MAVFI: An End-to-End Fault Analysis Framework with Anomaly Detection and Recovery for Micro Aerial Vehicles

Yu-Shun Hsiao*[1], Zishen Wan*[1,2], Tianyu Jia[1], Radhika Ghosal[1], Arijit Raychowdhury[2], David Brooks[1], Gu-Yeon Wei[1], and Vijay Janapa Reddi[1]

[1]Harvard University [2]Georgia Institute of Technology

yushun_hsiao@g.harvard.edu, zishenwan@gatech.edu

ABSTRACT

Reliability and safety are critical in autonomous machine services, such as autonomous vehicles and aerial drones. In this paper, we first present an open-source Micro Aerial Vehicles (MAVs) reliability analysis framework, MAVFI, to characterize transient fault's impacts on the end-to-end flight metrics, e.g., flight time, success rate. Based on our framework, it is observed that the end-to-end fault tolerance analysis is essential for characterizing system reliability. We demonstrate the planning and control stages are more vulnerable to transient faults than the visual perception stage in the common "Perception-Planning-Control (PPC)" compute pipeline. Furthermore, to improve the reliability of the MAV system, we propose two low overhead anomaly-based transient fault detection and recovery schemes based on Gaussian statistical models and autoencoder neural networks. We validate our anomaly fault protection schemes with a variety of simulated photo-realistic environments on both Intel i9 CPU and ARM Cortex-A57 on Nvidia TX2 platform. It is demonstrated that the autoencoder-based scheme can improve the system reliability by 100% recovering failure cases with less than 0.0062% computational overhead in best-case scenarios. In addition, MAVFI framework can be used for other ROS-based cyber-physical applications and is open-sourced at https://github.com/harvard-edge/MAVBench/tree/mavfi.

1. INTRODUCTION

Autonomous Unmanned Aerial Vehicles (UAVs) are predicted to have a significant market share for a variety of applications, such as package delivery [1], search and rescue [2], surveillance [3], sports photography [4], and more [5]. Micro Aerial Vehicles (MAVs) correspond to a sub-class of UAVs weighing less than 2kg and a limited flight range (shorter than 5km). MAVs' small form factor lends them greater maneuverability than larger UAVs, making them promising candidates for certain UAV applications. However, practical reliability considerations (i.e., completing unmanned tasks safely without collision) make MAVs and other autonomous vehicles challenging to deploy in a wide range of real-life scenarios. Due to autonomous machines' high-reliability requirements, there is a strong demand to understand sources of vulnerability and develop protection schemes.

*These two authors contributed equally to this work and are listed in alphabetical order.

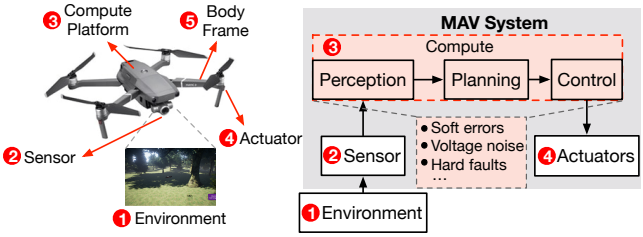


Figure 1: MAV system with Perception-Planning-Control compute paradigm and the common fault sources.

It is challenging to perform a comprehensive fault analysis for the MAV system as they are complex cyber-physical systems that include sensors, computing software (for perception, planning, and control) and hardware, and rotors or actuators [6, 7], as shown in Fig. 1. Within the system, there are multiple fault sources that could impact MAV's reliability. For example, environmental noises [8], adversarial attacks [9], and the inherent noises of each sensor can all create perturbation of sensor data. In the compute subsystem, there are soft errors, i.e. bit flip, may happen due to radiation [10], voltage noise [11]. During actuation, pitch, roll, and yaw displacements [12], or dynamic system errors [13] can happen by various types of faults that can impact MAV flight time and energy efficiency. In the past, most prior arts only focused on the mitigation of sensor data perturbation by using sensor fusion [8], anomaly detection [14], and neural network [15]. However, according to the California Department of Motor Vehicles (DMV), 64% of disengagements resulted from faulty or untimely decisions made by the computing system [16], which have not been fully analyzed. In [17], it is reported that soft errors can significantly impact the reliability of autonomous vehicles in multiple safety-critical scenes.

Transient fault injection and resiliency analysis have been extensively studied for isolated kernels on CPU and GPU, as shown in Tab. 1. However, prior methods [18, 19, 20, 22, 23] focusing on single algorithm fault analysis, which cannot be applied to analyze the end-to-end system behavior of MAV under transient faults. More recently, SASSIFI [21] and DriveFI [17] explored the reliability impact of transient fault for autonomous driving systems on power-hungry GPU platforms while without clear explanations of the portability. Therefore, there is no suitable fault injection framework for the end-to-end reliability analysis of MAV applications that

Table 1: Comparison between MAVFI and prior fault injection methods.

	LLFI [18]	**PINFI** [19]	**CLEAR** [20]	**SASSIFI** [21]	**DriveFI** [17]	**MAVFI (This work)**
OS	Linux	Linux	N/A	N/A	N/A	ROS + Linux
Analysis	Isolated	Isolated	Isolated	Isolated	End-to-end	End-to-end
Bit-flips	Single, double	Single	Single	Single, double	Single, double	Single, multiple
Benchmark	Parboil	SPEC, SPLASH	SPEC, DARPA	Rodinia, DOE	DriveAV, Apollo	MAVBench
Platform	Intel Xeon E5	Intel Core i7	Leon 3, IVM	Tesla K20, K40	Pegasus	Intel Xeon i9 / ARM Cortex-A57
Injection level	IR-level	Assembly-level	RLT-level	Assembly-level	Variable-level	Assembly-level, variable-level

typically run on CPU with Robot Operating System (ROS).

In this work, we developed an open-source fault injection and end-to-end reliability analysis framework for MAV application, MAVFI. The proposed MAVFI framework supports the capability of end-to-end reliability analysis by considering the transient error propagation between kernels, which breaks the analysis bounds of previous isolated fault analysis. During the kernel execution, multiple compute kernels are cooperatively running to stream the real-time sensor data across Perception-Planning-Control (PPC) compute pipeline. The lightweight and widely used robotic operating system ROS is leveraged to support inter-kernel communications. With the MAVFI framework, extensive end-to-end fault characterizations have been performed, which indicates that end-to-end fault tolerance analysis (i.e., including inter-kernel interactions) is essential. Analysis focusing on the individual compute stage, such as the perception stage only in most prior works, leading to suboptimal insights and results.

To improve the reliability of the MAV system, we further explore the error detection and recovery technique for MAV. Even though the redundancy-based hardware solutions (e.g., DMR, TMR) is widely used for reliability on autonomous vehicles, our experiments show the hardware redundancy leads to significant performance overhead for MAV (i.e., 1.9× flight energy increase), which is not tolerable for an energy-constrained MAV system. In addition, the DMR or TMR redundancy will increase the weight and form factor of MAV. Therefore, in this work, we propose two software-level low overhead anomaly error detection and recovery schemes, including Gaussian-based and autoencoder-based anomaly detection and corresponding recovery scheme. During the flight, a variety of inter-kernel "critical variables" from each PPC stage are monitored by the anomaly detector. Once the anomaly data is detected, the corrupted kernels will be recomputed to cease the error propagation.

We evaluate the Gaussian-based and autoencoder-based anomaly detection and recovery schemes' effectiveness across four types of environments on both Intel Xeon i9 CPU and ARM Cortex-A57 on the Nvidia TX2 platform. Our experimental results demonstrate that the Gaussian-based technique recovers up to 89.6% of failure cases, and autoencoder-based can recover all failures in the best-case scenario. In terms of quality-of-flight, the Gaussian-based technique can recover flight time by up to 63.5% and 73.0% for the autoencoder-based technique. Overall the autoencoder-based technique outperforms Gaussian-based in terms of both success rate and the quality-of-flight with less than 0.0062% overhead.

In summary, the contributions of this work are:

- A end-to-end reliability analysis framework (i.e., MAVFI tool) is proposed to analyze MAV applications' fault tolerance characteristics from the end-to-end compute viewpoint. MAVFI is portable to other ROS-based cyber-physical applications.
- Extensive fault tolerance characterizations of the MAV PPC pipeline have been performed. The results show that end-to-end fault analysis is essential to understand kernel vulnerability and fault's impact compared to the conventional isolated analysis approach.
- To improve the reliability of the MAV compute subsystem, two low-cost anomaly fault detection and recovery schemes are proposed and evaluated on different CPUs. The transient faults can be detected with high accuracy and recovered in real-time MAV applications with negligible overhead.

2. MAV BACKGROUND

We provide background on MAVs, including the software-hardware stack and safety requirements of MAVs. To explain the building blocks, we traverse down Fig. 2 which shows the abstraction layers of the MAV's computing stack. MAV's computing system can be generally divided into a system layer and a hardware layer.

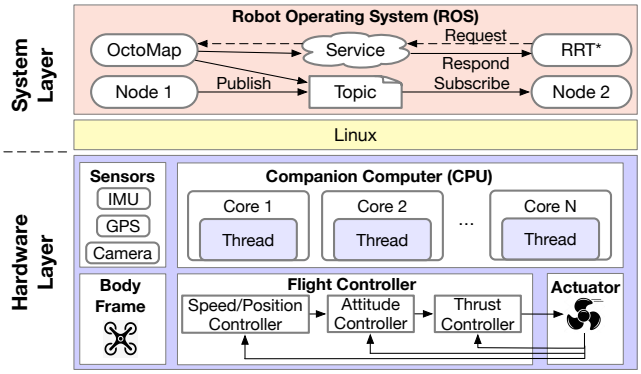


Figure 2: Computing Stack for MAV system.

