Silent Data Corruption in Robot Operating System: A Case for End-to-End System-Level Fault Analysis Using Autonomous UAVs

Yu-Shun Hsiao*†, Zishen Wan*†,‡, Tianyu Jia†,¶, Radhika Ghosal†, Abdulrahman Mahmoud†, Arijit Raychowdhury‡, David Brooks†, Gu-Yeon Wei†, and Vijay Janapa Reddi†

†Harvard University ‡Georgia Institute of Technology ¶Peking University

***Abstract*—Safety and resiliency are essential components of autonomous vehicles. In this research, we introduce ROSFI, the first robot operating system resilience analysis methodology, to assess the effect of silent data corruption (SDC) on mission metrics. We use unmanned aerial vehicles (UAVs) as a case study to demonstrate that system-level parameters, such as flight time and success rate, are necessary for accurately measuring system resilience. We demonstrate that downstream ROS tasks such as planning and control are more susceptible to SDCs than the visual perception stage in the Perception-Planning-Control (PPC) compute pipeline. This observation only becomes apparent when we consider the complete end-to-end system-level pipeline, as opposed to isolated compute kernels, as previous work does. To enhance the safety and robustness of robot systems bound by size, weight, and power (SWaP), we offer two low-overhead anomaly-based SDC detection and recovery algorithms based on Gaussian statistical models and autoencoder neural networks. Our anomaly error protection techniques are validated in numerous simulated environments. We demonstrate that the autoencoder-based technique can recover up to all failure cases in our studied scenarios with a computational overhead of no more than 0.0062 percent. Finally, our open-source methodology can be utilized to comprehensively test the robustness of other Robot Operating System (ROS)-based applications. It is available for public download at https://github.com/harvard-edge/MAVBench/tree/mavfi.**

***Index Terms*—Silent Data Corruption, Robot Operating System, Anomaly Detection, Unmanned Aerial Vehicle, Resilience**

I. Introduction

Silent Data Corruption (SDC) has become an important problem for computing [1]. It has shown a significant threat in server scale systems [2], [3]. However, there are emerging application areas where SDCs' effects extend beyond just computational reliability into safety. Such an emerging area is autonomous vehicles where safety and reliability are critical.

Prior works have studied SDCs in the context of autonomous cars [4], [5]. However, prior work has yet to carefully examine the system-level effects of the middleware that orchestrates the entire perception, planning, and control (PPC) flow, where resiliency can be baked in to ensure SDC detection and recovery. To this end, we focus on the system-level implications of fault injection on the Robot Operating System using unmanned aerial vehicles (UAVs) as a proof of concept vessel, as UAVs are agile and highly sensitive to real-time input. UAVs are predicted to have a significant market shortly due to their diversity in applications and uses [6], [7]. Nevertheless, practical safety considerations, such as performing unmanned tasks safely and without collision, impede the wide adoption of these safety-critical applications in many real-world scenarios. SDCs caused by external radiation [8] and voltage noise [9] in the computational element like the computing subsystem present a major threat to the safe deployment of UAVs [10], [11].

There are multiple error mitigation techniques, including dynamic verification [12] and redundancy [13] at the hardware or software level to improve AVs' resilience. Although current methods prove their effectiveness, they face impracticality when applied to SWaP-constrained AVs like UAVs, primarily due to the constraints imposed by power requirements and the physical dimensions of UAV systems. Recent software techniques [14] for the resilience of convolutional neural networks (CNNs) on GPU does not apply to UAVs that typically do not have access to power-hungry GPUs onboard. Moreover, UAVs operate under stringent constraints, including limited onboard battery capacity, which imposes strict limitations on the total flight duration. Therefore, UAVs need a lightweight fault mitigation technique to prevent SDC from detouring or even crashing the UAV without compromising flight performance and availability. To this end, we set out to answer three fundamental questions.

1) What is the *SDCs' impact on system-level autonomy metrics*, such as flight time, energy consumption, and mission success rate for autonomous aerial robots such as UAVs?
2) Could conventional single, isolated-kernel SDC analysis provide similar insights as our *end-to-end fault characterization* based on system- and application-level metrics?
3) How to *enhance the safety and resilience of an autonomous robotic system against SDCs* with a lightweight mitigation technique under SWaP constraints embedded inside ROS?

To answer the first and the second question, we propose system-level metrics for evaluation and perform extensive fault characterizations (§IV) on a real physical ROS-based autonomous system. The autonomous UAV compute consists of an end-to-end PPC pipeline (Figure 1) that generates flight commands based on the environment in real-time. The PPC pipeline is the decision-making center for a UAV to maneuver

*These two authors contributed equally, listed in alphabetical order. Corresponding authors: Yu-Shun Hsiao (yushun_hsiao@g.harvard.edu), Zishen Wan (zishenwan@gatech.edu).

¶This work was done while the author was at Harvard University.

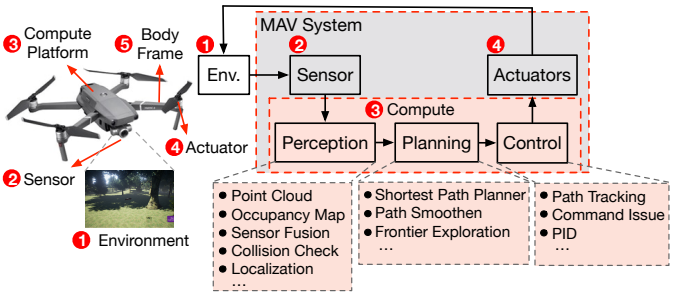


Fig. 1: End-to-end perception-planning-control computing (PPC) paradigm. Each PPC stage contains multiple kernels, and we study the safety and resilience of the end-to-end pipeline.

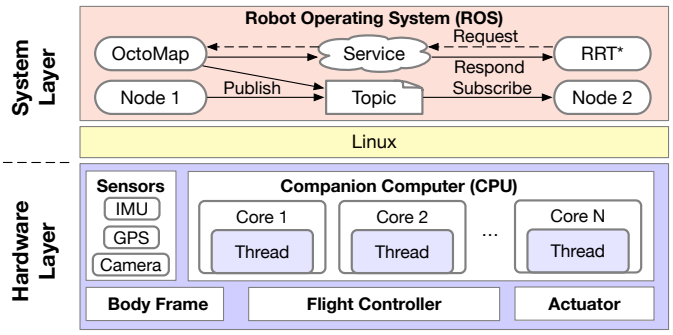


Fig. 2: System stack for a UAV. UAVs are complex cyber-physical systems with strong inter-dependencies between the computing and physical components. We focus on how faults in the companion computer affect the rest of the system.

safely. A SDC could cause a UAV to detour or even crash. To analyze the impact of SDCs, we adopt the bit-flip model for fault injections into the UAV's PPC pipeline and obtain quality-of-flight (QoF) metrics to quantify the impact of the faults on safety at the end-to-end whole application level.

Our findings show that application-aware metrics are essential for the resilience analysis of robotic applications. Analysis focusing on an individual computing stage without considering inter-kernel interactions leads to suboptimal insights and misguided conclusions. Prior works [15]–[18] rely on Silent Data Corruption (SDC) rate to determine the vulnerability of a single compute kernel. However, a high SDC rate at a kernel-level may have a negligible impact on the QoF metrics.

For the third question, we are interested in improving UAV's safety and resilience with a lightweight mitigation technique. To this end, we propose two software-directed and lightweight enhancements for the resilience of UAV systems (§V). Because agile robots like UAVs are constrained by size, weight, and power (SWaP), lightweight solutions are necessary. We perform data preprocessing to extract UAV's kinetics by calculating the delta value of the inter-kernel states. Based on the delta values, we perform two anomaly detection techniques. First, we perform a Gaussian-based anomaly detection (GAD) and recovery mechanism (§V-C). This technique features a Gaussian-based range detector to exclude outliers. Second, we use an autoencoder-based anomaly detection (AAD) technique for improved UAV resilience (§V-D). AAD adopts a neural network-based autoencoder to learn normal UAVs' kinematics and detect anomalies according to the reconstruction error of the input delta values. We show that our application-aware error detection and recovery techniques save energy by up to 1.91× than traditional redundancy-based hardware solutions (e.g., Dual Modular Redundancy (DMR), Triple Modular Redundancy (TMR)) that increase the weight and form factor of UAV and lead to performance overheads.

We evaluate the effectiveness of the two detection and recovery techniques across four vastly different types of environments on two computing platforms. Our experimental results demonstrate that the Gaussian-based technique recovers up to 89.6% of failure cases, and the autoencoder-based can recover all failures in the best-case scenario. Regarding QoF metrics, the Gaussian-based technique can recover the SDC-degraded flight time by up to 63.5% and 73.0% for the autoencoder-based technique. Furthermore, our measured overhead is less than 0.0062%. Moreover, our end-to-end fault analysis framework is more generally applicable to other types of (U)AVs.

