2024 ACM/IEEE 15th International Conference on Cyber-Physical Systems (ICCPS)

Sensor Data Transplantation for Redundant Hardware Switchover in Micro Autonomous Vehicles

Cailani Lemieux-Mack
Vanderbilt University
kailani.j.lemieux.mack@vanderbilt.edu

Kevin Leach
Vanderbilt University
kevin.leach@vanderbilt.edu

Kevin Angstadt
St. Lawrence University
kangstadt@stlawu.edu

***Abstract*—As our reliance on micro autonomous vehicles increases, security vulnerabilities and software defects threaten the successful completion of tasks and missions. Recent work has developed end-to-end toolchains that provide trusted and resilient operation in the face of defects and attacks. These toolchains enable automatically repairing (and patching) the control software in the event of a failure. Existing techniques force the subject control software to terminate and the vehicle to be motionless, making the restart or post-repair deployment more complex and slow. The challenge remains to ensure that vehicle control software can recover from attacks and defects quickly and safely, even while the target vehicle remains in motion.**

This paper presents a technique for faster, simpler, and seamless hardware switchover that operates while the vehicle is in motion. The key contribution is the ability to restart the control software post-repair while the vehicle is in motion by transplanting sensor data between onboard control computers to bypass a costly portion of initialization. Although existing checkpoint and restore methods allow software to recover execution at a known-functional state, they are not lightweight enough to support recovery during mission execution. Instead, our approach transplants known-good sensor data from a trusted, isolated execution environment in the onboard computing hardware. Our evaluation successfully reproduces prior simulation results in hardware. Further, sensor transplantation allows for successful initialization while in motion, reduces time-to-ready by 40%, and is robust to variances in sensor readings.

***Index Terms*—resilience, autonomous vehicles, redundant hardware**

I. Introduction

In recent years, uncrewed micro autonomous vehicles, such as quad-rotor drones and small rovers, have been used for a variety of tasks, including mapping and rescue missions [35], defensive military applications [23], agriculture [22], and sea-ice management [28]. It is critical that the software on these uncrewed vehicles operates correctly to minimize human injury and damage to the vehicle and surrounding environment.

Both security vulnerabilities and software defects have been documented in drone control software [1], [42], [47]. An attacker can remotely compromise control software running on the device, endangering the drone and its mission, and more generally, software faults can lead to dangerous malfunctions. Mitigation techniques often use hardware and software redundancy to provide assurance that the system can withstand a mission-threatening problem with the primary computer. Redundant software has been used in other domains, such as N-version systems, to detect and prevent security vulnerabilities in software [8]. Similarly, duplicate hardware has been used to mitigate the effects of software and hardware failure in satellites [33] and commercial aviation [12].

Techniques like binary hardening [13], [36], runtime monitoring [50], and automated program repair [24], [25] have all been used to improve the resiliency and fault tolerance of control software. Previous work has combined these three techniques to deploy resilient and fault-tolerant *automated resiliency frameworks* for vehicle control systems [26]. However, a key weakness of prior work is that the system and control software must be stopped mid-mission to address a software fault. Critically, there are no guarantees that the location where the vehicle must stop is both suitable for a restart and safe to remain for the length of time required to find, build, and deploy a software patch. Moreover, prior automated resiliency frameworks have been developed and evaluated only in simulation and have not yet been applied in practice. A practical application of such automated resiliency frameworks must operate on a real platform and support resilient operation while the vehicle is in motion. Further, the approach should be dependable and robust in that it can detect and recover from anomalous behavior such as attacks or faults.

When a mission *controller* starts, it must initialize its sensors as well as other parts of the software state before it is capable of autonomous control.[1] Current software implementations assume that the vehicle is stationary on the ground for this initialization. This assumption is undesirable because (1) it may not be possible for the vehicle to stop in certain terrain, and (2) upon encountering a defect or anomaly, the vehicle may need to stop in—or navigate through—a compromised environment. We lift this assumption by allowing software initialization while the vehicle is in motion under the control of a secondary computer. Our key insight is that the secondary computer is fully initialized and running; thus, we can transplant any necessary initialization data from the

This work was supported in part by the National Science Foundation (grant CCF-2211751) and by DARPA.

[1]We use the term *controller* as an all-encompassing term, representing the combination of both the *control algorithms* and also the hardware and software necessary to interface with motors and sensors.

secondary computer to the primary computer. This data can be used to start the controller in motion to retake control from a live state without requiring re-initialization from scratch in a stationary position.

In this paper, we present a real hardware system that leverages an automated resiliency framework to allow an autonomous vehicle to maintain autonomous operation while software faults are detected and repaired. Our design includes a *primary controller* that executes missions and a simplified *secondary controller* that can take control in the event of an anomaly or attack. We also present the first approach that allows the primary computer to restart while the uncrewed vehicle is in motion under the control of a secondary computer. Current software requires the vehicle to reinitialize sensors and internal controller state, a costly operation that threatens physical vehicle safety because the vehicle must remain stationary.