2.1 MAV System

System Layer. The system layer includes both Robot Operating System (ROS) and Linux. ROS is the commonly used operating system to provides communication functions and resource allocation for robotics applications, such as MAV. ROS typically consists of multiple ROS nodes, ROS services, and a ROS master to support the functionalities and communications in the system [24]. Underneath ROS, Linux system maps workloads to compute units and schedules tasks at runtime. Each ROS node is treated as a process scheduled to a thread on CPU cores.

There is a significant advancement of high-level MAV algorithms and systems in perception, localization, mapping, and deep learning in recent years [25,26,27,28,29,30,31]. Among all autonomy paradigms, Perception-Planning-Control (PPC)

computational pipeline is widely used for various cyber-physical systems [32]. In the PPC pipeline, the perception stage takes the sensor data and creates three-dimensional models to provide a volumetric representation of space, such as a point cloud [33] and occupancy map [34]. The three-dimensional models are then fed into the planning stage to determine a collision-free trajectory by running a motion planner (e.g., RRT [35] and their variants [36]). Based on MAV's dynamics, the control stage follows the planned path through PID [37] or other controllers.

Hardware Layer. The hardware mainly consists of sensors, a companion computer, and a flight controller. The companion computer is used to execute the PPC kernels, which are the ROS nodes in the system layer. These kernels are usually running on a general-purpose processor (e.g., Intel i7/9 CPUs). Unlike an autonomous vehicle, MAVs are limited in computing resources and energy budget, and thus, it is less common to equip MAV with computationally intensive algorithms that require access to GPU or ASIC. The companion computer would generate high-level flight commands (e.g., velocity in x, y, z directions) in response to the sensor readings (e.g., RGB-D, GPS). The flight controller converts the high-level flight commands to low-level actuation commands to control and stabilize the drone. In this work, we consider the transient faults in the companion computer and not the flight controller as the former one runs the complex end-to-end applications and system layer tasks.

2.2 MAV Safety

MAV applications have stringent requirements for safe and high-quality flights and also unique fault protection challenges compared to other autonomous vehicles such as cars. First, it is challenging to deploy dedicated fault protection hardware within resource-constrained MAVs. In autonomous vehicles, the redundancy protection techniques, such as dual modular redundancy (DMR) [38], and triple modular redundancy (TMR) [39], are adopted to deploy redundant hardware to improve auto-driving reliability. However, it is not feasible to deploy such redundancy due to both the power and form factor limitations of a MAV system. Moreover, MAVs typically have strict real-time latency and power constraints due to the limited onboard battery capacity. Therefore, MAVs need to reduce flight time and energy as much as possible. Finally, unlike cars moving on a 2D occupancy grid, drones are flying within free 3D space. Therefore, motion planning for drones is computed in a higher-dimensional space. The additional calculation of control variables along the z-axis, such as the rotation in yaw, roll, and pitch dimensions, causes more complex and vulnerable computation than in vehicles.

3. MAV FAULT ANALYSIS FRAMEWORK

To analyze faults' impact on the MAV system in an end-to-end fashion, we developed MAVFI, an open-source fault injection tool for MAV applications. MAVFI supports both assembly-level and variable-level fault injection and collects the end-to-end system Quality-of-Flight (QoF) metrics, such as flight time and success rate. At assembly-level, MAVFI can introduce bit-flips at a source or destination register of any ROS node. At variable-level, MAVFI can introduce bit-flips at the cross-stage variable of the PPC pipeline.

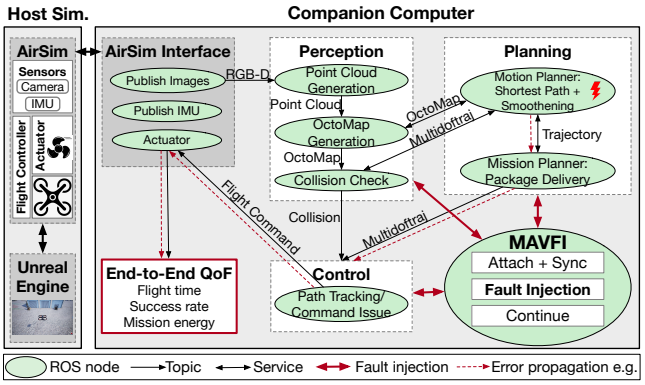


Figure 3: Overview of the interaction between the MAV's PPC pipeline and MAVFI fault injection node.

3.1 Overview of MAVFI

Fig. 3 illustrates the simulation infrastructure of the MAV system, including environment and drone simulation on the host simulator and the MAV's PPC pipeline integrated with MAVFI on the companion computer. Each ROS node comprises a single compute kernel, such as point cloud generation or motion planner. Each PPC stage contains one or multiple ROS nodes. ROS node communicates through ROS topics (one-to-many node communication) and/or ROS services (one-to-one node communication) as shown in the system layer of Fig. 2. To maintain our framework's portability, the MAVFI tool is built as a ROS node, which leverages the ROS communication protocol and Linux system call.

To establish end-to-end MAV experiments, we borrowed another open-source ROS-based MAV simulation infrastructure, i.e., MAVBench [32], which includes Unreal Engine to simulate the surrounding environment, AirSim simulator [40] to capture a MAV's dynamics and kinematics, and PPC computational pipeline to generate flight commands in real-time. The AirSim interface allows the PPC pipeline to access the sensor data (e.g., RGB-D and IMUs) and send back the flight commands to the flight controller in the AirSim simulator. The PPC pipeline processes the sensor data and generates flight commands continuously until the mission is complete. Finally, the end-to-end mission QoFs are recorded.

Fig. 3 also illustrates an error propagation example when a fault is injected at the *Motion Planner* kernel and manifests as a corruption of execution results (i.e., *Multidoftraj*, *Trajectory*), which eventually corrupts the flight command and impacts the overall end-to-end QoF. To the best of our knowledge, MAVFI is the first fault injection tool targeted at ROS-based MAV applications.

3.2 Details of MAVFI Fault Injection

MAVFI is supported on x86/Linux platforms. Fig. 4 shows the assembly-level fault injection details of MAVFI. Each oval node is a ROS node. The figure provides the detailed fault injection sequence using an example for ROS node 2. During the system initialization phase, the MAVFI node publishes its process ID (pID) to all the other nodes and subscribes to their pID. Thus, the MAVFI node can attach and manipulate the other ROS nodes in the system via the *ptrace* system call. The *ptrace* system call allows us to synchronize and manipulate the register file of processes in the system

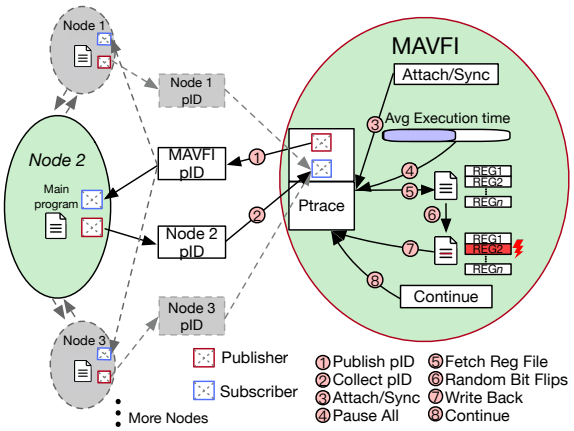


Figure 4: The design details for the interactions in MAVFI.

with much less overhead than the ROS communication protocol. MAVFI is the first fault injection framework built on top of both *ptrace* system call and ROS. During fault injection, the *ptrace* system call is leveraged to realize the node pause, fetching the register file, write back, and continue operations in sequence.

During the simulation of real-time MAV applications, MAVFI selects a random time point within the average execution time to pause all the nodes. All ROS nodes' execution will be stopped before conducting fault injection, ensuring every node following the original executive order. After all the nodes have stopped, the general-purpose and floating-point register files of the target node (i.e., node 2 in this case) are fetched via the *ptrace* system call, with the instruction pointer register decoded to access the current operating register. The number of registers being accessed by the current instruction ranges from 0 to 2. If the value is zero, MAVFI resumes all ROS nodes' execution and repeats the above steps to obtain a new instruction. For more than one register under operation, MAVFI would randomly choose one register to inject. The chosen register can be either a general-purpose register (i.e., *R8-R15*) or a floating-point register (i.e., *XMM0-XMM7*) in our x86 system. For the source register, according to the user-defined injection configuration, a single bit-flip or multiple bit-flips are introduced into the chosen register. For the destination register, before fault injection, MAVFI would step toward the next instruction to allow the current instruction to finish the writing operation, which avoids the corrupted destination register being overwritten by the current instruction. After fault injection, the corrupted register is written back to the target node's register files, and all nodes would be notified to resume the execution.

3.3 End-to-end (E2E) Fault Analysis

To evaluate the fault tolerance of different kernels in the MAV system, MAVFI can enable end-to-end fault tolerance analysis at assembly-level and variable-level.

E2E Fault Analysis at Assembly-level: The complete MAV application is built on top of the tightly coupled various PPC kernels, as shown in Fig. 3. To evaluate the error propagation through multiple kernels, MAVFI can inject faults into a register bit at assembly-level and obtain the final end-to-end flight metrics. In this way, each kernel's fault vulnerability can be evaluated through the impact on the entire end-to-end system metrics. Detailed analysis can be found in Section 4.2.

E2E Fault Analysis at Variable-level: To further study the vulnerabilities of PPC stages, MAVFI can introduce a single bit-flip into cross-stage variables at the variable-level. Similar to the end-to-end assembly-level fault tolerance analysis, we quantitatively evaluate the fault's end-to-end impact with the QoF metrics. As shown in Section 4.3, a notable vulnerability variation existed among variables, which motivates us to detect and recover transient fault based on the anomaly behavior of the cross-stage variables.

In Section 4, we analyze the end-to-end fault tolerance with fault injection at both assembly-level and variable-level for MAV-specific kernels, such as OctoMap, motion planner, and evaluate the system performance and safety impact considering error propagation. This end-to-end analysis provides insights into the most vulnerable kernels at the system level, which can guide the design of a more resilient system for MAV. As shown in Section 4.2, it is interestingly observed that the planning and control stages are more vulnerable than the perception stage.