In summary, the contributions of this work are as follows:

- First, we present an *end-to-end ROS-based application-aware resilience analysis framework ROSFI* to analyze robot applications' fault tolerance characteristics with proper metrics. ROSFI is seamlessly integrated with the ROS ecosystem and can be adapted for various ROS-based applications.
- Second, we conduct *fault tolerance characterizations of the PPC pipeline* from both kernel-level and system-level. We show that application-aware metrics are essential to understanding kernel vulnerability and fault's impact compared to the conventional isolated analysis.
- Third, we present *two low-cost anomaly error detection and recovery schemes* and evaluate them on different UAV configurations. By integrating anomaly error detection and recovery in ROS, We demonstrate that SDC impact on safety can be rectified in real-time with negligible overhead.

II. Background and Motivation

Safety standards. Many efforts have been dedicated to autonomous vehicle safety [19], [20]. The safety standard ISO 26262 [21] has been developed to provide guidance and safety requirements for vehicle and their systems. 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 analyses has been performed for the computing system [4], [22] of AVs. Unfortunately, to date, there are no comprehensive standards for *autonomous* UAV assessment. Prior works mainly focus on evaluating learning-based navigation system [23]–[25]. However, PPC compute paradigm-based UAV system is widely adopted in UAV systems nowadays, and its fault tolerance has not been adequately explored. Therefore, we take the first step to explore how SDCs propagate through the PPC pipeline and impact the safety of UAV systems. In this work, we define UAV reliability as the fault tolerance of autonomy kernels and UAV safety as the quality of flight at the application level for mission execution.

TABLE I: Comparison of fault injection techniques.

Abstraction Layer	Platform	Perf. (cycles/sec)	Exec. Time (1 run)	Exec. Time (1000 runs)
RTL	IVM Alpha processor RTL simulation [35]	6×10^2	4.2×10^5 hours	1.74×10^7days
Micro-architecture	gem5 simulator [36]	3×10^6	83.3 hours	3472 days
FPGA Emulation	OpenSPARC T1 FPGA emulation [37]	1×10^7	25 hours	1040 days
Architecture	SPARC simulator [38]	6×10^7	4.17 hours	173.6 days
Software (Ours)	x86 processor [39]	3×10^9	5 mins	3.48 days

System Layer. To understand how to address safety and resilience in UAVs, we must understand their complex system configuration. To this end, Fig. 2 presents the software-hardware stack of a UAV system. The system layer includes both Robot Operating System (ROS) and Linux. ROS is the commonly used "operating system" to provide communication functions and resource allocation for robotic applications. Despite its name, ROS is not an operating system but a collection of robotics middleware and tools aimed at managing cyber-physical systems by providing services for heterogeneous computing, low-level device control logic, and message-passing between processes. ROS consists of multiple ROS nodes, ROS services, and a ROS master to support the functionalities and communications [26], [27]. Underneath ROS, the Linux system maps workloads to compute units and schedules tasks at runtime. Each ROS node is treated as a process that is scheduled to a thread on CPU cores.

Hardware Layer. This layer consists of sensors, a companion computer, and a flight controller. The companion computer is used to execute the PPC kernels. These kernels usually act as ROS nodes and run on a general-purpose processor (e.g., CPU). Unlike autonomous vehicles, UAVs are limited in computing resources and energy budget, and thus, it is less common to equip UAVs with power-intensive GPU. The companion computer would generate high-level flight commands (e.g., velocity in x, y, z directions) in response to the sensor readings. The flight controller converts the high-level flight commands to low-level actuation commands to control and stabilize the UAV. In this work, we consider the faults in the companion computer and not the flight controller. The former determines the flight commands based on the real-time sensor readings, while the latter executes the commands. For instance, a corrupted yaw rotation generated by the companion computer could direct the UAV to point toward an obstacle and cause collisions. Meanwhile, the flight controller only executes the given commands without knowing the world models.

Algorithms. There has been significant advancement in perception, localization, mapping, and deep learning. Among all autonomy paradigms, the PPC computational pipeline is a widely used system [28]–[30]. 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 [31] and occupancy map [32]. The three-dimensional models are then fed into the planning stage to determine a collision-free trajectory by running a motion planner [33]. Finally, based on the UAV's dynamics, the control stage follows the planned path through controllers [34].

III. ROS Fault Injection

To analyze SDCs' impact on ROS, we first and foremost need a fault injection framework in the ROS middleware for injecting faults into the end-to-end UAV application pipeline to assess their impact systematically. This section presents ROSFI that supports fault injection with QoF metrics for evaluation.

A. Fault Injection Method Choices

Faults can be injected and simulated at different levels of the stack, ranging from low-level RTL [40] to high-level software [4], as shown in Tab. I. Although RTL simulation can accurately capture logic errors at the logic or gate levels, it requires an extremely long simulation time. In addition, it needs the RTL design or netlist of target processors, which is normally unavailable. On the other hand, software-level error injection has been widely utilized for system analysis with large vulnerability exploration space, showing significantly shorter simulation time and wide error cover range [41].

We adopt a software-level fault injection method to support system-level analysis, which aligns with previous fault tolerance studies for AVs by NVIDIA [4]. Software-level fault injection presents the best approach for an *end-to-end* study of the UAV pipeline; end-to-end implies the flow of data from the perception stage to the planning and control stages.

We assume that faults injected in ROSFI can corrupt the architectural states. Memory and caches are assumed to be protected with ECC. Each injected fault is characterized by its location and the injected value [4]. The faults injected into the architectural states of processors can manifest as errors in the inputs, outputs, and internal states of application-level ROS nodes. ROSFI can directly inject errors in ROS node outputs by corrupting the corresponding variables based on hardware layer results. These variables are ultimately stored in different levels of storage hierarchies. Single- or multiple-bit faults cause corruption of variables when not masked in hardware. Hence, faults are injected into these memory units, and the application-level variables are corrupted accordingly to emulate the faults.

The need for a software-based approach is justified by Fig. 3, which demonstrates the fault injection execution time of single-kernel and UAV experiments at different layers. One UAV run includes hundreds of single-kernel executions for the UAV autonomous navigation experiments, which takes 5 minutes per run or 3.48 days per scenario (i.e., 1000 runs) with our software-level execution. Consequently, it is infeasible for extensive fault analysis involving thousands of fault injections at the lower levels of the abstraction layers.

B. ROS Fault Injector Implementation

Fig. 4 illustrates the fault injection infrastructure of a ROS-based UAV system, including environment and UAV simulation on the host simulator and the UAV's PPC pipeline integrated with ROSFI on the companion computer. Each PPC stage contains one or multiple ROS nodes, and each ROS node comprises a single compute kernel, such as point cloud generation or motion planner. ROS node communicates through ROS topics (one-to-many communication) and ROS services (one-to-one communication). The ROSFI is built as a ROS

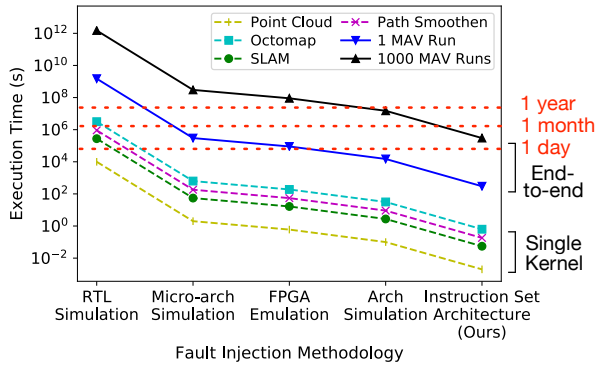


Fig. 3: Comparison of techniques at layers of abstraction. Performing end-to-end analysis requires fast execution time.

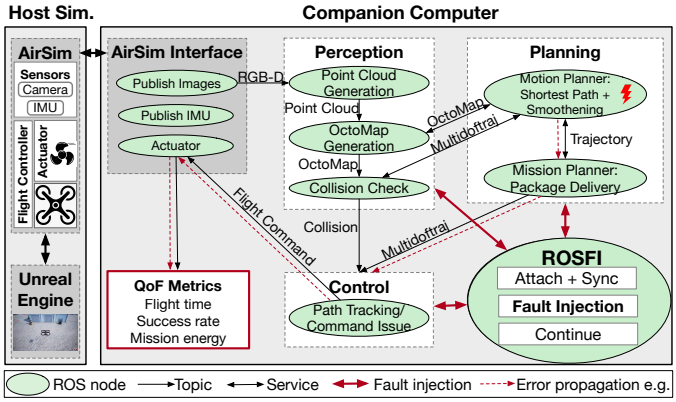


Fig. 4: Overview of the end-to-end application-aware ROSFI resilience analysis framework.

node to maintain our framework's portability, which leverages the ROS communication protocol and Linux system call.