We evaluate our data transplantation algorithm using controlled experiments on our dual-controller platform. First, we measure the success rate and time needed for the control software to initialize when the vehicle is both stationary and in motion. Then, we measure the sensor data accuracy needed for successful controller initialization. Finally, we replicate experimental scenarios from [26] to test the robustness of our dual-controller design and its ability to support switchover when software faults are detected. Our experiments show that sensor data transplantation allows for the previously unsupported in-motion initialization and reduces static initialization time an average of 40–45%. Our replication study demonstrates that our dual-controller design can successfully respond to—and switch control for—a suite of thirteen different indicative autonomous vehicle defect scenarios.

To summarize, the main contributions of this paper are:

- A low-cost, small, redundant, dual-controller autonomous vehicle platform for resilient mission operation in the face of software defects and vulnerabilities.
- A sensor data transplantation algorithm that allows control software to initialize while a vehicle is in motion.
- An empirical analysis of our platform demonstrating a 40% improvement in initialization time and successful hardware switchover across a suite of autonomous vehicle defect scenarios.

Ultimately, this work improves the state-of-the-art by expanding the conditions under which autonomous vehicles can respond to threats and software defects.

II. Background

In this section we provide relevant background information. We provide additional information in Appendices A and B.

A. State Estimation Filters

Autonomous vehicle controllers make actuation decisions based on sensor measurements of the physical world. For correct operation, autonomous vehicles must maintain an accurate model of position, orientation, and other metrics, which we collectively refer to as the vehicle's *state*. However, sensor data is known to be noisy [45] which causes error in the internal state of the vehicle and can interfere with safe and trusted operation. Thus, it is common to use a state estimation filter to increase the accuracy of the known state.

A state estimation filter is a statistical digital-signal-processing filter that fuses sensor measurements with previous time-step estimations. This estimation is done using a physical model and the previous accepted state. The result of this fusion of measurements and estimate then becomes the new accepted state for the next cycle of the control loop. This process allows for calculation of an accurate state without completely trusting any one sensor measurement.

Our platform uses an Extended Kalman Filter (EKF) [11] for state estimation. To transplant the state from the secondary controller to the primary controller, we interface directly with the EKF state representation. We leverage the state calculated by a second, previously-initialized, EKF filter to "jump start" the position estimation in our primary EKF filter. Thus, our approach does not require the full initialization of the primary filter (which would require a stationary vehicle) and can restart control software on a moving vehicle.

B. Simplex Architecture

Previous work has examined the use of dual-controller architectures to provide high assurance in various systems, including autonomous vehicles [15], [26], [44]. The Simplex architecture is one such seminal effort that combines a *simple*, feature-reduced controller with a *complex*, full-featured controller to achieve high assurance. A full-featured autonomous vehicle control stack is too complex to employ in-depth static analyses (e.g., symbolic execution [37]) or certain dynamic analyses (e.g., high-coverage fuzzing [6]), making rapid detection of vulnerabilities or aberrant behavior difficult. A feature-reduced, simplified instance of the controller software with a smaller trusted code base can provide stronger guarantees about the correct system operation. As a result, more coarse-grained checks can be applied to the complex controller and, if an issue is detected, control can change to the simple controller, allowing the platform to operate through attacks or defects. This architecture has been adapted across a number of domains, including embedded real-time operating systems [5], cyber-physical systems [46], and multi-agent systems [34]. However, the Simplex architecture is primarily focused on verification and validation in high-assurance settings—it allows changing control to a verified, simple controller, but does not provide any facility for restoring control to the complex controller during mission operation. In this paper, we augment a Simplex-architecture-based framework by providing support for restoring control to the complex controller once a patch or repair is found, thus allowing continued operation of the vehicle's mission with high assurance.

C. Software Techniques for Automated Resilience and Trust

Previous work has developed an end-to-end automated resiliency framework for automatically detecting, repairing, and recovering from exploits and faults in autonomous vehicles [15], [26]. Known as Software Techniques for Automated Resilience and Trust, or START, it combines together static and dynamic program analyses, instrumentation, and automated program repair. We provide a detailed summary of the START framework in Appendix B.

In this work, we extend the START framework to implement the Simplex architecture to support deployment on a small autonomous vehicle. We develop a dual-controller-based vehicle that divides pieces of the START framework between a complex, primary controller, and a simple, secondary controller. We also develop the algorithms necessary to restart the primary controller while the vehicle is navigating using the secondary controller. Our proposed approach also reduces how long post-repair deployment takes. While we use START in this work, our approach could be used to with any automated resilience framework such as PGPatch [21] to seamlessly integrate a repair into vehicle control software.

III. Sensor Data Transplantation

In this section, we present our algorithm for transplanting sensor data from a primary controller to a secondary controller. This algorithm is needed to successfully initialize the control software while the vehicle is in motion, improving autonomous vehicle resiliency by reducing downtime and precluding the need to stop or land the vehicle during a restart. While we present our approach in the context of redundant control software executing on two computers, this same approach may also be used for software run on the same hardware.