3.4 Fault Model

MAVFI simulates transient faults (i.e., soft errors) that occur in the processor's register file, ALUs, and pipeline registers by introducing bit-flip at the source or destination register, which is known as assembly-level fault injection. MAVFI can inject either single or multiple bit-flips simultaneously. In the previous study [41], it was shown that single bit-flip yields a higher percentage of data corruption compared to multiple-bit errors. Therefore, for the analysis results in the paper, we mainly focus on single bit-flip, which also aligns with the previous fault tolerance analysis for autonomous vehicles [17]. As shown in Section 4, with single fault injection, the corrupted register may manifest as incorrect output values of computational kernels and impact the flight metrics of MAVs. MAVFI also supports bit-flips at the variable-level to further analyze the error propagation of the end-to-end MAV applications. We do not consider faults in the memories or caches as they are typically protected by error correction codes (ECCs). The fault model used in this work is in line with previous works in this area [42, 43, 44, 45, 46].

4. MAV FAULT TOLERANCE ANALYSIS

This section presents the fault tolerance analysis at different granularity levels, i.e., single-kernel level and end-to-end system level performance. Through the comparison, we observe that single-kernel analysis provides different or even opposite insights on the vulnerabilities of kernels than the end-to-end fault tolerance analysis. This is evidence that end-to-end fault analysis is crucial to capture transient fault's impact at the system level. For end-to-end analysis, we analyze the fault injection at both assembly-level and variable-level in Section 4.2 and Section 4.3, respectively.

4.1 Single-Kernel Fault Tolerance Analysis

To prove the importance of end-to-end fault analysis, we first conduct single-kernel fault injection and analysis at assembly-level. The single-kernel fault injection flow is similar to the prior fault injection tool [47]. We evaluate the commonly used kernels in the PPC pipeline, including *OctoMap*

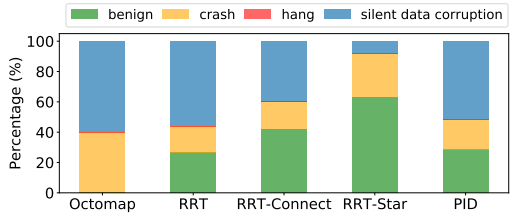
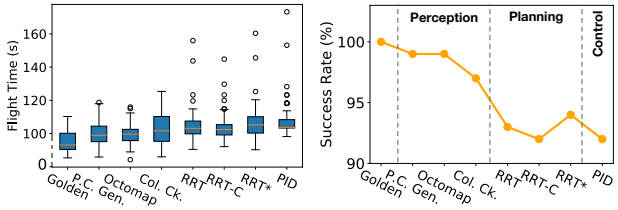


Figure 5: Execution outcomes of individual kernels with single-bit injections.



(a) Flight time. (b) Flight success rate.

Figure 6: The QoF metrics with fault injection at PPC kernels.

for the perception stage, three sampling-based motion planners (i.e., *RRT*, *RRTConnect*, *RRTstar*) for the planning stage, and Proportional-Integral-Derivative (PID) controller for the control stage. A single bit-flip is injected into a randomly picked source or destination register in each experiment run, and there are in total 5000 runs for each kernel. Each kernel is run without fault injection to obtain the error-free golden results. With fault injection, there are four types of outcomes: execution results different from the golden results (i.e. *silent data corruption* (SDC)), execution exceptions (i.e. *crash*), infinite execution time (i.e. *hang*), and execution results same as the golden results (i.e. *benign*) [48].

From the single-kernel perspective, the perception stage is most vulnerable to faults. As shown in Fig. 5, most compute kernels are more than 25% *benign* error-tolerant except for *OctoMap* at the perception stage. This is because transient faults can easily corrupt the output (*Octree*) with noisy values for the *OctoMap* kernel. Hence, *OctoMap* is less resilient than sampling-based planning and PID algorithms. On the other hand, the path planning kernels are all sampling-based algorithms, known for their high efficiency and performance. Injected errors should not affect output results as long as the corrupted way-point is not sampled. The more way-points are sampled, the higher probability the planning algorithms could sample a corrupted way-point, resulting in SDC. *RRT-Connect* runs two *RRT* algorithms from both start and goal, ending up with slightly fewer sampled way-points than *RRT*. *RRTstar* is the optimized version of *RRT* algorithm to find the shortest path by selecting even fewer way-points than *RRT* and *RRTConnect*, making *RRTstar* having the least SDC. The PID algorithm at the control stage also experiences around 25% *benign* cases since the PID has a simple self-healing mechanism to clip data points outside of a bounded range.

4.2 End-to-End Fault Tolerance Analysis

Compared to the single-kernel fault analysis, we further investigate and demonstrate that end-to-end analysis is needed based on our MAVFI framework. MAVFI is capable to evaluate faults' impact on end-to-end performance considering error propagation across kernels. We evaluate the MAV's end-to-end performance by QoF metrics, including flight time and success rate. Flight time is the total amount of time for the MAV to reach a given destination. Since rotors dominate mission energy, flight energy is proportional to flight time. Due to the space restrictions, we only show flight time in this section. Success rate is the percentage of successful cases over total experiment runs. We define a successful case as the drone reaches the destination without any collision, and a failure case as the drone collides into obstacles or fails to find a feasible path to the destination.

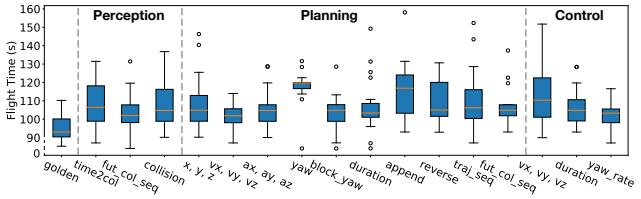
We introduce a single bit-flip at assembly-level in all PPC kernels. In our default settings, the PPC pipeline includes *Point cloud generation, OctoMap, Collision check* for perception, *RRT** for planning, and *PID* for control. Two other common planning algorithms are evaluated at the planning stage, i.e., *RRT* and *RRTConnect*. Each kernel has been experimented with 100 fault injection runs. Besides fault injection, 100 error-free experiment runs are defined as *Golden*. In each experiment, all kernels in the PPC pipeline would be launched by ROS to complete a given navigation task. In fault injection runs, only one of the kernels would have a one-time fault injection for each flight mission. Without loss of generality, we limit our discussion to a navigation task in the *Sparse* environment (Fig. 11c) here. More results on different environments can be found in Section 6.

From an end-to-end perspective, the perception stage is relatively resilient to faults. For the perception stage, both *Point Cloud Generation* and *OctoMap* have little impact on the overall QoF metrics as shown in Fig. 6. However, according to Fig. 5, *OctoMap* has the highest percentage of SDC among the evaluated kernels, which is regarded as the most vulnerable kernel in the single-kernel analysis. The reason why *OctoMap* is the most reliable kernel in the end-to-end analysis is that even if an occupied voxel is corrupted and mistaken as a free voxel, all other voxels around it are still occupied. Therefore, the MAV can still determine obstacles' locations provided the *OctoMap*'s resolution is sufficient. This counter-intuitive insight is difficult to discover without end-to-end analysis. *Collision Check* kernel is the most critical one in the perception stage since a false alarm can lead to trajectory re-planning or even collision.

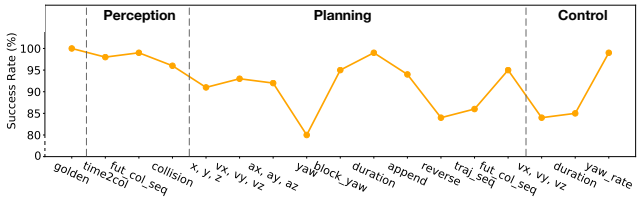
From an end-to-end perspective, planning, and control are more vulnerable to faults. While the SDC percentages of planning and control kernels are lower than *OctoMap*, corrupted outputs (e.g., yaw, velocity) from these two stages can directly lead to a detour or crash of the MAV. From Fig. 7a, even though the average flight time is similar, the range of *RRT, RRTConnect, RRT*, and PID* is much wider than *Octomap* and *Golden*. The error propagation of the corrupted execution results could greatly increase the flight time by up to 57.3% and even lead to degradation of success rate by up to 8% as shown in Fig. 6b. Therefore, the planning and control stages are more vulnerable than the perception stage from an end-to-end perspective, which is also counter-intuitive from the single-kernel analysis shown in Section 4.1.

4.3 End-to-End Fault Tolerance Analysis at the Variable-level

We further analyze end-to-end fault tolerance at variable-level by introducing bit-flip at cross-stage variables to explore

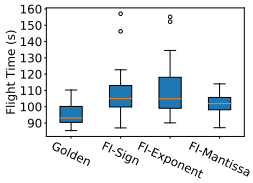


(a) Flight time.

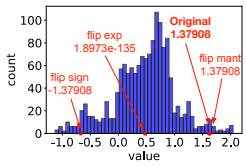


(b) Flight success rate.

Figure 7: Flight time and success rate with single-bit injections at the cross-stage variables.



(a) Flight time.



(b) Bit-flip analysis.

Figure 8: The impact of fault injection at different data fields on flight time and value histogram.

Table 2: MAV's QoF metrics with single- and multiple-bit injections.

Bit Flips	Flight Time (s)	Flight Distance (m)	Total Energy (kJ)	Number of Re-plans	Success Rate (%)
0 (golden)	94.6	49.5	51.4	3.71	100
1	105.6	55.9	57.2	4.07	92
3	107.3	56.8	58.7	4.19	91
5	111.8	59.4	61.1	4.28	89

the vulnerable cross-stage variables. A better understanding of vulnerable variables provides insights to improve the PPC kernels and facilitate anomaly fault detection in Section 5. Similar to Section 4.2, 100 navigation tasks are run for each variable with a single-bit injection. The summary of QoF metrics is shown in Fig. 7.