To establish UAV experiments, we leveraged an open-source ROS-based UAV simulator, MAVBench [28]. MAVBench includes Unreal Engine to simulate the surrounding environment, AirSim simulator [42] to capture a UAV'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 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 mission QoF metrics are recorded. Although we use UAVs as an example in our framework, the fault analysis methodology is broadly applicable to any ROS-based use case.

Fig. 4 also illustrates an error propagation example within the system. For instance, 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 a flight command and impacts the overall QoF. The framework works on x86/Linux platforms. Fig. 5 shows the instruction-level fault injection details of ROSFI. Each oval node is a ROS node. The figure illustrates the fault injection sequence using an example for ROS node 2. During the system initialization phase, the ROSFI node publishes its process ID (pID) to all the other nodes and subscribes to their pID. Thus, the ROSFI node can attach and manipulate the other ROS nodes in the system via the *ptrace* system call supported by Linux. The *ptrace* system call allows synchronization and manipulation of processes' register files with much less overhead than the ROS communication protocol.

ROSFI is the first fault injection framework built on top of *ptrace* system call and ROS. It emulates SDCs that occur in the processor's functional units (e.g., arithmetic and logic units) by introducing transient bit-flips at the source or destination register of only one dynamic instruction, which is known as *instruction-level fault injection* [43]–[45]. We do not consider faults in the memories or caches as they can be protected by error correction codes (ECCs) in safety-critical applications. ECC is used to protect memory for robots that use TX2-level hardware (as considered in the paper). We also assume no faults in the processor's control logic, which constitutes only a small portion of the processor. This is in line with previous fault analyses. [16], [17], [46]. Hence, while our approach does not cover all the fault injection sites, it provides us with quality early-stage, end-to-end insights.

To inject faults, ROSFI selects a random time point to pause all the nodes during the simulation of real-time ROS applications. All ROS nodes' execution will be stopped before fault injection, ensuring that every node follows 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) 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 an instruction ranges from zero to two. If the value is zero, ROSFI resumes all ROS nodes' execution and repeats the above steps to obtain a new instruction. For more than one register under operation, ROSFI randomly chooses one register to inject. For the source register, according to the user-defined injection configuration, a single bit-flip or multiple bit-flips are introduced. For the destination register, before fault injection, ROSFI would step toward the next instruction to allow the current instruction to finish the write, 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 are notified to resume the execution. The faults injected into the registers of the processors could manifest as errors in the inputs, outputs, and internal states of the computational kernels. To better understand error propagation among PPC stages, ROSFI can inject errors into the inter-kernel states (the input of the other kernel) via *source-level fault injection* [4].

ROSFI can inject either single or multiple bit-flips simultaneously. In a previous study [47], it was shown that a single bit-flip is good enough for fault analysis since it can capture first-order vulnerability characteristics as well as multiple-bit-flips analyses. Therefore, for the analysis results in the paper, we mainly focus on a single bit-flip. For simplicity and clarity, we refer to single-bit-flip fault injection in the rest of the paper unless multiple bit-flips are specified.

ROSFI has the potential to extend to cover the memory and control logic of the processor. For memory, in this paper,

TABLE II: Comparison between ROSFI and prior fault injection methods.

	LLFI [44]	PINFI [43]	CLEAR [48]	SASSIFI [15]	DriveFI [4]	ROSFI (This work)
Autonomous Vehicle	✗	✗	✗	✗	✓(Vehicle)	✓(Drone)
Support ROS	✗	✗	✗	✗	✗	✓
Platform	CPU	CPU	CPU	GPU	GPU	CPU
Injection level	IR-level	Instruction-level	RLT-level	Instruction-level	Instruction-level, Source-level	Instruction-level, Source-level
Single Bit-flip	✓	✓	✓	✓	✓	✓
Double Bit-flips	✓	✗	✗	✓	✓	✓
Multiple Bit-flips	✗	✗	✗	✗	✓	✓

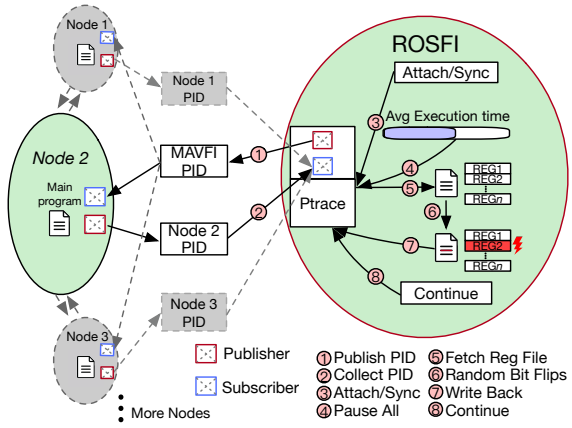


Fig. 5: The design details of the ROSFI fault injection node and its interactions with other ROS nodes.

we assume memory and caches are protected with SECDED codes. The faults injected in instructions may result in accessing the wrong data for computation, thus corrupting the variables. Faults in memory will result in corrupted computation data as well. These variables are ultimately stored in different levels of storage hierarchies (e.g., registers or caches). For control, since ROSFI obtains the whole instruction, it is able to modify the opcode of the instruction, thus the control logic.

C. Comparison to Prior Art

SDCs and resilience analysis have been studied for single kernels on CPU and GPU, as shown in Tab. II. However, prior methods [15]–[18] focus on the SDC rate of a single kernel, which does not directly translate to the impact of SDCs on UAVs' QoF metrics. On the one hand, more recently, DriveFI [4] explored the resilience impact of SDCs for autonomous driving systems on power-hungry GPU platforms.

However, *there currently does not exist a fault injection framework to analyze the resilience of ROS-based applications where the ROS nodes typically run on CPU with ROS [49].* Furthermore, a difference between prior autonomous vehicles resilience analysis studies and UAVs is that the *compute* environment of a UAV diverges from the recent trend in autonomous vehicles. While we find that most recent systems and tools are migrating to GPUs for increased parallel processing and DNN acceleration (and thus, may require GPU-centric resilience analysis tools such as SASSIFI [15] or NVBitFI [50]), UAVs continued to operate in the CPU realm for flexibility and lower energy needs. Currently, UAVs are intrinsically associated with CPUs due to the reliance on ROS (§II).

IV. End-to-End PPC Pipeline Fault-Tolerance

This section presents the fault tolerance analysis at two granularity levels, i.e., single-kernel level and application-aware system-level performance. We explore how errors would impact a single kernel and propagate through the whole PPC pipeline to affect UAV QoF metrics. Through the comparison, *we observe that isolated, single-kernel analysis (as is common practice) provides different or even opposite insights on the vulnerabilities of kernels than the application-aware analysis, which shows that end-to-end application-aware fault analysis is crucial to capturing SDCs' impact at the system level.*

Metrics: For single-kernel level analysis (§IV-A), we use benign, crash, hang, and SDC rates to evaluate the resilience of representative autonomy kernels. For application-level analysis (§IV-B), we use UAV QoF (i.e., mission success rate, time, energy) to evaluate end-to-end system performance and resilience characteristics.

A. Kernel-level Fault Tolerance Analysis

To prove the importance of application-aware fault analysis, we first conduct the single-kernel analysis with instruction-level fault injection. The single-kernel fault injection flow is similar to prior fault injection works [51]. We show that conducting similar analyses in a complex PPC pipeline can lead to misguided conclusions, specifically for UAV systems.

We evaluate the common kernels in the PPC pipeline, including *OctoMap* [52] for the perception stage, three sampling-based motion planners [53] (i.e., *RRT*, *RRTConnect*, *RRT**) for the planning stage, and PID controller [28] for the control stage. For each kernel, we perform instruction-level fault injection for 5000 runs in total. Each kernel is run without fault injection to obtain the error-free golden results. With fault injection, there are four outcomes: execution results same as the golden results (i.e. *benign*), execution exceptions (i.e. *crash*), infinite execution time (i.e. *hang*), and execution results different from the golden results (i.e. *silent data corruption* (SDC)) [54].

