* is A (Tbl. III), for which we empirically loose the output outlier detection



(a) Example of an error detected and recovered by BRAUM protector in twist_gate node.

(b) Example of an error detected and recovered in twist_filter node.

(c) Example of an error detected and recovered in velocity_set node.

Fig. 15: Concrete examples of how the BRAUM protection works.

threshold to 6 $sigma$. Loosening the detection threshold can help reduce the frequency of re-execute the code, thus saving protection overhead. BRAUM protection reduces the EPR of *twist_filter* by 87.6%, **IODR** is able to reduce the EPR by 56.6% and the result for **OODR** is 50.1%. Fig. 15a shows an example of how the protection scheme works. **IODR+OODR** achieves slight better EPR (8.1%) compared to BRAUM.

Faults in *velocity_set*. Fig. 13 shows the error propagation rate after the protection is applied on *velocity_set*. The EPR reduce from 36.3% to 7.6%. We also find that the simple predictor we implemented has a very high accuracy. The mis-prediction rate is only 2.7%. Fig. 15c shows a concrete example of error mitigated in *velocity_set*.

Faults in *detection_lidar_detector*. Although we do not perform any protection on *detection_lidar_detector*, the EPRs are all 0 due to unconditional masking patterns.

Protection on unseen errors. We evaluate our protection method on a set of unseen errors with the same scenarios. The unseen errors has the same amount compared to the original evaluation and within the same range of error amplitude.

For the new error set, the EPR (lower the better) in the 4 nodes are 8.2%, 7.2%, 8.3%, and 0%, similar to the current results reported in the paper (8.3%, 8.8%, 7.6% and 0%), indicating our method is still effective in unseen errors. We also perform another evaluation on the protection method by using a different scenario with a different error set compared to the analysis phase. As an example, if faults take places in the node *twist_gate*, our protection method reduces the EPR to 9.1%, similar to the 8.3% results we currently have.

Protection overhead. BRAUM protector is lightweight and brings minimum overhead to the AV systems. We show the average runtime comparison between the baseline and after applying BRAUM protector in Fig. 14. For *twist_gate* and *twist_filter*, the overhead is relatively higher (50.1% in *twist_gate* and 43.3% in *twist_filter*). This is because those two nodes are extremely lightweight that do not perform any complicated computation, both have an average runtime of less than 0.6ms. The overhead reduce to 11.2% in *velocity_set*. 0% overhead is introduced on *detection_lidar_detector*.

As compared to **IODR**, BRAUM has an average 2.5% higher runtime overhead as the use of output protection. However, the runtime is saved by 47.2% compared to **OODR**, as the always re-execution adds enormous overhead. **IODR+OODR** has the highest protection protection overhead, BRAUM saves runtime by 55% compared to **IODR+OODR**.

As a comparison with existing protection methods, we compare with DeepFense [46] and Dual-core Lock Step (DCLS) [62]. DeepFense utilizes redundancy to protect perception modules. BRAUM achieves the same protection accuracy with 93.75% less run-time overhead. DCLS is a hardware mechanism to detect and recover from hardware transient faults. BRAUM reduces the error protection rate from 35.2% to 7.9%. DCLS requires two times hardware area overhead, whereas BRAUM requires none.

VI. RELATED WORK

Error injection into software. Simulating the errors that can possibly happen in software has been studied by prior works. Such errors include soft errors [54], [55], adversarial attacks [32], [33], [63] and software bugs [15]. Most of these works try to relate errors they simulate with a metric such as mission success rate and quality of service (QoS) to demonstrate the errors injected do create reliability issues. We, however, instead of only caring about how many errors finally propagate to the end with EPR metric, also try to understand what types of fault tolerance mechanisms successfully mask the errors at the software level.

Masking of the errors. The masking of soft errors takes place at circuit level [64], [65], micro-architecture level [30], [66], and architecture level [67], [68]. We go one step further to understand the inherent fault masking of the entire AV software stack. We show that even with all the error masking down the system stack, specific operations (i.e., integration) or interactions from multiple algorithms (i.e., fusion) in the AV software present new fault masking opportunities.