The functionality and position of the variable impact its fault tolerance. Variables have different resilience to injected bit-flips. For example, in the perception stage, *future_collision_seq* is much more robust than *time_to_collision*, whose QoF metrics noticeably vary when compared to the golden run. The functionality and position of these variables influence their fault tolerance and impact on MAV performance. Faults in *time_to_collision* can skew the drone's perceived distance to obstacles. Similarly, data corruption of *(x,y,z)* and *yaw* of way-points planned by motion planner can lead to an incorrect direction or crash into obstacles, and faults in *(vx, vy, vz)* could make the drone fail to keep track of a trajectory. As a result, the distorted trajectory lead to collision or an increase in flight time and mission energy.

Sign bits and exponent fields are more vulnerable to faults than mantissa fields. Single bit-flip in different data fields of a variable has a distinct level of impact on MAV performance. We conduct 100 fault injection experiments at the float64 variable *(x, y, z)*, which contains 1 sign bit, 11 exponent bits, and 52 mantissa bits. Faults in sign and exponent fields have a greater impact on drone's reliability and result in increased flight time, energy, and failure cases, as shown in Fig. 8a. Faults in the sign and exponent will result in a greater change in the variable's value compared to faults in the mantissa field. For example, when a single bit-flip at the exponent and sign, a value is corrupted from 1.38 to 0 and -1.38, respectively, as illustrated in Fig. 8b.

As MAVFI supports both single-bit and multiple-bit injection, we also evaluate the end-to-end performance impact with multiple bit-flips, as shown in Tab. 2. In this experiment, 100 fault injections are performed for 1-, 3-, 5-bits at *(ax, ay, az)*, which are the output variables of the planning stage. By injecting faults from 1-bit to 5-bits, the average flight time and energy increase by 6.2s and 3.9kJ, respectively, and the success rate decreases by 3%. Since more bit-flips are more likely to affect the sign and exponent fields, the value changes could be more dramatic for multiple-bit injection.

5. ANOMALY DETECTION AND RECOVERY

To increase the reliability of the MAV system, we further explore the detection and recovery technique based on the observations from MAVFI. As the conventional redundancy-based hardware protection (e.g. DMR) introduces significant overhead, we propose two software-level low-overhead fault anomaly detection and recovery schemes for MAV. The proposed schemes detect anomalous behavior of SDC in the PPC and cease the error propagation before sending the corrupted flight commands to the MAV, thus ensuring MAV's safety.

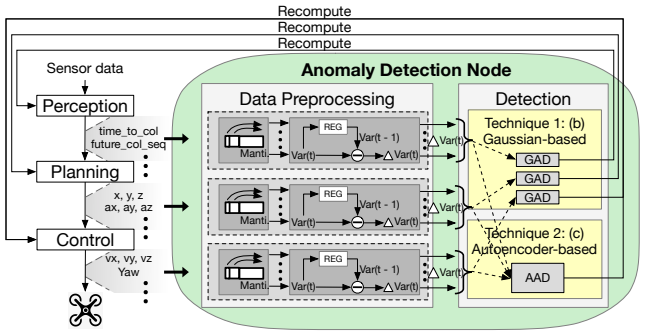
5.1 Overview of Detection and Recovery

Anomaly detection techniques have already been adopted in several application domains, including fraud detection for online transactions [49] and anomaly detection in safety-critical systems [50]. There are several anomaly detection techniques [51], including clustering-based classifier, Gaussian-based classifier, neural network-based classifier, and autoencoder, as shown in Tab. 3. Among all the techniques, the conventional clustering-based and Gaussian-based techniques do not need a training process but can only achieve moderate detection accuracy. The neural network-based classifier can achieve better detection accuracy, while it is supervised learning, which requires a large amount of labeled training data. Furthermore, the inherently biased dataset toward normal data makes supervised learning hard to replicate its high accuracy in anomaly detection. The recent autoencoder neural network uses an unsupervised learning mechanism, which does not require labeled data for training, and achieves decent detection accuracy as a traditional neural network classifier [52]. We implemented both Gaussian-based and autoencoder-based anomaly detection techniques for transient fault detection in this work. It is observed that both techniques can achieve efficient fault detection and flight time reduction. At the same time, the autoencoder-based scheme obtained better performance with lower computational overhead.

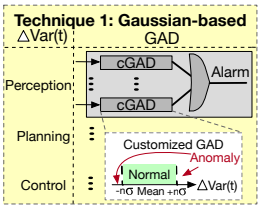
Fig. 9a shows the proposed anomaly detection and recovery scheme integrated with the PPC pipeline, which forms a closed-loop system. According to the analysis in Section 4.3,

Table 3: Properties of anomaly detection techniques.

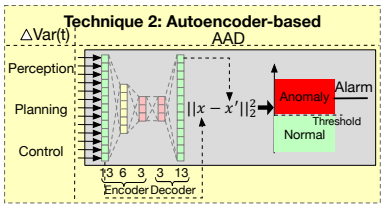
Techniques	Training required	Initialization dataset	Computation overhead	Detection effectiveness
Autoencoder (This work)	Unsupervised	Large	Large	Good
Neural network based	Supervised	Large	Large	Good
Clustering based	No	Moderate	Moderate	Moderate
Gaussian model (This work)	No	Moderate	Low	Moderate



(a) Overview.



(b) Gaussian-based.



(c) Autoencoder-based

Figure 9: The proposed anomaly detection and recovery scheme for MAV computational pipeline.

the critical cross-stage variables as shown in Fig. 7 are monitored for the anomalous SDC. The monitored variables pass through a data preprocessing module to increase the detection performance while further reducing the computational overhead. After data preprocessing, the processed variables go into either of the proposed anomaly detection techniques. Once an anomalous behavior, i.e., data corruption, is detected, an alarm signal will be raised, triggering the recomputation of the corresponding stage, prevent the corrupted execution results from propagating to the following stage. The proposed closed-loop detection and recovery system can greatly increase the reliability of MAV's PPC pipeline against SDC that degrades the safety and flight performance of MAV. The effectiveness and overhead of these two detection techniques are extensively experimented with and compared in Section 6. Our approach focuses on SDC as *crash* can be detected by the ROS system. The ROS master node would restart the node automatically if it crashes [53].

5.2 Data Preprocessing

In Fig. 9a, the monitored cross-stage variables from the PPC pipeline are processed in the data preprocessing block before sent to the anomaly detection block. There are two steps in data preprocessing, including data format transformation and delta calculation. First, for data format transformation, the sign and exponent bits of *float64* variables are transformed into 16-bits integer variables, reducing double-precision floating-point operations into 16-bits integer operations. Since transient faults at the mantissa bits of *float64* are insignificant as shown in Section 4.3, only the sign and exponent bits are monitored to reduce the overhead of detection. Second, the deltas of the incoming variables are calculated. We define delta as the number of value changes from the previous time point to the current time point for a variable. Fig. 10 shows the insight of using the variable's delta for anomaly detection. For most variables, the value could have either uniform distribution or Gaussian distribution. For instance, the *multi_x* variable of a way-point indicates the local goal in the x-axis and has a uniform distribution as shown in Fig. 10a. However, the uniform value distribution is not well suited to Gaussian-based anomaly detection, leading to very low detection accuracy. By leveraging the variables' continuous property, the uniform distribution can be transformed into Gaussian distribution by calculating the variables' delta. Furthermore, the range of the variable's delta is much smaller than the original data. For instance, as shown in Fig. 10, range of *multi_x*, *multi_vx*, and *multi_ax* variables are reduced by 98%, 94%, and 76%, respectively, making the differences between normal and anomaly data even more. Thus, the data preprocessing can increase the performance of the anomaly detection while decreasing the overhead of detection.

5.3 Anomaly Detection

5.3.1 *Gaussian-based Anomaly Detection*

Fig. 9b shows the design details of the Gaussian-based Anomaly Detection (GAD). Each PPC stage has a corresponding GAD that consists of several customized GAD (cGAD) for each variable. If the value of an incoming variable is outside the range of its normal data distribution, its cGAD will send out an alarm. The alarms from each cGAD are gathered for each of the PPC stages, respectively. An alarm from a GAD would trigger the recomputation path of its corresponding stage, stopping the error propagation to the following stage.

The Gaussian model parameters (i.e., mean, standard deviation) for each cGAD are estimated as following equations:

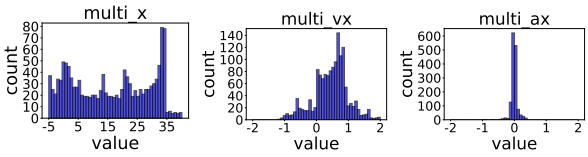
$$M_k = M_{k-1} + (x_k - M_{k-1})/k \quad (1)$$

$$S_k = S_{k-1} + (x_k - M_{k-1})(x_k - M_k) \quad (2)$$

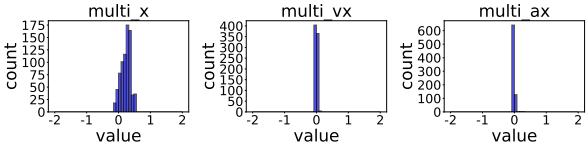
where k is the number of samples, M_k is the mean value for the k samples, and S_k is an auxiliary term used in the computation of standard deviation σ. At initialization, we introduce and set the terms $M_1 = x_1, S_1 = 0$. The model parameters are updated online with the recurrence formulas above for a new incoming data x_k [54]. For $k \geq 2$, the standard deviation σ can be derived by $\sigma = \sqrt{S_k/(k-1)}$. Whenever the value of the incoming data is n sigma away from the mean value, the alarm of the cGAD will be raised. The number of sigma n is a configurable variable that can be optimized based on the complexity of the flight task and environment. To ensure the Gaussian models have sufficient samples before starting anomaly detection, we first have the model updated with error-free training environments as presented in Section 6.1.

5.3.2 *Autoencoder-based Anomaly Detection*

Fig. 9c shows the Autoencoder-based Anomaly Detection (AAD). The AAD block collects the processed variables from all PPC stages as input. An alarm will be raised and triggers



(a) Histogram of the variables' value.