From a single-kernel perspective, the perception stage is the most critical when an SDC manifests. Fig. 6 shows that most compute kernels are more than 25% *benign* error-tolerant except for *OctoMap* at the perception stage. This is because SDC could easily manifest at the output (*Octree*) with noisy values for the *OctoMap* kernel. Hence, *OctoMap* is less resilient than sampling-based planning and PID control algorithms. On the other hand, the path planning kernels (*RRT*, *RRTConnect*, *RRT**) are all sampling-based algorithms, which are known for their high efficiency and performance for low-dimensional planning. Injected faults should not affect output results as

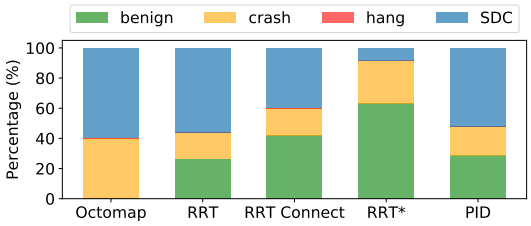


Fig. 6: Conventional isolated single-kernel analysis.

long as the corrupted way-point is not sampled. The more way-points we sample, the higher the probability the planning algorithms could sample a corrupted way-point, resulting in SDC. *RRTConnect* runs two *RRT* algorithms from both start and goal positions, ending up with fewer sampled way-points than *RRT*. *RRT** is the optimized version of *RRT* algorithm to find the shortest path by selecting even fewer way-points than *RRT* and *RRTConnect*, making *RRT** having the least SDC. The PID algorithm at the control stage also experiences around 25% *benign* cases as the PID has a simple self-healing mechanism to clip data outside of a bounded range.

B. End-to-End, System-level Fault Tolerance Analysis

Compared to the single-kernel fault analysis, we next conduct application-aware fault analysis based on our ROSFI framework. This end-to-end system-level characterization investigates how kernel errors would propagate through PPC pipelines and impact UAV performance. We assess the performance of the UAV using QoF metrics, encompassing key aspects such as flight success rate, distance, time, and energy. Ultimately, these metrics matter from an "application" or system-level perspective. The success rate quantifies the ratio of successful missions to the total number of flight runs. We define a mission as successful when the UAV reaches its destination without encountering any collisions. Conversely, a failure occurs when the UAV either collides with obstacles or is unable to identify a viable path to its intended destination. Flight time represents the total duration required for the UAV to reach its designated destination. Similarly, flight energy signifies the overall energy expended by the UAV to reach the destination. Since rotors dominate mission energy (∼95% [28]), flight energy positively correlates with flight time. It is worth mentioning that with reduced single-mission flight energy (E), the number of completed missions (N) under a battery charge (E_b) and success rate (SR) will increase through $N = SR \times (E_b/E)$.

We adopt the instruction-level fault injector supported by ROSFI to corrupt the 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 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. Only one of the kernels would have a one-time fault injection during each flight mission for fault injection runs. We achieved a 4.45% error margin with a 95% confidence level with 100 experiment runs per configuration. Without loss of generality, we limit our discussion to a navigation task in the *Sparse* environment here. More results are demonstrated in §VI.

Counter to the single-kernel perspective, from an end-to-end application perspective, the perception stage is the least critical when a SDC manifests. Prior works generally tend to focus on error resilience of the perception stage [1], [14], [55]. These are aligned with the single-kernel analysis, which shows that *OctoMap*, the main algorithm for perception, has the highest percentage of SDC among the evaluated kernels. However, as we show, for the perception stage both *Point Cloud Generation* and *OctoMap* have little to negligible impact on the system metrics, as shown in Fig. 7.

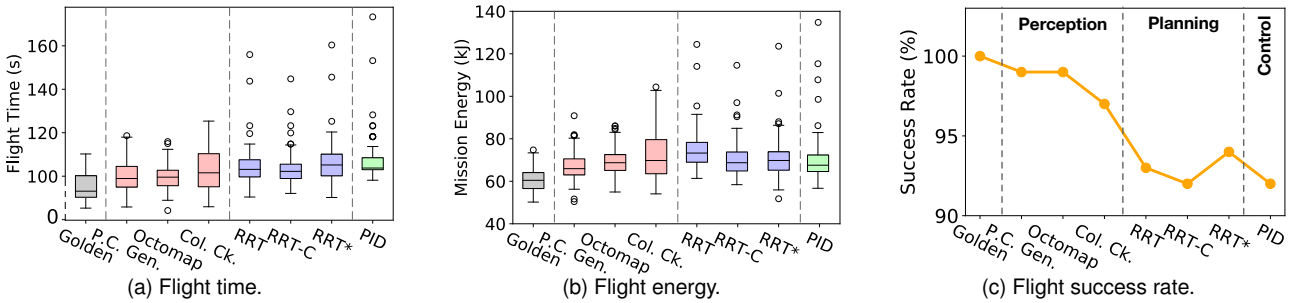
The reason why *OctoMap* has the least impact on QoF metrics is that even if an occupied voxel is corrupted and mistaken as a free voxel, the presence of all other surrounding voxels as occupied ensures that the UAV can still accurately determine the locations of obstacles. This holds as long as the resolution of the *OctoMap* is adequate, allowing the UAV to make the correct decisions regarding its flight path. This counter-intuitive insight underscores the significance of conducting comprehensive end-to-end analysis.*Collision Check* is the critical kernel in the perception stage since a false alarm can lead to trajectory re-planning or collisions.

From the end-to-end application-level perspective, planning, and control are more critical than perception, counter-intuitive to the single-kernel analysis. While the SDC percentages of planning and control kernels are lower than *OctoMap*, corrupted outputs (e.g., yaw, roll, pitch, velocity) from these two stages can directly lead to a detour or crash of the UAV. 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. 7c. Hence, the planning and control stages are more critical than the perception stage from an end-to-end application perspective.

C. Error Propagation Across PPC Stages

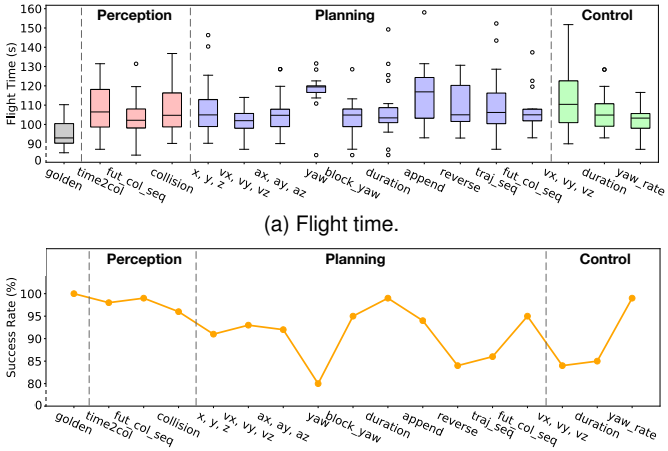
To understand error propagation across kernels, we analyze the impact of corrupted inter-kernel states in the PPC pipeline. This provides insights to improve the PPC kernels and facilitate error detection and mitigation in §V. We do source-level fault injection for this inter-kernel error propagation study with 100 navigation task runs for each evaluation.

As shown in Fig. 8, inter-kernel states exhibit different resilience to faults and have diverse impacts on UAV QoF metrics based on their functionality. 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. Faults in *time_to_collision* can skew the UAV's perceived distance to obstacles. Similarly, data corruption of *(x,y,z)* and *yaw* of way-points planned by motion planner can lead to a wrong direction or crash into obstacles, and faults in *(vx, vy, vz)* could make the UAV fail to keep track of a trajectory. As a result, the distorted trajectory leads to collision or increased flight time and mission energy.



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

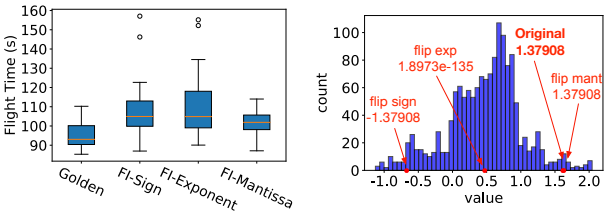
Fig. 7: Application-aware system-level end-to-end resilience analysis (flight time, energy, success rate) with ROSFI framework.



(a) Flight time.

(b) Flight success rate.

Fig. 8: End-to-end fault tolerance analysis.



(a) Flight time. (b) Bit-flip analysis.

Fig. 9: The impact of fault injection at different data fields.

Bit-flips in different data fields have distinct levels of impact on UAV performance. Prior works have evaluated data field impact on the processor and neural network [1], and we further corroborate this in end-to-end UAV systems from the application-level perspective. We conduct source-level fault injection at the float64 inter-kernel states *(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 the UAV's resilience and result in increased flight time, energy, and failure cases, as shown in Fig. 9a. Faults in the sign and exponent will result in a greater change in the inter-kernel states than faults in the mantissa field. For example, a single bit-flip at the exponent and sign could corrupt 1.38 to 0 and -1.38, respectively, as illustrated in Fig. 9b. The huge differences show that sign and exponent fields are more critical to the UAV system when a SDC manifests and propagates through the PPC pipeline. We further leverage this insight in lightweight UAV anomaly detection in §V.