Protecting AV software. To counter errors happen in AV software, two categories of protection methods have been proposed. The first one is utilizing modular redundancy, both temporal [39]–[41] and spatial [42], [43]. The second category is to detect anomaly outputs and recover [55]. Most methods are agnostic to all the nodes in AV software and bring heavy overhead. We propose selective protection, which spends limited resource on the most vulnerable nodes such as the ones with no fault tolerance mechanisms inherently.

VII. DISCUSSIONS

Other AV software and machines. Our work focuses on Autoware. The core idea of this work, which is to identify

software-inherent error masking capabilities and relax protection overhead, is generalizable. However, we do *not* claim that the exact conclusions obtained on Autoware generalize to other software stacks such as Baidu Apollo. For example, the fusion algorithm in Baidu Apollo fuses perception results from three different sensors and uses a more complex fusion algorithms compared to that in Autoware. Thus, the perception module in Apollo might be even stronger than that in Autoware. In addition, the infrastructure-vehicle cooperative paradigm presents new challenges and opportunities for reliability [69].

Similarly, our overall method of analyzing and selectively protecting faults applies to other autonomous machines such as drones and mobile robots, but details might vary. For example, *if* a mobile robot operates in an environment where adversarial attacks are rare, the fault injection campaign could be more focused on the other two sources of faults. In addition, what is defined as an error might vary: drones might care about arrival percentage and quality of flight [55].

Implication to robust AV system design. We show that an AV system can spend limited resources on protecting the vulnerable nodes, which reveals broad opportunities space for designing robust and efficient AV system. For instance, if the costs of trusted execution, encryption, and decryption are limited, they should be prioritized over the most vulnerable components in the software (i.e., with high FTLs).

VIII. Acknowledgement

This work was supported in part by Semiconductor Research Corporation (SRC) Artificial Intelligence Hardware (AIHW) program. Yuhao Zhu and Jingwen Leng are corresponding authors.

IX. Conclusion

AV software is vulnerable to different sources of errors such as hardware soft errors, adversarial attacks and software bugs. Instead of blindly protecting the entire software stack, Braum proposes selectively protecting software modules based on their fault tolerance levels in order to reduce the protection overhead. We demonstrate protection can be effective yet with low overhead, opening up large opportunities for dynamic protection on AV software.

References

[1] N. Kalra and S. M. Paddock, “Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability?” *Transportation Research Part A: Policy and Practice*, vol. 94, pp. 182–193, 2016.

[2] P. Koopman and M. Wagner, “Autonomous vehicle safety: An interdisciplinary challenge,” *IEEE Intelligent Transportation Systems Magazine*, vol. 9, no. 1, pp. 90–96, 2017.

[3] B. Yu, W. Hu, L. Xu, J. Tang, S. Liu, and Y. Zhu, “Building the computing system for autonomous micromobility vehicles: Design constraints and architectural optimizations,” in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 1067–1081.

[4] N. E. Boudette, “‘it happened so fast’: Inside a fatal tesla autopilot crash,” 2021. [Online]. Available: https://www.irishtimes.com/life-and-style/motors/it-happened-so-fast-inside-a-fatal-tesla-autopilot-crash-1.4650265

[5] ——, “U.s. will investigate tesla’s autopilot system over crashes with emergency vehicles,” 2021. [Online]. Available: https://www.nytimes.com/2021/08/16/business/tesla-autopilot-nhtsa.html

[6] J. Moody, N. Bailey, and J. Zhao, “Public perceptions of autonomous vehicle safety: An international comparison,” *Safety science*, vol. 121, pp. 634–650, 2020.

[7] A. Reschka, “Safety concept for autonomous vehicles,” in *Autonomous Driving*. Springer, 2016, pp. 473–496.

[8] Y. Gan, Y. Qiu, J. Leng, M. Guo, and Y. Zhu, “Ptolemy: Architecture support for robust deep learning,” in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 241–255.

[9] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, “The soft error problem: An architectural perspective,” in *11th International Symposium on High-Performance Computer Architecture*. IEEE, 2005, pp. 243–247.

[10] A. Dixit and A. Wood, “The impact of new technology on soft error rates,” in *2011 International Reliability Physics Symposium*. IEEE, 2011, pp. 5B–4.

[11] M. Zhang and N. R. Shanbhag, “Soft-error-rate-analysis (sera) methodology,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 2140–2155, 2006.