(b) Histogram of the variables' delta.

Figure 10: Histogram comparison between the variables' value and delta after data processing.

the recomputation of the control stage if an anomaly is detected. The proposed autoencoder comprises an encoder with three fully connected layers and a decoder with two fully connected layers. The encoder has an input layer of 13 neurons, a hidden layer of 6 neurons, and an output layer of 3 neurons. The decoder has an input layer of 3 neurons, which takes the compressed data from the encoder, and an output layer of 13 neurons. The output of the decoder represents the reconstructed input data. The reconstruction error is the difference between the input and output of the autoencoder. We use the mean squared error during the unsupervised training as the reconstruction error minimized by the Adam optimizer [55]. At the inference phase, if the reconstruction error is beyond the threshold, the alarm will be raised. The threshold is the upper bound of the reconstruction error of the error-free golden run.

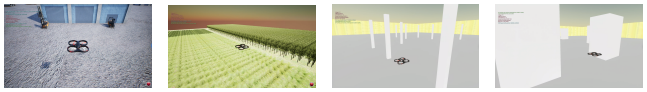
Rather than the separate Gaussian-based detection module for each PPC stage, we use a single autoencoder for the whole PPC pipeline to leverage the correlation among stages. Once an anomaly is detected, the alarm triggers the recomputation of the control stage. In this way, the autoencoder scheme achieves higher detection performance while dramatically reducing the recomputation overhead, as shown in Tab. 5.

5.4 Recovery Scheme

Once a data anomalous behavior has been detected, the recomputation path would be triggered to cease the error propagation. The compute stage which is notified to recompute will fetch the latest data from the previous compute stage or sensor (Airsim simulator) and re-generate the results.

Take the navigation task as an example. If an anomaly alarm is raised in the perception stage, the stage starts to recompute and fetch the current RGB-D camera data from the AirSim simulator. Then, *Point Cloud Generation*, *Octomap*, and *Collision Check* kernels process the data and generate results for the following stage. Similarly, if an alarm is raised in the planning stage, the planning algorithm will fetch the latest occupancy map from the perception stage and plans a new trajectory. Finally, the flight command is monitored for the control stage before being sent back to the AirSim simulator. If an alarm is raised, the control stage will abandon the current anomalous waypoint and fetch the next waypoint of the trajectory, generating correct flight commands.

5.5 Implementation of Anomaly Detection and Recovery on ROS Layer



(a) UE Factory. (b) UE Farm. (c) Sparse. (d) Dense.

Figure 11: The environments for evaluation.

In our experiment, the anomaly detection and recovery scheme is built as a ROS node, referred to as the detection node in the following, which continuously monitors the cross-stage variables to guarantee the safety of MAV's compute subsystem. The detection node contains the data preprocessing and anomaly detection functions as explained in Section 5.2 and Section 5.3, respectively. The detection node subscribes to the topics containing the cross-stage variables from each PPC stage as input and publishes recomputation signals to the corresponding stages if an alarm is raised by the detection function. The detection node can thus continuously supervise the PPC pipeline, increasing the reliability of MAV's computational pipeline with negligible overhead.

6. EXPERIMENTAL EVALUATION

This section demonstrates the effectiveness of the anomaly detection and recovery schemes across different environments. First, we demonstrate the schemes can increase the mission success rate and QoF metrics with trajectories analysis. Next, we experiment with more efficient fault detection and recovery with only recovering particular stages. Finally, we measure and compare the compute overhead of two anomaly detection and recovery schemes.

6.1 Experiment Setup

Hardware-in-the-loop Simulator. To demonstrate the reliability and flight performance benefits of the anomaly detection and recovery schemes, we used a closed-loop MAV simulator, MAVBench, as the experimental platform [32]. The environments are simulated in Unreal Engine (UE), and the MAV's kinematics and sensors are captured by AirSim, which also supports various flight controllers. Sensors, including RGB-D camera and IMU, used in the experiments, are common for MAVs. An Intel i9-9940X CPU and an Nvidia GTX 2080 Ti GPU are used as the host machine to simulate environments and the drone. The companion computer is equipped with an Intel i9-9940X CPU, which takes sensory data from the environment and generates flight commands for the drone.

Training Environments. To create a training dataset for the autoencoder-based technique, we built an environment generator with configurable parameters (i.e., obstacle density and size of obstacle). The obstacle density is defined as the probability of a 10 * 10 grid spawned with an obstacle. Each obstacle is a cuboid with $n * n$ and infinite height at the center of a grid, where n is a discrete number from 1 to 10. [*obstacle density, size of obstacles*] is defined as an environment configuration pair. We collect data from randomized environments with the combinations of two obstacle densities (i.e., 0.05 and 0.2) and two sizes of obstacles (i.e., 3 and 5). Therefore, there are four configuration pairs in total and each is run 25 times. For a given configuration, a random seed is used to randomize the environment in each experiment run. For Gaussian-based technique, the Gaussian models are also

Table 4: The flight success rate in 4 evaluation environments.

Environment	Factory	Farm	Sparse	Dense
Golden Run	100.0%	100.0%	95.0%	85.0%
Injection Run	91.7%	97.3%	88.3%	75.3%
Gaussian-based	98.7%	99.3%	94.3%	83.0%
Autoencoder-based	99.3%	100.0%	95.0%	84.7%

updated with the same error-free training environments.

Evaluation Environments. The anomaly detection and recovery schemes are evaluated in four environments. The *Factory* (Fig. 11a) and *Farm* (Fig. 11b) are provided by UE, representing common navigation scenarios with blocks, walls, and hedges. We generate the *Sparse* (Fig. 11c) with [*0.05, 3*] and the *Dense* (Fig. 11d) with [*0.2, 5*] using our environment generator. The random seed is fixed during evaluation.

Overheads. The QoF metrics do not include the fault injection time since the ROS nodes are paused during fault injection. In terms of simulation time, for one-time fault injection, MAVFI only takes less than 5 milliseconds, which is negligible for a typical flight mission that takes more than 100 seconds. For the anomaly detection and recovery experiments, we quantify the overhead for Gaussian-based and autoencoder-based techniques in Section 6.5.

6.2 Detection and Recovery Effectiveness

To evaluate the anomaly detection and recovery scheme, we run 100 error-free simulations for each environment as the baseline (golden run). Then, we conduct 900 single-bit injections at assembly-level for each environment, including 300 runs for each setting (i.e., *FI*, *D&R(G)*, and *D&R(A)*) as shown in Fig. 12, respectively. In each setting, we have 100 fault injections for each PPC stage. The number of total 1000 experiments is chosen considering each experiment run takes about 5 minutes. Even though MAVFI introduces a negligible overhead of only 5 milliseconds, the experiment time is a limiting factor for the total run number.

Improvement of success rate. Tab. 4 shows the success rates of MAV flight in the golden run, injection run, Gaussian- and autoencoder-based scheme across four environments. In the fault injection runs, the success rate is reduced by up to 9.7% in the complex *Dense* environment. Faults may easily cause more collision or fail to find a collision-free path in complex environments. By contrast, *Farm* is an obstacles-free environment. Even a drone detours from its path, there are more feasible paths toward the destination than a complex environment. With the anomaly detection and recovery scheme, Gaussian- and autoencoder-based techniques recover up to 89.6% and 100% (fully recover) of failure cases, respectively. Generally, the autoencoder-based scheme recovers more failure cases caused by transient faults than the Gaussian-based scheme and increases the success rate close to or same as the error-free golden runs in all four environments.

Improvement of QoF metrics. Fig. 12 demonstrates the flight time of all successful cases in Tab. 4 across four environments. Similar to Section 4.2, the fault injection runs results in a much wider range of flight time than the golden run and increase the flight time by 73.8%, 74.2%, 62.6%, 93.3%, and mission energy by 74.9%, 73.1%, 62.8%, 94.2% in the worst case for each environment, respectively. With Gaussian-based anomaly detection and recovery, many of the outliers can be recovered and the worst-case flight time is recovered by 56.4%, 63.5%, 49.0%, 58.7%, and mission energy by 55.8%, 63.3%, 48.6%, 59.9%. On the other hand, the autoencoder-based technique recovers most of the outliers and can recover the worse-case flight time by 64.2%, 68.4%, 57.8%, 73.0%, and mission energy by 64.7%, 67.4%, 56.5%, 73.9%, outperforming Gaussian-based technique.

Comparison of Gaussian-based and autoencoder-based schemes. The autoencoder-based technique consistently outperforms the Gaussian-based technique in all four environments in terms of both success rate and QoF metrics. The reason is that the autoencoder can leverage the correlation among different variables, and thus it can detect the subtle discrepancy among variables. On the contrary, the Gaussian-based technique does not have the correlation information among variables and can only detect each variable separately, which may fail to detect anomalies if the corrupted data is still inside the range of the normal data distribution.

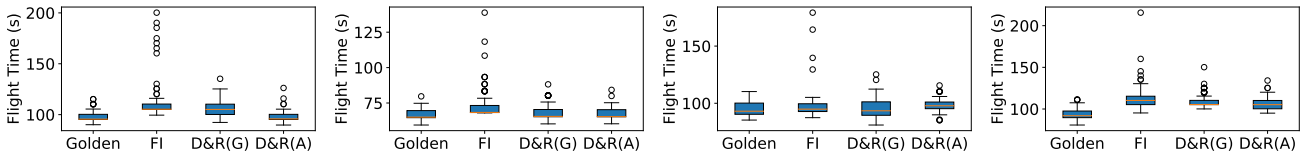
Comparison of environments. More challenging environment with a higher density of obstacles is also more difficult for the anomaly detection and recovery schemes to recover from errors. For the *Dense* environment, a MAV has more complex trajectories to follow and more dynamic actions in response to the obstacles, making the range of the variable distribution wider. The wider distribution increases the number of false-negative detection. Thus, there is still a 20.1% gap between autoencoder-based recovery results and golden for the worse case. On the other hand, for the obstacle-free *Farm* environment or *Sparse*, the autoencoder-based technique can achieve a similar performance as the golden run.