To compare single bit-flip with multiple bit-flips, we evaluate the performance impact with multiple bit-flips fault injection as shown in Tab. III. In this experiment, 100 fault injections are performed for 1-, 3-, and 5-bits, respectively, at *(ax, ay, az)*, which are the output states of the planning stage. From 1-bit to 5-bits fault injection, the average flight time and energy slightly 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 with multiple bit-flip fault injection. However, the slight differences between single bit-flip and multiple bit-flips also show that single bit-flip can capture the first-order vulnerability characteristics as shown in the prior work [47].

TABLE III: QoF metrics with a different number of bit-flip injections.

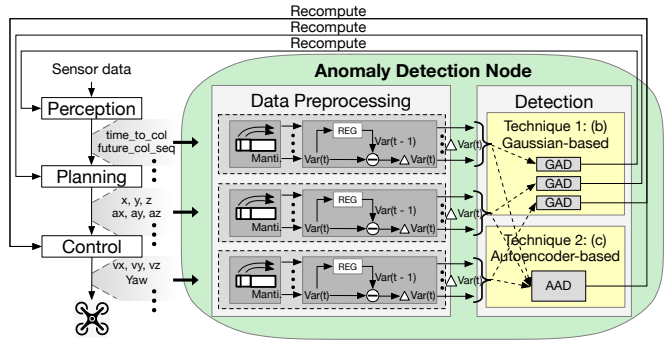
Bit Flips	Flight Time (s)	Flight Distance (m)	Total Energy (kJ)	Number of Re-plans	Success Rate (%)
0 (golden)	94.6	49.5	51.4	3.71	100
1	105.6	55.9	57.2	4.07	92
3	107.3	56.8	58.7	4.19	91
5	111.8	59.4	61.1	4.28	89

V. Error Detection and Recovery

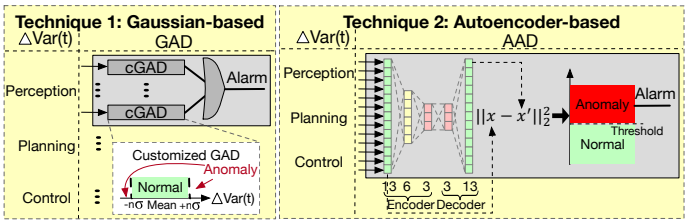
To enhance safety and resiliency, we further explore the detection and recovery technique based on the observations from ROSFI. As the conventional redundancy-based hardware protection introduces significant overhead, we propose two software-level low-overhead anomaly detection and recovery schemes for UAVs. The proposed schemes detect anomalous behavior of the inter-kernel states in the PPC pipeline and cease the error propagation, ensuring UAV's safety.

A. Overview of Detection and Recovery

Due to the low overhead and high effectiveness of anomaly detection , it has been used to distinguish anomaly from normal data distribution in many applications [56]. However, there is no effective general anomaly detection technique for different domains. Moreover, autonomous machines are complex systems that typically involve multiple kernels' heterogeneous computing. It is infeasible to separate normal data from anomaly based on the system's input (e.g., sensor readings) and output (e.g., flight commands). The heterogeneity also makes it hard to extract information from the system for anomaly detection. As



(a) Overview.



(b) Gaussian-based. (c) Autoencoder-based.

Fig. 10: The proposed anomaly detection and recovery scheme for UAV computational pipeline.

a consequence, no prior work has focused on anomaly detection to enhance the resilience of autonomous aerial vehicles.

We propose two anomaly detection techniques to detect SDC that could cause a safety hazard for UAVs, including Gaussian- and autoencoder-based techniques. It is observed that both techniques can greatly enhance the safety and resilience of UAVs with a low computational overhead. Fig. 10a shows the proposed anomaly detection and recovery scheme for UAVs. According to the analysis in Section IV-C, the inter-kernel states as shown in Fig. 8 are monitored for the anomalous SDC. The monitored states pass through a data preprocessing module to increase the detection performance while further reducing the computational overhead. After data preprocessing, the processed states go into either of the proposed anomaly detection techniques for supervision.

Error recovery is a feedback loop from the detection modules to the PPC pipeline. Once an anomalous behavior is detected, an alarm signal will be raised by the detection modules, triggering the recomputation of the corresponding stage, which prevents the corrupted inter-kernel states from propagating to the other kernels. The proposed detection and recovery system can greatly increase the resilience of UAV's PPC pipeline against SDCs that degrade the safety and flight performance of UAVs. Our approach focuses on SDC as ROS node *crash* can be detected by the ROS system. The ROS master node would restart the node automatically if it crashes [57].

B. Data Preprocessing

In Fig. 10a, the monitored inter-kernel states from the PPC pipeline are processed in the data preprocessing block before being sent to the anomaly detection block. Data preprocessing has two steps, including data format transformation and delta calculation. First, for data format transformation, the sign and exponent bits of *float64* states are transformed into 16-bits integer states. Since SDC at the mantissa bits of *float64* are insignificant as shown in §IV-C, only the sign and exponent bits are monitored to reduce the detection overhead. Second, the deltas of the incoming states are calculated. We define delta as the number of value changes from the previous time point to the current time point for an inter-kernel state.

Fig. 11 shows the insight of using the state's delta for anomaly detection. For most states, the value could have either uniform or Gaussian distribution. However, the uniform value distribution is not well suited to Gaussian-based anomaly detection, leading to very low detection accuracy. The uniform distribution can be transformed into a Gaussian distribution by calculating the states' delta, leveraging the inter-kernel states' temporal dependency. Furthermore, the state's delta range is much smaller than the original data. For instance, as shown in Fig. 11, the range of *multi_x*, *multi_vx*, and *multi_ax* states are reduced by 98%, 94%, and 76%, respectively, making the differences between normal and anomaly data even larger. Thus, data preprocessing can increase anomaly detection performance while decreasing the overhead of detection.

C. Gaussian-based Anomaly Detection

Fig. 10b shows the design of the Gaussian-based Anomaly Detection (GAD). Each PPC stage has a corresponding GAD that consists of several customized GAD (cGAD) for each inter-kernel state. If the value of an incoming state 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 PPC stage respectively. An alarm from a GAD would trigger the recomputation path of its corresponding stage, stopping the error propagation to the next stage.

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

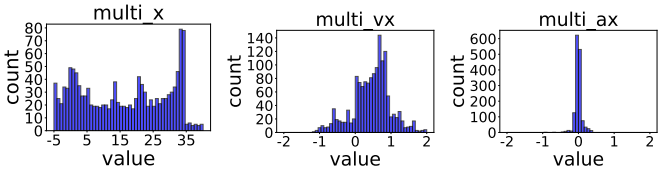
$$M_k = M_{k-1} + (x_k - M_{k-1})/k \tag{1}$$

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

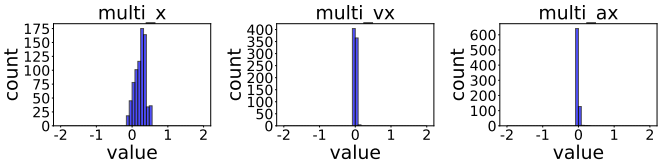
where k is the number of samples, M_k is the mean value for the k samples, and S_k is an auxiliary term used to compute standard deviation σ. At initialization, we introduce and set the terms $M_1 = x_1, S_1 = 0$. The parameters are updated online with the recurrence formulas above for new incoming data x_k [58]. 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.

D. Autoencoder-based Anomaly Detection

Fig. 10c shows the Autoencoder-based Anomaly Detection (AAD). The AAD block collects the processed states from all PPC stages as input. An alarm will be raised and triggers the recomputation of the control stage if an anomaly is detected. The proposed autoencoder comprises an encoder



(a) Histogram of the states' value.



(b) Histogram of the states' delta.

Fig. 11: Histogram comparison between the states' value and delta after data processing.

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. If the reconstruction error is beyond the threshold at the inference phase, the alarm will be raised. The threshold is the upper bound of the reconstruction error in the error-free golden run.

Rather than a separate Gaussian-based detection module for each PPC stage, we use a single autoencoder for the whole PPC pipeline to leverage the correlation among the inter-kernel states. 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 reducing the recomputation overhead as shown in §VII-D.

E. Recovery Scheme

Once an anomalous inter-kernel state is detected, the recomputation path will be triggered to cease the error propagation. The corresponding compute stage fetches the newest data from the previous compute stage or sensor and re-generates the results. Take the navigation task as an example. If an alarm is raised in the perception stage, the stage starts to recompute and fetch the newest RGB-D camera data. 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 plan a new trajectory. Finally, the flight command is monitored at the control stage before being sent back to the UAV. 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.