[12] S. Komkov and A. Petiushko, “Advhat: Real-world adversarial attack on arcface face id system,” in *2020 25th International Conference on Pattern Recognition (ICPR)*. IEEE, 2021, pp. 819–826.

[13] Y. Guo, X. Wei, G. Wang, and B. Zhang, “Meaningful adversarial stickers for face recognition in physical world,” *arXiv preprint arXiv:2104.06728*, 2021.

[14] H. Lengyel, V. Remeli, and Z. Szalay, “A collection of easily deployable adversarial traffic sign stickers,” *at-Automatisierungstechnik*, vol. 69, no. 6, pp. 511–523, 2021.

[15] J. Garcia, Y. Feng, J. Shen, S. Almanee, Y. Xia, Chen, and Q. Alfred, “A comprehensive study of autonomous vehicle bugs,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 385–396.

[16] P. Koopman and M. Wagner, “Challenges in autonomous vehicle testing and validation,” *SAE International Journal of Transportation Safety*, vol. 4, no. 1, pp. 15–24, 2016.

[17] J. A. Abraham and D. P. Siewiorek, “An algorithm for the accurate reliability evaluation of triple modular redundancy networks,” *IEEE Transactions on Computers*, vol. 100, no. 7, pp. 682–692, 1974.

[18] C. Engelmann, H. H. Ong, and S. L. Scott, “The case for modular redundancy in large-scale high performance computing systems,” in *Proceedings of the 8th IASTED international conference on parallel and distributed computing and networks (PDCN)*, 2009, pp. 189–194.

[19] E. P. Kim and N. R. Shanbhag, “Soft n-modular redundancy,” *IEEE Transactions on Computers*, vol. 61, no. 3, pp. 323–336, 2010.

[20] Y. Qiu, J. Leng, C. Guo, Q. Chen, C. Li, M. Guo, and Y. Zhu, “Adversarial defense through network profiling based path extraction,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 4777–4786.

[21] M. Ahmed, A. N. Mahmood, and J. Hu, “A survey of network anomaly detection techniques,” *Journal of Network and Computer Applications*, vol. 60, pp. 19–31, 2016.

[22] V. Chandola, A. Banerjee, and V. Kumar, “Anomaly detection: A survey,” *ACM computing surveys (CSUR)*, vol. 41, no. 3, pp. 1–58, 2009.

[23] A. Patcha and J.-M. Park, “An overview of anomaly detection techniques: Existing solutions and latest technological trends,” *Computer networks*, vol. 51, no. 12, pp. 3448–3470, 2007.

[24] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek, “Intrusion recovery using selective re-execution,” in *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[25] A. Roth, “Store vulnerability window (svw): Re-execution filtering for enhanced load optimization,” in *32nd International Symposium on Computer Architecture (ISCA’05)*. IEEE, 2005, pp. 458–468.

[26] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi, “Autoware on board: Enabling autonomous vehicles with embedded systems,” in *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, 2018, pp. 287–296.

[27] “Baidu Apollo team (2017), Apollo: Open Source Autonomous Driving, howpublished = https://github.com/apolloauto/apollo, note = Accessed: 2019-02-11.”

[28] X. Iturbe, B. Venu, and E. Ozer, “Soft error vulnerability assessment of the real-time safety-related arm cortex-r5 cpu,” in *2016 IEEE In-*

ternational Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE, 2016, pp. 91–96.

[29] J. Blome, S. Mahlke, D. Bradley, and K. Flautner, “A microarchitectural analysis of soft error propagation in a production-level embedded microprocessor,” in *In Proceedings of the First Workshop on Architecture Reliability*. Citeseer, 2005.

[30] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, “A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor,” in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36*. IEEE, 2003, pp. 29–40.

[31] V. Bandeira, I. Oliveira, F. da Rosa, R. Reis, and L. Ost, “Soft error reliability analysis of autonomous vehicles software stack,” in *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, 2019, pp. 253–254.

[32] X. Yuan, P. He, Q. Zhu, and X. Li, “Adversarial examples: Attacks and defenses for deep learning,” *IEEE transactions on neural networks and learning systems*, vol. 30, no. 9, pp. 2805–2824, 2019.

[33] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, “Towards deep learning models resistant to adversarial attacks,” *arXiv preprint arXiv:1706.06083*, 2017.