6.3 Trajectory Analysis

To demonstrate the impact of faults and the effectiveness of the anomaly detection and recovery schemes, we visualize MAV's trajectories in the *Dense* environment as shown in Fig. 13. We present the trajectories with the autoencoder-based technique, while the Gaussian-based technique has similar results in a successful detection.

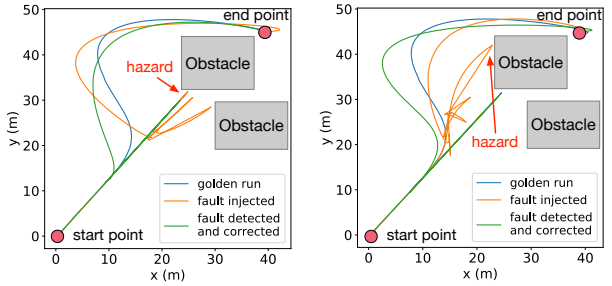
Fig. 13 shows the scenario that a single-bit injection in the PPC stage can lead to flight detour and how the detection and recovery scheme improves the flight. Without fault injection (blue curve), the drone takes off at the start point and flies towards the endpoint in the beginning phase. When facing an obstacle, it stops at a safe distance and re-plans a new trajectory that flies back slightly and bypasses the obstacle. When transient faults get injected and corrupt the critical variable, such as the coordinate of a way-point, the path may be distorted, making the drone flying in a wrong direction. The drone may not stop until it faces an obstacle (orange curve), which causes the drone to fly back or re-plan its trajectory. The more often the drone re-plan and detour from its path, the longer it takes to reach the destination, which increases the flight time by 21.9% and 24.5% for Fig. 13a and Fig. 13b, respectively. With the detection scheme, once the anomaly has been detected, the corrupted way-point is abandoned. The alarm raised by the detection module triggers the recomputation of the corresponding stage. Therefore, the drone would follow a better trajectory (green curve) without detour, which recovers the QoF metrics.

6.4 Anomaly Detection and Recovery for Different Compute Stages



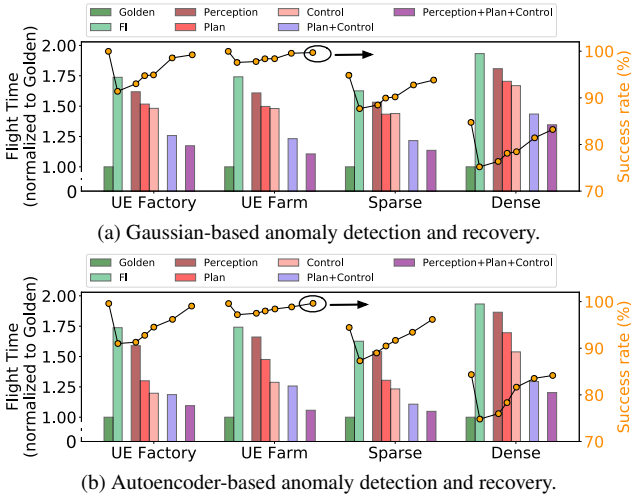
(a) UE Factory. (b) UE Farm. (c) Sparse. (d) Dense.

Figure 12: The effectiveness of the proposed anomaly detection and recovery schemes in terms of flight time. D&R(G) and D&R(A) represent the Gaussian-based and autoencoder-based schemes, respectively.



(a) Fault injection in perception. (b) Fault injection in planning.

Figure 13: Trajectories of golden run, with fault injection, with both fault injection and fault detection and recovery.



(a) Gaussian-based anomaly detection and recovery.

(b) Autoencoder-based anomaly detection and recovery.

Figure 14: Worst-case QoF metrics with different fault detection and recovery stages (normalized to golden run).

To evaluate the effectiveness of error detection in different PPC stages, we further experiment with the anomaly detection and recovery scheme for certain compute stages in the MAV compute pipeline, as shown in Fig. 14.

Single-stage detection and recovery. We first experiment the anomaly detection and recovery by only detecting a single pipeline stage. As shown in Fig. 14, with only detect and recover anomaly for a single stage, the Gaussian-based technique recovers the flight time by 16.2%, 29.9%, 34.7% and the autoencoder-based technique recovers the flight time by 20.1%, 59.3%, 73.2% for perception, planning, and control, respectively, along with the success rate improvement, in *Factory* environment. A similar trend has been demonstrated in the other three environments. Both techniques show that the flight time can be recovered the most by detecting the faults

Table 5: Compute time overhead of anomaly detection and recovery scheme for each stage.

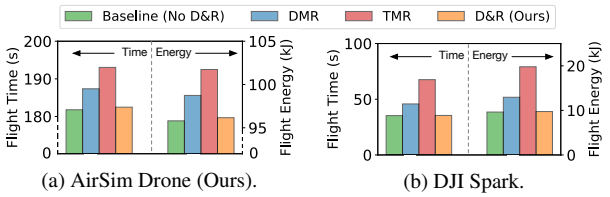
Environment	Factory		Farm		Sparse		Dense	
	DET	RECOV	DET	RECOV	DET	RECOV	DET	RECOV
Perception	<0.0001%	0.9603%	<0.0001%	1.0902%	<0.0001%	0.9788%	<0.0001%	1.1932%
Planning	<0.0001%	1.0199%	<0.0001%	0.7801%	<0.0001%	0.9421%	<0.0001%	1.0279%
Control	0.0008%	<0.0001%	0.0007%	<0.0001%	0.0009%	<0.0001%	0.0012%	<0.0001%
sum (Gaussian)	1.9810%		1.8710%		1.9218%		2.2223%	
PPC	0.0042%	<0.0001%	0.0037%	<0.0001%	0.0047%	<0.0001%	0.0062%	<0.0001%
sum (AutoE)	0.0042%		0.0037%		0.0047%		0.0062%	

that happened in the control stage. The reasons are twofold. First, the planning and control stages are more vulnerable to transient faults from an end-to-end perspective. Second, the control stage is the last stage in the PPC pipeline, and thus any error propagated from previous stages has to pass through the control stage before actually corrupting the flight command. The vulnerability analysis of the individual stage also lines up with the analysis in Section 4.2.

Multi-stages detection and recovery. To understand how different pipeline stages affect the anomaly detection and recovery schemes, we apply the scheme on multi-stages, namely the planning-and-control (PC) stage and all PPC stages. The Gaussian-based approach recovers the flight time by 65.1%, 76.5%, and the autoencoder-based recovers the flight time by 74.8%, 87.1% for PC and PPC, respectively, along with the increase in success rate, in *Factory* environment. For the Gaussian-based technique, detecting the PC stage significantly outperforms the single-stage detection and recovery in all environments. For the autoencoder-based technique, detecting PC stage achieve slightly better performance than only detecting control in *Factory*, *Farm*, and *Sparse* environment. However, in *Dense* environment, detecting the PC stage with the autoencoder-based scheme greatly outperforms detecting the control stage by 47.4%. Results from both techniques show that a MAV achieves similar or higher performance by monitoring more stages, and the performance benefit is more for complex environments.

6.5 Compute Overhead

Software-level protection. We first evaluate the overhead of the proposed software-level anomaly detection and recovery scheme in this work across the evaluation environments. The detection and recovery overhead is the total amount of detection and recomputation time for each mission, respectively. Tab. 5 shows that the overall overhead of the autoencoder is much smaller than the Gaussian-based technique. The overhead of the Gaussian-based technique is dominated by the recovery of perception and planning stages, which takes around 289 *ms* for each occupancy map generation and 83 *ms* for each trajectory generation. On the other hand, even if the autoencoder-based technique's detection overhead is higher, the recovery overhead is negligible since the recomputation



(a) AirSim Drone (Ours). (b) DJI Spark.

Figure 15: Comparison of redundancy-based schemes (DMR and TMR) and the proposed anomaly detection and recovery schemes on ARM Cortex-A57.

of the control stage only takes 0.46 *ms*. As our proposed fault detection and recovery is operated at software-level with negligible overhead, it is possible to deploy multiple anomaly detection nodes to improve the robustness of detection nodes, which can be the future work.

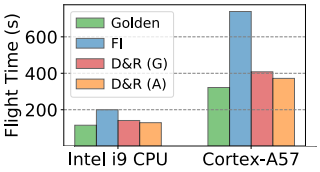
Hardware-level protection. To demonstrate the benefits of our proposed software-level schemes over conventional redundancy-based hardware protections, we adopt a MAV visual performance model from [7] to evaluate the performance overhead of microarchitecture-based redundancy schemes (DMR and TMR) on MAV. In the comparison, two types of drones, AirSim drone (used in this work) and DJI Spark (with the same specs as [7]), are used as experimental platforms. Fig. 15 shows that TMR incurs a flight time increase by 1.06× on AirSim drone and 1.91× on DJI compared to anomaly detection scheme. The rationale is that hardware redundancy brings higher compute power with higher thermal design power and weight, thus lowering flight velocity and increasing flight time. Given the tight resource constraints for the MAV system, our software-level anomaly detection and recovery scheme demonstrates negligible end-to-end performance overhead.

6.6 Computing Platform Comparison

To show the portability of our proposed schemes, we conduct variable-level fault injection on both Intel i9 CPU and ARM Cortex-A57 CPU on the Nvidia TX2 platform, by introducing a single bit-flip at the cross-stage variables as in Section 4.3. As the results shown in Fig. 16, we observe a similar error detection and recovery trend for both platforms. On the TX2, the worst flight time increases 2.8× since the TX2 is an edge platform that has longer latency and slower responses to environmental changes. With the dedicated anomaly detection ROS node continuously monitors the anomaly of cross-stage variables, the flight time has been successfully recovered by 79.3% and 88.0% with Gaussian-based and autoencoder-based techniques, respectively. In this experiment, we chose the variable-level fault injection as TX2 CPU uses ARM instruction set architecture, which needs a different assembly-level injection than x86 Intel CPU.