F. Anomaly Detection and Recovery on ROS

The anomaly detection and recovery scheme are built as a ROS detection node. This node contains the data preprocessing and anomaly detection functions (as explained previously). The detection node subscribes to the topics containing the inter-kernel states in the PPC pipeline as input and publishes recomputation signals to the corresponding stages if the detection function raises the alarm. The detection node can thus continuously supervise inter-kernel states of the PPC pipeline, avoiding error propagation among kernels and thus increasing the resilience of UAV's computational pipeline with negligible overhead.

VI. Experimental Setup

Hardware-in-the-loop Simulator. We used a closed-loop simulator as the experimental platform [28], including Unreal Engine (UE) to simulate the scenarios and AirSim to capture the UAV's kinematics. Sensors, including RGB-D camera and IMU, used in the experiments, are common for UAVs. An Intel i9-9940X CPU and an NVIDIA GTX 2080 Ti GPU are used as the host machine to simulate environments and the UAV. The companion computer has a CPU that takes sensory data and generates flight commands for UAVs.

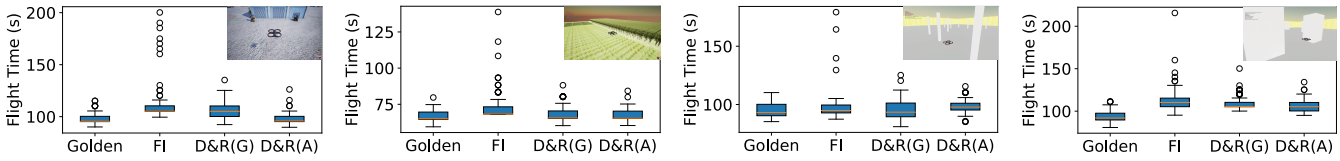
Training Environments. To create a training dataset for the autoencoder-based technique, we built an environment generator with configurable parameters (i.e., obstacle density and size of obstacle). The obstacle density is defined as the probability of a 10 * 10 grid spawned with an obstacle. Each obstacle is a cuboid with $n * n$ and infinite height (n is in [1, 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 (0.05 and 0.2) and two sizes of obstacles (3 and 5). Therefore, there are four configuration pairs in total, and each is run 25 times. A random seed is used to randomize the environment. For the Gaussian-based technique, the Gaussian models are updated with the same error-free training environments.

Evaluation Environments. The anomaly detection and recovery schemes are evaluated in four environments, which are unknown to the UAV. So we are not evaluating on training data. The *Factory* and *Farm* are provided by UE, representing common navigation scenarios with blocks, walls, and hedges. We generate the *Sparse* with [*0.05, 3*] and the *Dense* with [*0.2, 5*] using our environment generator. The random seed is fixed.

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, ROSFI only takes less than 5 *ms* for one-time fault injection, which is negligible for a typical flight mission that takes more than 100 *s*. For anomaly detection and recovery runs, we quantify the overhead of Gaussian and autoencoder-based techniques in §VII-D.

VII. Evaluation

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 instruction-level for each environment, including



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

Fig. 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.

TABLE IV: 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%

300 runs for each setting (i.e., *fault injection (FI)*, *detection & recovery with Gaussian (D&R(G))*, and *detection & recovery with autoencoder (D&R(A))*), as shown in Fig. 12. In each setting, we have 100 fault injections for each PPC stage. Each run includes a one-time single-bit injection. A total of 1000 runs is chosen where each run takes about 5 minutes. Even though ROSFI introduces a negligible overhead of only 5 *ms*, the experiment time is a limiting factor for the total runs.

A. Safety Metrics

Improvement of success rate. Tab. IV shows the success rates of UAV flights across four environments. In the fault injection runs, the success rate drops 9.7% in the *Dense* environment. Faults may easily cause collisions or fail to find a collision-free path in complex environments. By contrast, *Farm* is an obstacles-free environment. Even if a UAV 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 recovers more failure cases than the Gaussian-based scheme and increases the success rate close to or the same as the error-free runs.

Improvement of flight time. Fig. 12 shows the flight time of all successful cases in Tab. IV across four environments. Similar to §IV-B, the fault injection runs result in a much wider range of flight time than the golden run and increase the flight time by 73.8%, 74.2%, 62.6%, and 93.3% in the worst case for each environment, respectively. However, with Gaussian-based anomaly detection and recovery, many outliers can be recovered, and the worst-case flight time is recovered by 56.4%, 63.5%, 49.0%, and 58.7%. On the other hand, the autoencoder-based technique recovers most of the outliers and can recover the worst-case flight time by 64.2%, 68.4%, 57.8%, and 73.0%, outperforming the Gaussian method.

Comparison of Gaussian-based and autoencoder-based schemes. The autoencoder-based technique consistently outperforms the Gaussian-based technique in success rate and QoF metrics. The reason is that the autoencoder can leverage the correlation among the inter-kernel states; thus, it can detect the subtle discrepancy of the states. However, the Gaussian-based technique does not have correlation information among states. Therefore, it 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.

We provide both methods in the ROSFI framework. The Gaussian method serves as a practical and efficient solution for anomaly detection. It requires minimal data collection to update the standard deviation and mean values for each inter-kernel state. This simplicity and real-time adaptability are especially valuable in scenarios where immediate anomaly detection is critical, as it minimizes computation and overhead. The Autoencoder method, while more effective in terms of detection accuracy, necessitates offline training, making it more suitable for scenarios where detection accuracy is paramount and time constraints permit offline model training. In essence, the choice to utilize both methods stems from pragmatic consideration of the diverse needs in UAV operations.

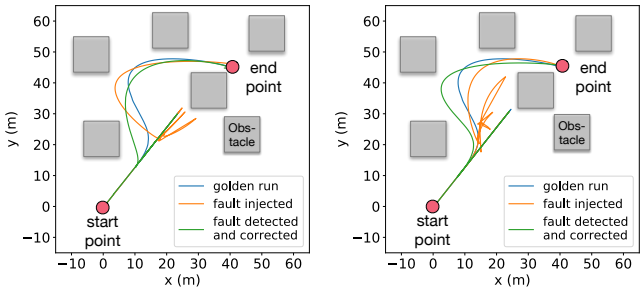
Comparison of environments. More dense environments with a higher density of obstacles make it difficult to recover from errors. For the *Dense* environment, a UAV has more complex trajectories to follow and more dynamic actions in response to the obstacles, making the range of the variable distribution wider. The wider distribution increases the number of false-negative detections. Thus, there is still a 20.1% gap between autoencoder-based recovery results and golden for the worst 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.

B. Trajectory Analysis

To show the impact of faults and the effectiveness of our detection and recovery schemes, we visualize UAV's trajectories in the *Dense* environment. We present the trajectories with the autoencoder-based technique, while the Gaussian-based technique has similar results when successful.

Fig. 13 shows the scenario in which a single-bit injection in the PPC stage can lead to a flight detour and how the detection and recovery scheme improves the flight. Without fault injection (blue curve), the UAV takes off at the start point and flies towards the endpoint in the beginning phase. Then, 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 faults corrupt critical inter-kernel states, such as the



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

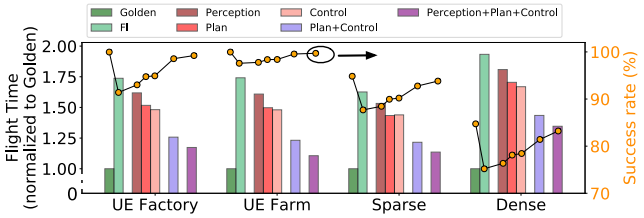
Fig. 13: Trajectories of the golden run, with fault injection, with both fault injection and error detection and recovery.

coordinate of a way-point, the path may be distorted. The UAV may not stop until it faces an obstacle (orange curve), which causes the UAV to fly back or re-plan its trajectory. The more often the UAV re-plans and detours 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, the corrupted way-point will be abandoned once an anomaly is detected. The alarm raised by the detection module triggers the stage recomputation. Therefore, the UAV would follow a better trajectory (green curve) without detour, which recovers the QoF metrics.

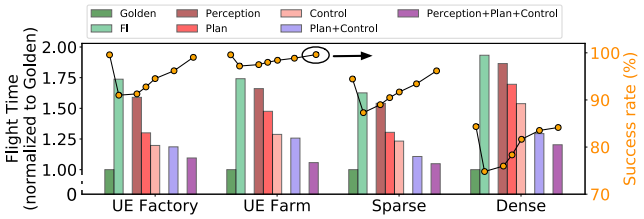
C. Anomaly Detection and Recovery

To evaluate error detection effectiveness in different PPC stages, we experiment with the anomaly detection and recovery scheme for certain compute stages.