[34] J. Li, F. Schmidt, and Z. Kolter, “Adversarial camera stickers: A physical camera-based attack on deep learning systems,” in *International Conference on Machine Learning*. PMLR, 2019, pp. 3896–3904.

[35] B. Phan, F. Mannan, and F. Heide, “Adversarial imaging pipelines,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 16 051–16 061.

[36] Y. Zhang, B. Dong, and F. Heide, “All you need is raw: Defending against adversarial attacks with camera image pipelines,” *arXiv preprint arXiv:2112.09219*, 2021.

[37] D. Liu, R. Yu, and H. Su, “Extending adversarial attacks and defenses to deep 3d point cloud classifiers,” in *2019 IEEE International Conference on Image Processing (ICIP)*. IEEE, 2019, pp. 2279–2283.

[38] J. Zhang, L. Chen, B. Liu, B. Ouyang, Q. Xie, J. Zhu, W. Li, and Y. Meng, “3d adversarial attacks beyond point cloud,” *arXiv preprint arXiv:2104.12146*, 2021.

[39] B. K. Kim, “Reliability analysis of real-time controllers with dual-modular temporal redundancy,” in *Proceedings Sixth International Conference on Real-Time Computing Systems and Applications. RTCSA’99 (Cat. No. PR00306)*. IEEE, 1999, pp. 364–371.

[40] A. T. Rivers, *Modeling software reliability during non-operational testing*. North Carolina State University, 1998.

[41] J. Oplinger and M. S. Lam, “Enhancing software reliability with speculative threads,” *ACM SIGARCH Computer Architecture News*, vol. 30, no. 5, pp. 184–196, 2002.

[42] 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.

[43] F. L. Kastensmidt, L. Sterpone, L. Carro, and M. S. Reorda, “On the optimal design of triple modular redundancy logic for sram-based fpgas,” in *Design, Automation and Test in Europe*. IEEE, 2005, pp. 1290–1295.

[44] D. D. S. Pete Bannon, Ganesh Venkataramanan, “Fsd chip - tesla,” 2019. [Online]. Available: https://en.wikichip.org/wiki/tesla_(car_company)/fsd_chip

[45] K. S. Yim, V. Sidea, Z. Kalbarczyk, D. Chen, and R. K. Iyer, “A fault-tolerant programmable voter for software-based n-modular redundancy,” in *2012 IEEE Aerospace Conference*. IEEE, 2012, pp. 1–20.

[46] B. D. Rouhani, M. Samragh, M. Javaheripi, T. Javidi, and F. Koushanfar, “Deepfense: Online accelerated defense against adversarial deep learning,” in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[47] Á. B. de Oliveira, G. S. Rodrigues, and F. L. Kastensmidt, “Analyzing lockstep dual-core arm cortex-a9 soft error mitigation in freertos applications,” in *Proceedings of the 30th Symposium on Integrated Circuits and Systems Design: Chip on the Sands*, 2017, pp. 84–89.

[48] A. B. de Oliveira, G. S. Rodrigues, F. L. Kastensmidt, N. Added, E. L. Macchione, V. A. Aguiar, N. H. Medina, and M. A. Silveira, “Lockstep dual-core arm a9: Implementation and resilience analysis under heavy ion-induced soft errors,” *IEEE Transactions on Nuclear Science*, vol. 65, no. 8, pp. 1783–1790, 2018.

International Symposium on High-Performance Computer Architecture (HPCA). IEEE, 2021, pp. 827–840.

[49] Y. Gan, Y. Bo, B. Tian, L. Xu, W. Hu, S. Liu, Q. Liu, Y. Zhang, J. Tang, and Y. Zhu, “Eudoxus: Characterizing and accelerating localization in autonomous machines industry track paper,” in *2021 IEEE*

[50] W. Liu, B. Yu, Y. Gan, Q. Liu, J. Tang, S. Liu, and Y. Zhu, “Archytas: A framework for synthesizing and dynamically optimizing accelerators for robotic localization,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 479–493.

[51] C. Sample and K. Schaffer, “An overview of anomaly detection,” *IT Professional*, vol. 15, no. 1, pp. 8–11, 2013.