7. RELATED WORK

Fault injection. As shown in Table 1, the most prior fault injection works perform isolated kernel analysis at application-level [10, 56, 57, 58, 59], micro-architecture level [60, 61, 62], or RTL-level [63, 64] to understand the fault tolerance characteristics. [65] indicated that lower-level fault injection is more accurate than the application-level approach.



	i9-9940X	Cortex-A57
Core Number	14	4
Core Freq. (GHz)	3.3	2
Power (Watt)	165	<15
Flight time (s)	115	322
Flight energy (kJ)	61.7	177.1

Figure 16: Comparison of anomaly detection and recovery schemes on Intel i9 CPU and ARM Cortex-A57.

However, to conduct lower-level fault injection, micro-architecture level detail of the evaluated edge device is needed. Furthermore, for the end-to-end simulation of MAV, even the simple *factory* environment takes 5 minutes per run and needs thousands of runs in total. Consequently, it is more practical for sophisticated applications (i.e., navigation task for MAV) to adopt system-level fault injection methodology as the proposed MAVFI than the low-level approach.

Safety standards. Many efforts have been dedicated for autonomous vehicle safety from both academia and industry [66, 67, 68]. For example, the safety standard ISO 26262 [69] has been developed to provide comprehensive guidance and safety requirements for vehicle and their systems, e.g., safety goal under hazardous events [70]. There are also online safety protection hardware systems developed for vehicles, such as the NXP FS4500 system for functional safety measurement. A variety of fault tolerance analysis has been performed at the software level for environment noises [8], sensor noises [71], and faults/vulnerabilities in the computing system for autonomous vehicle [16, 17, 72, 73, 74]. Unfortunately, to date, there are no comprehensive standards for *autonomous* UAV assessment. The most related is ISO 21384 [75], which is released in December 2019 to cover the safety and security requirements for standard UAVs. The ISO committee is still working on more complementary standards regarding safety, quality, and terminology. However, with the growing number of autonomous drones [76], there is a strong demand for assessing autonomous drone reliability.

System recovery. Besides redundancy system [77, 78, 79], there are recovery schemes based on checkpoint and idempotency. The former [80, 81, 82, 83, 84] set checkpoints at a different level of granularity and restart from the checkpoint when an error occurred. Due to the high overhead of checkpoint-based recovery, [85, 86, 87] leveraged the idempotency property to alleviate the checkpoint overhead. A program is idempotent if its multiple executions lead to the same results. Therefore, recomputation can recover from the errors for idempotent programs. Asymmetric Resilience [88] further proposed task-level idempotency for accelerators, which is more vulnerable to transient error than CPU and alleviates accelerator from expensive resiliency optimization.

8. CONCLUSION

Practical reliability considerations for MAV systems require a better understanding of fault tolerance in complex end-to-end computation pipelines. We built an open-source fault analysis framework, MAVFI, to enable comprehensive end-to-end fault tolerance analysis. Compared to conventional single kernel fault injection analysis, our work shows that end-to-end analysis is essential to capturing the kernel vulnerability. Beyond MAVFI, we further propose Gaussian-

based and autoencoder-based anomaly detection and recovery schemes with low compute overhead. Experimental results demonstrate that with less than 0.0062% compute overhead, the autoencoder-based scheme can fully recover from all failure cases in the best-case scenario with negligible overhead.

Acknowledgements

We thank Abdulrahman Mahmoud (from Harvard) for his constructive feedback on improving the paper. This work was sponsored in part by the ADA (Applications Driving Architectures) Center and C-BRIC (Center for Brain-inspired Computing) , two of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

REFERENCES

[1] A. Goodchild and J. Toy, "Delivery by drone: An evaluation of unmanned aerial vehicle technology in reducing co2 emissions in the delivery service industry," *Transportation Research Part D: Transport and Environment*, vol. 61, pp. 58–67, 2018.

[2] S. Waharte and N. Trigoni, "Supporting search and rescue operations with uavs," in *2010 International Conference on Emerging Security Technologies*, pp. 142–147, IEEE, 2010.

[3] D. R. C. McCullough, "Unmanned Aircraft Systems(UAS) Guidebook in Development." https://cops.usdoj.gov/html/dispatch/08-2014/UAS_Guidebook_in_Development.asp.

[4] R. Feltman, "The Future of Sports Photography: Drones." https://www.theatlantic.com/technology/archive/2014/02/the-future-of-sports-photography-drones/283896/.

[5] Z. Wan, B. Yu, T. Y. Li, J. Tang, Y. Zhu, Y. Wang, A. Raychowdhury, and S. Liu, "A survey of fpga-based robotic computing," *arXiv preprint arXiv:2009.06034*, 2020.

[6] S. Krishnan, Z. Wan, K. Bhardwaj, P. Whatmough, A. Faust, G.-Y. Wei, D. Brooks, and V. J. Reddi, "The sky is not the limit: A visual performance model for cyber-physical co-design in autonomous machines," *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 38–42, 2020.

[7] S. Krishnan, Z. Wan, K. Bharadwaj, P. Whatmough, A. Faust, S. Neuman, G.-Y. Wei, D. Brooks, and V. J. Reddi, "Machine learning-based automated design space exploration for autonomous aerial robots," *arXiv preprint arXiv:2102.02988*, 2021.

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

[9] Y. Cao, C. Xiao, D. Yang, J. Fang, R. Yang, M. Liu, and B. Li, "Adversarial objects against lidar-based autonomous driving systems," *arXiv preprint arXiv:1907.05418*, 2019.

[10] 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*, pp. 243–247, IEEE, 2005.

[11] R. Bertran, A. Buyuktosunoglu, P. Bose, T. J. Slegel, G. Salem, S. Carey, R. F. Rizzolo, and T. Strach, "Voltage noise in multi-core processors: Empirical characterization and optimization opportunities," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 368–380, IEEE, 2014.

[12] J. A. Guzmán-Rabasa, F. R. López-Estrada, B. M. González-Contreras, G. Valencia-Palomo, M. Chadli, and M. Perez-Patricio, "Actuator fault detection and isolation on a quadrotor unmanned aerial vehicle modeled as a linear parameter-varying system," *Measurement and Control*, vol. 52, no. 9-10, pp. 1228–1239, 2019.

[13] X. Qi, Z. Liu, Y. He, L. Yang, and J. Han, "Self-healing control framework against actuator fault of single-rotor unmanned helicopters," *Recent Progress in Some Aircraft Technologies*, p. 113, 2016.

[14] T. He, L. Zhang, F. Kong, and A. Salekin, "Exploring inherent sensor redundancy for automotive anomaly detection," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2020.

[15] M. Lee, B. Mudassar, T. Na, and S. Mukhopadhyay, "Warningnet: a deep learning platform for early warning of task failures under input perturbation for reliable autonomous platforms," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2020.

[16] S. S. Banerjee, S. Jha, J. Cyriac, Z. T. Kalbarczyk, and R. K. Iyer, "Hands off the wheel in autonomous vehicles?: A systems perspective on over a million miles of field data," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 586–597, IEEE, 2018.

[17] S. Jha, S. Banerjee, T. Tsai, S. K. Hari, M. B. Sullivan, Z. T. Kalbarczyk, S. W. Keckler, and R. K. Iyer, "Ml-based fault injection for autonomous vehicles: a case for bayesian fault injection," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 112–124, IEEE, 2019.

[18] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman, "Llfi: An intermediate code-level fault injection tool for hardware faults," in *2015 IEEE International Conference on Software Quality, Reliability and Security*, pp. 11–16, IEEE, 2015.

[19] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 375–382, IEEE, 2014.

[20] E. Cheng, S. Mirkhani, L. G. Szafaryn, C.-Y. Cher, H. Cho, K. Skadron, M. R. Stan, K. Lilja, J. A. Abraham, P. Bose, *et al.*, "Clear: Cross-layer exploration for architecting resilience-combining hardware and software techniques to tolerate soft errors in processor cores," in *Proceedings of the 53rd Annual Design Automation Conference*, pp. 1–6, 2016.

[21] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, "Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 249–258, IEEE, 2017.

[22] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R. K. Iyer, "Nftape: a framework for assessing dependability in distributed systems with lightweight fault injectors," in *Proceedings IEEE International Computer Performance and Dependability Symposium. IPDS 2000*, pp. 91–100, IEEE, 2000.

[23] D. Skarin, R. Barbosa, and J. Karlsson, "Goofi-2: A tool for experimental dependability assessment," in *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pp. 557–562, IEEE, 2010.

[24] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, p. 5, Kobe, Japan, 2009.

[25] S. Lynen, T. Sattler, M. Bosse, J. A. Hesch, M. Pollefeys, and R. Siegwart, "Get out of my lab: Large-scale, real-time visual-inertial localization.," in *Robotics: Science and Systems*, vol. 1, 2015.

[26] S. Liu, M. Watterson, S. Tang, and V. Kumar, "High speed navigation for quadrotors with limited onboard sensing," in *2016 IEEE international conference on robotics and automation (ICRA)*, pp. 1484–1491, IEEE, 2016.

[27] T. Gao, Z. Wan, Y. Zhang, B. Yu, Y. Zhang, S. Liu, and A. Raychowdhury, "ielas: An elas-based energy-efficient accelerator for real-time stereo matching on fpga platform," *arXiv preprint arXiv:2104.05112*, 2021.

[28] S. Krishnan, B. Borojerdian, W. Fu, A. Faust, and V. J. Reddi, "Air learning: An ai research platform for algorithm-hardware benchmarking of autonomous aerial robots," *arXiv preprint arXiv:1906.00421*, 2019.

[29] A. Loquercio, A. I. Maqueda, C. R. Del-Blanco, and D. Scaramuzza, "Dronet: Learning to fly by driving," *IEEE Robotics and Automation Letters*, vol. 3, no. 2, pp. 1088–1095, 2018.