Single-stage detection and recovery. We first experiment with anomaly detection and recovery by only detecting a single pipeline stage. As shown in Fig. 14, the Gaussian-based technique recovers the flight time by 16.2%, 29.9%, and 34.7% and the autoencoder-based technique recovers the flight time by 20.1%, 59.3%, and 73.2% for perception, planning, and



(a) Gaussian-based anomaly detection and recovery (flight time).



(b) Autoencoder-based anomaly detection and recovery (flight time).

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

TABLE V: Compute time overhead of detection and recovery.

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%	

control, respectively, along with the success rate improvement and flight energy savings, 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 that happened in the control stage. The reasons are twofold. First, the planning and control stages are more vulnerable to faults from the system perspective. Second, any error propagated from previous stages has to pass through the control stage before corrupting the flight command. The evaluation of the individual stage lines up with the analysis in §IV-B.

Multi-stage detection and recovery. To understand how different stages affect anomaly detection and recovery, we apply the scheme to multi-stages, namely the planning-and-control (PC) stage and all PPC stages. The Gaussian method recovers the flight time by 65.1%, 76.5%, and the autoencoder-based recovers by 74.8%, 87.1% for PC and PPC, respectively, along with the success rate increase and fight energy savings, in *Factory* environment. For the Gaussian method, detecting the PC stage significantly outperforms the single-stage detection and recovery in all environments. For the autoencoder-based technique, detecting PC stage achieves 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 indicate that a UAV achieves similar or higher performance by monitoring more stages, and the performance benefit is greater for complex environments.

D. Compute Overhead

Software-level protection. We study the overhead of the proposed software-level anomaly detection and recovery scheme across the tested environments. The overhead is the total detection and recomputation time for each mission. Tab. V 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 is 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 as the control stage recomputation only takes 0.46 *ms*. As the scheme is operated at the software level with negligible overhead, it is possible to deploy multiple anomaly detection nodes to improve the robustness of detection nodes.

Hardware-level protection. To demonstrate the benefits of our schemes over redundancy-based hardware protections, we adopt a UAV visual performance model from [59] to evaluate the performance overhead of microarchitecture-based redundancy schemes (DMR and TMR) on UAV. Two types

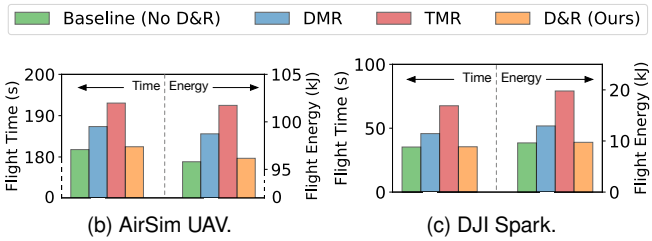
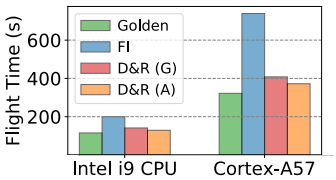


Fig. 15: Comparison of DMR, TMR, and the anomaly detection and recovery schemes on ARM Cortex-A57.



	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

Fig. 16: Comparison of detection and recovery schemes.

of UAVs, AirSim UAV and DJI Spark (with the same specs as [60]), are used as experimental platforms. Fig. 15 shows that TMR incurs a flight time increase by 1.06× on AirSim UAV and 1.91× on DJI compared to the 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 of the UAV system, our scheme demonstrates negligible performance overhead.

E. Computing Platform Comparison

To show the portability we conduct fault injection on different platforms by introducing a single bit-flip at the inter-kernel states as in §IV-C. Fig. 16 shows a similar error trend for both platforms. On the TX2, the worst flight time increases 2.8× since TX2 is an edge platform that has slower responses to environmental changes. However, with the anomaly detection ROS node continuously monitoring the anomaly of inter-kernel states, the flight time is recovered by 79.3% and 88.0% with Gaussian-based and autoencoder-based techniques.

VIII. Conclusion and Future Work

Safety is paramount in autonomous vehicles. We present the first ROSFI fault analysis framework to enable system-level resilience analysis and show that system-level analysis is essential to capture system vulnerability compared to isolated, single-kernel fault injection analysis which is the common approach. Furthermore, we propose two anomaly detection and recovery schemes and demonstrate that with less than 0.0062% compute overhead, the autoencoder-based scheme can recover up to 100% failure cases in the tested scenario. Being an instruction-level fault injection framework, ROSFI has limitations. Nonetheless, we believe it moves the field of fault and resilience analysis forward in a significant way, providing a unique contribution of application-aware fault and resilience analysis in the context of robotics. Future directions include extending the fault model to consider micro-architecture level errors that also manifest as bit flips in architecture states read by an instruction and incorporating different AV pipelines.

References

[1] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding error propagation in deep learning neural network (dnn) accelerators and applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2017.

[2] H. D. Dixit, S. Pendharkar, M. Beadon, C. Mason, T. Chakravarthy, B. Muthiah, and S. Sankar, "Silent data corruptions at scale," *arXiv preprint arXiv:2102.11245*, 2021.

[3] P. H. Hochschild, P. Turner, J. C. Mogul, R. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat, "Cores that don't count," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, pp. 9–16, 2021.

[4] 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.

[5] S. Jha, S. Cui, S. S. Banerjee, T. Tsai, Z. Kalbarczyk, and R. Iyer, "Ml-driven malware that targets av safety," *arXiv preprint arXiv:2004.13004*, 2020.

[6] H. Shakhatreh, A. H. Sawalmeh, A. Al-Fuqaha, Z. Dou, E. Almaita, I. Khalil, N. S. Othman, A. Khreishah, and M. Guizani, "Unmanned aerial vehicles (uavs): A survey on civil applications and key research challenges," *Ieee Access*, vol. 7, pp. 48572–48634, 2019.

[7] 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," *IEEE Circuits and Systems Magazine*, vol. 21, no. 2, pp. 48–74, 2021.

[8] 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.

[9] 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.

[10] Z. Wan, K. Swaminathan, P.-Y. Chen, N. Chandramoorthy, and A. Raychowdhury, "Analyzing and improving resilience and robustness of autonomous systems," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pp. 1–9, 2022.

[11] Y.-S. Hsiao, Z. Wan, T. Jia, R. Ghosal, A. Mahmoud, 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," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, IEEE, 2023.

[12] R. Nathan and D. J. Sorin, "Nostradamus: Low-cost hardware-only error detection for processor cores," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, IEEE, 2014.

[13] E. Talpes, D. D. Sarma, G. Venkataramanan, P. Bannon, B. McGee, B. Floering, A. Jalote, C. Hsiong, S. Arora, A. Gorti, *et al.*, "Compute solution for tesla's full self-driving computer," *IEEE Micro*, vol. 40, no. 2, pp. 25–35, 2020.

[14] S. K. S. Hari, M. Sullivan, T. Tsai, and S. W. Keckler, "Making convolutions resilient via algorithm-based error detection techniques," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[15] 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.

[16] 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.

[17] Z. Chen, G. Li, K. Pattabiraman, and N. DeBardeleben, "Binfi: an efficient fault injector for safety-critical machine learning systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–23, 2019.

[18] G. Papadimitriou and D. Gizopoulos, "Demystifying the system vulnerability stack: Transient fault effects across the layers," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 902–915, IEEE, 2021.

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

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

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

[22] 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.

[23] Z. Wan, A. Anwar, Y.-S. Hsiao, T. Jia, V. J. Reddi, and A. Raychowdhury, "Analyzing and improving fault tolerance of learning-based navigation systems," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 841–846, IEEE, 2021.

[24] Z. Wan, A. Anwar, A. Mahmoud, T. Jia, Y.-S. Hsiao, V. J. Reddi, and A. Raychowdhury, "Frl-fi: Transient fault analysis for federated reinforcement learning-based navigation systems," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 430–435, IEEE, 2022.

[25] Z. Wan, N. Chandramoorthy, K. Swaminathan, P.-Y. Chen, V. J. Reddi, and A. Raychowdhury, "Berry: Bit error robustness for energy-efficient reinforcement learning-based autonomous systems," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2023.

[26] 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.

[27] V. Mayoral-Vilches, J. Jabbour, Y.-S. Hsiao, Z. Wan, A. Martínez-Fariña, M. Crespo-Alvarez, M. Stewart, J. M. Reina-Munoz, P. Nagras, G. Vikhe, *et al.*, "Robotperf: An open-source, vendor-agnostic, benchmarking suite for evaluating robotics computing system performance," *arXiv preprint arXiv:2309.09212*, 2023.

[28] 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.

[29] S. Liu, Z. Wan, B. Yu, and Y. Wang, "Robotic computing on fpgas," *Synthesis Lectures on Computer Architecture*, vol. 16, no. 1, pp. 1–218, 2021.