[52] S. He, J. Zhu, P. He, and M. R. Lyu, “Experience report: System log analysis for anomaly detection,” in *2016 IEEE 27th international symposium on software reliability engineering (ISSRE)*. IEEE, 2016, pp. 207–218.

[53] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, “Carla: An open urban driving simulator,” in *Conference on robot learning*. PMLR, 2017, pp. 1–16.

[54] V. Porpodas, “Zofi: Zero-overhead fault injection tool for fast transient fault coverage analysis,” *arXiv preprint arXiv:1906.09390*, 2019.

[55] Y.-S. Hsiao, Z. Wan, T. Jia, R. Ghosal, A. Raychowdhury, D. Brooks, G.-Y. Wei, and V. J. Reddi, “Mavfi: An end-to-end fault analysis framework with anomaly detection and recovery for micro aerial vehicles,” *arXiv preprint arXiv:2105.12882*, 2021.

[56] M. Patil, X. Wang, X. Wang, and S. Mao, “Adversarial attacks on deep learning-based floor classification and indoor localization,” in *Proceedings of the 3rd ACM Workshop on Wireless Security and Machine Learning*, 2021, pp. 7–12.

[57] X. Wang, X. Wang, S. Mao, J. Zhang, S. C. Periaswamy, and J. Patton, “Adversarial deep learning for indoor localization,” *IEEE Internet of Things Journal*, 2022.

[58] J. Nitsch, M. Itkina, R. Senanayake, J. Nieto, M. Schmidt, R. Siegwart, M. J. Kochenderfer, and C. Cadena, “Out-of-distribution detection for automotive perception,” in *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*. IEEE, 2021, pp. 2938–2943.

[59] H. Gao, B. Cheng, J. Wang, K. Li, J. Zhao, and D. Li, “Object classification using cnn-based fusion of vision and lidar in autonomous vehicle environment,” *IEEE Transactions on Industrial Informatics*, vol. 14, no. 9, pp. 4224–4231, 2018.

[60] X. Du, M. H. Ang, and D. Rus, “Car detection for autonomous vehicle: Lidar and vision fusion approach through deep learning framework,” in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017, pp. 749–754.

[61] P. Li, P. Wang, K. Berntorp, and H. Liu, “Exploiting temporal relations on radar perception for autonomous driving,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 17 071–17 080.

[62] M. Peña-Fernández, A. Serrano-Cases, A. Lindoso, M. García-Valderas, L. Entrena, A. Martínez-Álvarez, and S. Cuenca-Asensi, “Dual-core lockstep enhanced with redundant multithread support and control-flow error detection,” *Microelectronics Reliability*, vol. 100, p. 113447, 2019.

[63] Z. Xiong, H. Xu, W. Li, and Z. Cai, “Multi-source adversarial sample attack on autonomous vehicles,” *IEEE Transactions on Vehicular Technology*, vol. 70, no. 3, pp. 2822–2835, 2021.

[64] M. P. Baze, S. P. Buchner, and D. McMorrow, “A digital cmos design technique for seu hardening,” *IEEE Transactions on Nuclear Science*, vol. 47, no. 6, pp. 2603–2608, 2000.

[65] S. Lin, Y.-B. Kim, and F. Lombardi, “Design and performance evaluation of radiation hardened latches for nanoscale cmos,” *IEEE transactions on very large scale integration (VLSI) systems*, vol. 19, no. 7, pp. 1315–1319, 2010.

[66] D. Borodin and B. H. Juurlink, “Protective redundancy overhead reduction using instruction vulnerability factor,” in *Proceedings of the 7th ACM international conference on Computing frontiers*, 2010, pp. 319–326.

[67] V. Sridharan and D. R. Kaeli, “Quantifying software vulnerability,” in *Proceedings of the 2008 Workshop on Radiation effects and fault tolerance in nanometer technologies*, 2008, pp. 323–328.

[68] J. Leng, A. Buyuktosunoglu, R. Bertran, P. Bose, Q. Chen, M. Guo, and V. J. Reddi, “Asymmetric resilience: Exploiting task-level idempotency for transient error recovery in accelerator-based systems,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 44–57.

[69] S. Liu, B. Yu, J. Tang, Y. Zhu, and X. Liu, “Communication challenges in infrastructure-vehicle cooperative autonomous driving: A field deployment perspective,” *IEEE Wireless Communications*, 2022.