[30] S. Krishnan, S. Chitlangia, M. Lam, Z. Wan, A. Faust, and V. J. Reddi, "Quantized reinforcement learning (quarl)," *arXiv preprint arXiv:1910.01055*, 2019.

[31] A. Anwar and A. Raychowdhury, "Autonomous navigation via deep reinforcement learning for resource constraint edge nodes using

transfer learning," *IEEE Access*, vol. 8, pp. 26549–26560, 2020.

[32] B. Boroujerdian, H. Genc, S. Krishnan, W. Cui, A. Faust, and V. Reddi, "Mavbench: Micro aerial vehicle benchmarking," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 894–907, IEEE, 2018.

[33] R. B. Rusu and S. Cousins, "3d is here: Point cloud library (pcl)," in *2011 IEEE international conference on robotics and automation*, pp. 1–4, IEEE, 2011.

[34] F. Fleuret, J. Berclaz, R. Lengagne, and P. Fua, "Multicamera people tracking with a probabilistic occupancy map," *IEEE transactions on pattern analysis and machine intelligence*, vol. 30, no. 2, pp. 267–282, 2007.

[35] J. J. Kuffner and S. M. LaValle, "Rrt-connect: An efficient approach to single-query path planning," in *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*, vol. 2, pp. 995–1001, IEEE, 2000.

[36] D. Gonzalez, J. Perez, V. Milanes, and F. Nashashibi, "A review of motion planning techniques for automated vehicles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 17, no. 4, pp. 1135–1145, 2016.

[37] K. H. Ang, G. Chong, and Y. Li, "Pid control system analysis, design, and technology," *IEEE transactions on control systems technology*, vol. 13, no. 4, pp. 559–576, 2005.

[38] I. B. M. Matsuo, L. Zhao, and W.-J. Lee, "A dual modular redundancy scheme for cpu–fpga platform-based systems," *IEEE Transactions on Industry Applications*, vol. 54, no. 6, pp. 5621–5629, 2018.

[39] S. Hudson, R. S. Sundar, and S. Koppu, "Fault control using triple modular redundancy (tmr)," in *Progress in Computing, Analytics and Networking*, pp. 471–480, Springer, 2018.

[40] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," *CoRR*, vol. abs/1705.05065, 2017.

[41] B. Sangchoolie, K. Pattabiraman, and J. Karlsson, "One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors," in *2017 47th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pp. 97–108, IEEE, 2017.

[42] J. J. Cook and C. Zilles, "A characterization of instruction-level error derating and its implications for error detection," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pp. 482–491, IEEE, 2008.

[43] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: probabilistic soft error reliability on the cheap," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1, pp. 385–396, 2010.

[44] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 123–134, 2012.

[45] Q. Lu, G. Li, K. Pattabiraman, M. S. Gupta, and J. A. Rivers, "Configurable detection of sdc-causing errors in programs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 3, pp. 1–25, 2017.

[46] G. Li, K. Pattabiraman, S. K. S. Hari, M. Sullivan, and T. Tsai, "Modeling soft-error propagation in programs," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 27–38, IEEE, 2018.

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

[48] B. Fang, Q. Lu, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "epvf: An enhanced program vulnerability factor methodology for cross-layer resilience analysis," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 168–179, IEEE, 2016.

[49] M. Ahmed, A. N. Mahmood, and M. R. Islam, "A survey of anomaly detection techniques in financial domain," *Future Generation Computer Systems*, vol. 55, pp. 278–288, 2016.

[50] M. Zhang, A. Raghunathan, and N. K. Jha, "Medmon: Securing medical devices through wireless monitoring and anomaly detection," *IEEE Transactions on Biomedical circuits and Systems*, vol. 7, no. 6, pp. 871–881, 2013.

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

[52] Z. Chen, C. K. Yeo, B. S. Lee, and C. T. Lau, "Autoencoder-based network anomaly detection," in *2018 Wireless Telecommunications Symposium (WTS)*, pp. 1–5, IEEE, 2018.

[53] Open Sources Robotics Foundation. http://wiki.ros.org/roslaunch/XML/node.

[54] D. E. Knuth, *Art of computer programming, volume 2: Seminumerical algorithms*. Addison-Wesley Professional, 2014.

[55] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[56] 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.*, pp. 29–40, IEEE, 2003.

[57] V. Sridharan and D. R. Kaeli, "Eliminating microarchitectural dependency from architectural vulnerability," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pp. 117–128, IEEE, 2009.

[58] C.-Y. Cher, M. S. Gupta, P. Bose, and K. P. Muller, "Understanding soft error resiliency of blue gene/q compute chip through hardware proton irradiation and software fault injection," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 587–596, IEEE, 2014.

[59] Q. Guan, X. Hu, T. Grove, B. Fang, H. Jiang, H. Yin, and N. DeBadeleben, "Chaser: An enhanced fault injection tool for tracing soft errors in mpi applications," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 355–363, IEEE, 2020.

[60] D. S. Khudia and S. Mahlke, "Harnessing soft computations for low-budget fault tolerance," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 319–330, IEEE, 2014.

[61] M. Didehban and A. Shrivastava, "nzdc: A compiler technique for near zero silent data corruption," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2016.

[62] H. So, M. Didehban, Y. Ko, A. Shrivastava, and K. Lee, "Expert: Effective and flexible error protection by redundant multithreading," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 533–538, IEEE, 2018.

[63] M. Maniatakos, N. Karimi, C. Tirumurti, A. Jas, and Y. Makris, "Instruction-level impact analysis of low-level faults in a modern microprocessor controller," *IEEE Transactions on Computers*, vol. 60, no. 9, pp. 1260–1273, 2010.

[64] S. E. Michalak, A. J. DuBois, C. B. Storlie, H. M. Quinn, W. N. Rust, D. H. DuBois, D. G. Modl, A. Manuzzato, and S. P. Blanchard, "Assessment of the impact of cosmic-ray-induced neutrons on hardware in the roadrunner supercomputer," *IEEE Transactions on Device and Materials Reliability*, vol. 12, no. 2, pp. 445–454, 2012.

[65] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," in *Proceedings of the 50th Annual Design Automation Conference*, pp. 1–10, 2013.

[66] "Uber's us safety report." https://www.uber.com/us/en/about/reports/us-safety-report/.

[67] "Waymo safety report on the road to fully self-driving," 2018. Technical report.

[68] "Automated driving systems 2.0: A vision for safety," 2018. Technical report.

[69] R. Palin, D. Ward, I. Habli, and R. Rivett, "Iso 26262 safety cases: Compliance and assurance," 2011.

[70] J. Birch, R. Rivett, I. Habli, B. Bradshaw, J. Botham, D. Higham, P. Jesty, H. Monkhouse, and R. Palin, "Safety cases and their role in iso 26262 functional safety assessment," in *International Conference on Computer Safety, Reliability, and Security*, pp. 154–165, Springer, 2013.

[71] A. H. M. Rubaiyat, Y. Qin, and H. Alemzadeh, "Experimental

resilience assessment of an open-source driving agent," in *2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC)*, pp. 54–63, IEEE, 2018.

[72] S. Jha, S. S. Banerjee, J. Cyriac, Z. T. Kalbarczyk, and R. K. Iyer, "Avfi: Fault injection for autonomous vehicles," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pp. 55–56, IEEE, 2018.

[73] S. Jha, T. Tsai, S. Hari, M. Sullivan, Z. Kalbarczyk, S. W. Keckler, and R. K. Iyer, "Kayotee: A fault injection-based system to assess the safety and reliability of autonomous vehicles to faults and errors," *arXiv preprint arXiv:1907.01024*, 2019.

[74] G. Li, Y. Li, S. Jha, T. Tsai, M. Sullivan, S. K. S. Hari, Z. Kalbarczyk, and R. Iyer, "Av-fuzzer: Finding safety violations in autonomous driving systems," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pp. 25–36, IEEE, 2020.

[75] Clare Naden, "DRONE MARKET SET TO TAKE OFF WITH NEW ISO STANDARD." https://www.iso.org/news/ref2461.html.

[76] Sarah Wray, "Cities should prepare for an increase in delivery drones." https://cities-today.com/cities-should-prepare-for-an-increase-in-delivery-drones/.

[77] 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.

[78] X. Iturbe, B. Venu, J. Jagst, E. Ozer, P. Harrod, C. Turner, and J. Penton, "Addressing functional safety challenges in autonomous vehicles with the arm tcl s architecture," *IEEE Design & Test*, vol. 35, no. 3, pp. 7–14, 2018.

[79] E. Times, "Tesla's Kitchen-Sink Approach to AVs." https://www.eetimes.com/teslas-kitchen-sink-approach-to-avs/.

[80] A. Kadav, M. J. Renzelmann, and M. M. Swift, "Tolerating hardware device failures in software," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 59–72, 2009.

[81] A. Kadav, M. J. Renzelmann, and M. M. Swift, "Fine-grained fault tolerance using device checkpoints," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 473–484, 2013.

[82] A. J. Peña, W. Bland, and P. Balaji, "Vocl-ft: introducing techniques for efficient soft error coprocessor recovery," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2015.

[83] A. Rezaei, G. Coviello, C.-H. Li, S. Chakradhar, and F. Mueller, "Snapify: capturing snapshots of offload applications on xeon phi manycore processors," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pp. 1–12, 2014.

[84] I. Akturk and U. R. Karpuzcu, "Acr: Amnesic checkpointing and recovery," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 30–43, IEEE, 2020.

[85] S. Feng, S. Gupta, A. Ansari, S. A. Mahlke, and D. I. August, "Encore: low-cost, fine-grained transient fault recovery," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 398–409, 2011.

[86] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Clover: Compiler directed lightweight soft error resilience," *ACM Sigplan Notices*, vol. 50, no. 5, pp. 1–10, 2015.

[87] J. Menon, M. De Kruijf, and K. Sankaralingam, "igpu: exception support and speculative execution on gpus," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 72–83, 2012.

[88] 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)*, pp. 44–57, IEEE, 2020.