[30] Q. Liu, Z. Wan, B. Yu, W. Liu, S. Liu, and A. Raychowdhury, "An energy-efficient and runtime-reconfigurable fpga-based accelerator for robotic localization systems," in *2022 IEEE Custom Integrated Circuits Conference (CICC)*, pp. 01–02, IEEE, 2022.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[37] A. Pellegrini, R. Smolinski, L. Chen, X. Fu, S. K. S. Hari, J. Jiang, S. V. Adve, T. Austin, and V. Bertacco, "Crashtest'ing swat: Accurate, gate-level evaluation of symptom-based resiliency solutions," in *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1106–1109, IEEE, 2012.

[38] M. Daněk, L. Kafka, L. Kohout, J. Sỳkora, and R. Bartosiński, "The leon3 processor," in *UTLEON3: Exploring Fine-Grain Multi-Threading in FPGAs*, pp. 9–14, Springer, 2013.

[39] K. S. Yim, Z. Kalbarczyk, and R. K. Iyer, "Measurement-based analysis of fault and error sensitivities of dynamic memory," in *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pp. 431–436, IEEE, 2010.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] A. Mahmoud, R. Venkatagiri, K. Ahmed, S. Misailovic, D. Marinov, C. W. Fletcher, and S. V. Adve, "Minotaur: Adapting software testing techniques for hardware errors," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 1087–1103, 2019.

[46] 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.

[47] 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.

[48] 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.

[49] V. Mayoral-Vilches, S. M. Neuman, B. Plancher, and V. J. Reddi, "Robotcore: An open architecture for hardware acceleration in ros 2," in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2022.

[50] "NVBitFI: An Architecture-level Fault Injection Tool for GPU Application Resilience Evaluations." https://github.com/NVlabs/nvbitfi.

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

[52] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "Octomap: An efficient probabilistic 3d mapping framework based on octrees," *Autonomous robots*, vol. 34, pp. 189–206, 2013.

[53] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The international journal of robotics research*, vol. 30, no. 7, pp. 846–894, 2011.

[54] 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.

[55] F. F. dos Santos, P. F. Pimenta, C. Lunardi, L. Draghetti, L. Carro, D. Kaeli, and P. Rech, "Analyzing and increasing the reliability of convolutional neural networks on gpus," *IEEE Transactions on Reliability*, vol. 68, no. 2, pp. 663–677, 2018.

[56] L. Ruff, J. R. Kauffmann, R. A. Vandermeulen, G. Montavon, W. Samek, M. Kloft, T. G. Dietterich, and K.-R. Müller, "A unifying review of deep and shallow anomaly detection," *Proceedings of the IEEE*, 2021.

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

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

[59] 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.

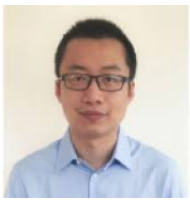
[60] S. Krishnan, Z. Wan, K. Bhardwaj, N. Jadhav, A. Faust, and V. J. Reddi, "Roofline model for uavs: A bottleneck analysis tool for onboard compute characterization of autonomous unmanned aerial vehicles," in *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 162–174, IEEE, 2022.

Yu-Shun Hsiao received the B.E. degree in electrical engineering from National Tsing Hua University, Hsinchu, Taiwan, in 2018. He is currently pursuing a Ph.D. degree in computer science at Harvard University, Cambridge, MA, USA.

He was a research scientist intern at NVIDIA Architecture Research Team, USA, in 2021 and 2022. His research interests include computer architecture and system-software design to enable efficient and resilient autonomous machines, including vehicles, drones, and robotic arms.

Zishen Wan (Student Member, IEEE) received the B.E. degree in electrical engineering and automation from Harbin Institute of Technology, Harbin, China, in 2018, and the M.S. degree in electrical engineering from Harvard University, Cambridge, MA, USA, in 2020. Currently, he is pursuing Ph.D. degree with the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, USA.

His research interests include computer architecture, VLSI, and embedded systems, with a focus on designing efficient and reliable hardware and systems for autonomous machines and edge intelligence.



Tianyu Jia (Member, IEEE) received the B.S. and M.S. degrees from Beijing University of Posts and Telecommunications, Beijing, China, and Ph.D. degree in computer engineering from Northwestern University, Evanston, IL, USA, in 2019. He was a Postdoctoral Fellow at Harvard University and an Assistant Research Professor at Carnegie Mellon University. He is currently an Assistant Professor at Peking University, Beijing, China. His research interests include accelerator design for domain-specific applications, heterogeneous SoC design and optimization. Dr. Jia was awarded the IEEE Solid-State Circuit Society Predoctoral Achievement Award in 2020.

Radhika Ghosal received her B.Tech. degree in computer science and engineering from Indraprastha Institute of Information Technology, Delhi, India in 2019. She is currently a Ph.D. student in computer science at Harvard University, Cambridge, MA. Her research interests lie in designing high performance compute hardware and software for robotics, and her research is supported by the NSF Graduate Research Fellowship.

Abdulrahman Mahmoud (Member, IEEE) received his B.S.E. degree in electrical engineering from Princeton University, Princeton, NJ, USA in 2013, and Ph.D. degree in computer science from University of Illinois at Urbana-Champaign, Urbana, IL, USA in 2020. He is currently a Postdoctoral Fellow at Harvard University, Cambridge, MA, USA. His current research interests include resilient and efficient machine learning systems using hardware-software co-design. Dr. Mahmoud is the lead developer of multiple research tools along this line of work, including PyTorchFI and GoldenEye, which have been well received and used for efficient ML design in recent years.

Arijit Raychowdhury (Fellow, IEEE) received the Ph.D. degree in electrical and computer engineering from Purdue University, West Lafayette, IN, USA, in 2007. He is currently the Steve W Chaddick Chair and a Professor with the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, USA. He is also the Director of the Center for the Co-Design of Cognitive Systems (CoCoSys), a Joint University Microelectronics Program 2.0. His research interests include low-power digital and mixed-signal circuit design, signal processors, and exploring interactions of circuits with device technologies.

Dr. Raychowdhury is currently a Distinguished Lecturer of the IEEE Solid State Circuits Society (SSCS). He serves on the Technical Program Committee of key circuits and design conferences, including ISSCC, VLSI Symposium, DAC, and CICC. He is the winner of several awards, including the SRC Technical Excellence Award in 2021, the Qualcomm Faculty Award in 2021 and 2020, the IEEE/ACM Innovator under 40 Award, and the NSF CISE Research Initiation Initiative Award (CRII) in 2015.

David Brooks (Fellow, IEEE) received the B.S. degree in electrical engineering from the University of Southern California, Los Angeles, CA, USA, in 1997, and the M.A. and Ph.D. degrees in electrical engineering from Princeton University, Princeton, NJ, USA, in 1999 and 2001, respectively. He is currently the Haley Family Professor of computer science with the School of Engineering and Applied Sciences, Harvard University, Cambridge, MA, USA. His current research interests include resilient and power-efficient computer hardware and software design for high-performance and embedded systems. Dr. Brooks was a recipient of several honors and awards, including the ACM Maurice Wilkes Award and ISCA Influential Paper Award.

Gu-Yeon Wei (Senior Member, IEEE) received the B.S., M.S., and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, USA, in 1994, 1997, and 2001, respectively. He is currently a Robert and Suzanne Case Professor of electrical engineering and computer science with the Paulson School of Engineering and Applied Sciences (SEAS), Harvard University, Cambridge, MA, USA. His research interests span multiple layers of a computing system: mixed-signal integrated circuits, computer architecture, and design tools for efficient hardware. His research efforts focus on identifying synergistic opportunities across these layers to develop energy-efficient solutions for a broad range of systems from flapping-wing microrobots to machine learning hardware for the Internet of Things (IoT) devices to large-scale servers.

Vijay Janapa Reddi (Member, IEEE) received the Ph.D. degree in computer science from Harvard University, Cambridge, MA, USA, in 2010. He is currently an Associate Professor with the John A. Paulson School of Engineering and Applied Sciences, Harvard University, where he is the Director of the Edge Computing Laboratory. His research interests include computer architecture and systemsoftware design, with an emphasis on the context of mobile and edge computing platforms based on machine learning. Dr. Reddi was a recipient of multiple awards, including the Best Paper Award at MICRO 2005 and HPCA 2009, the IEEE's Top Picks in Computer Architecture Awards in 2006, 2010, 2011, 2016, and 2017, the Google Faculty Research Awards in 2012, 2013, 2015, and 2017, the Intel Early Career Award in 2013, the National Academy of Engineering Gilbreth Lecturer Honor in 2016, and the IEEE TCCA Young Computer Architect Award in 2016.