First, we enumerate our assumptions and provide a formal problem description. Then, we apply domain knowledge to determine the appropriate data to transplant between two controllers. Finally, we describe our algorithm for transplanting this data.

A. Hardware and Software Assumptions

First, we assume that the vehicle contains two distinct pieces of computational hardware: the primary and secondary computers. We also assume that each of these computers has access to an identical set of sensors. While the sensors are identical, we make no explicit assumptions about the sensors themselves, either the sensors that are available or the consistency of their readings across computers. For increased isolation and experimental expediency, we assume that these sensors are *not shared*, but any sensors whose values are simply polled (i.e., read-only) could be shared between the two computers[2]. We therefore assume that the trusted, secondary computer is always capable of taking control of the vehicle.

We assume that the primary state representation is a subset of the secondary state representation. In our implementation, we achieve this by using identical EKFs. This is to ensure that the secondary state is capable of completely restoring the primary state. For transplantation simplicity, we also assume that the memory layout used to store EKF state and covariance on both controllers is identical. This assumption can be relaxed by implementing a transformation system for mapping the fields of the state representation from one system to another. Other aspects of the software, such as navigation modes, may be different. We discuss techniques that can be employed to harden the secondary controller software in Section VIII.

We also assume that there is sufficient computational power for the repair task to be executed on the vehicle. Alternatively, the vehicle will need to have the ability to communicate with a server where the repair task may be offloaded.

Critically, we cannot assume that it is safe for the vehicle to stop at an arbitrary location along the path of the mission. Locations may be unsafe to stop due to physical characteristics, such as unsafe terrain, or bureaucratic considerations, such as rights of way or jurisdictions. As such, successful mission operation is dependent on the ability to remain in motion even if the control software has a fault or is attacked.

B. Problem Statement

Given these assumptions and constraints, we formulate the following problem statement:

> In the event of a detected fault or attack, control must seamlessly switch from the primary controller to the secondary controller while the vehicle is in motion. After a suitable repair is found and applied, the primary controller must successfully restart and reinitialize while the vehicle is in motion and under the control of the secondary controller. Finally, the primary controller must seamlessly regain control of the vehicle and continue with navigation.

Next, we describe techniques for restarting and initializing the control software. Section IV describes the hardware we used to provide support for seamless control transfer.

C. Transplanting Sensor Data

We leverage the redundant hardware in our design to start and initialize the control software while the vehicle is in motion. While we cannot assume that the primary controller's state is uncompromised during an attack, the secondary controller *does* have a viable, fully initialized copy of the vehicles state. Moreover, since the secondary controller has a reduced trusted code base, we can transplant its more trusted view of the state to the primary controller once a patch is generated.

Our analysis of the source code revealed that the software supports autonomous navigation as soon as the assumed error in the state falls below certain thresholds.[3] These thresholds ensure that various components of the state are consistent with each other and this requirement takes up a large portion of the static initialization time. On the secondary controller, the assumed error is already below these thresholds so we are able to immediately satisfy this condition.

Because the complete initialization of the state estimation filter is the key component allowing for autonomous control,

[2]For sensors that require both polling and updates (i.e., read-write access), we could apply a mutual exclusion mechanism to allow for shared access.

[3]The relevant source code in the ArduPilot codebase can be found here: https://github.com/ArduPilot/ardupilot/blob/e58d2ecf2f5705b375928be61bc734766690968b/libraries/AP_NavEKF3/AP_NavEKF3_Control.cpp#L618

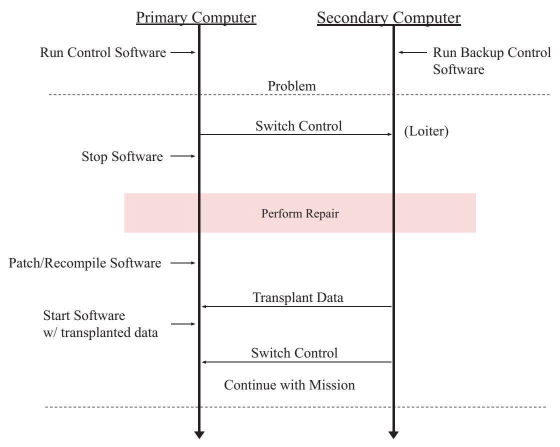

Fig. 1. Timeline of transplant algorithm. Control software runs on the primary computer until a problem is detected. Control is switched to the secondary computer while a repair is performed. Post-repair, EKF data is transplanted from the secondary computer to the primary computer and the control software is restarted. Finally, control is transferred back to the primary computer, and the mission continues.

we focus our efforts here. This means that we allow other components of the software to initialize as they normally do. Many of these components interact directly with the hardware sensors and perform the necessary handshakes to begin receiving data. When the software restarts, these handshakes are still necessary. On our platform, we only identified one hardware sensor initialization that requires a stationary vehicle—the inertial measurement unit (IMU). Our solution is to initialize the IMU at the base station (where it must be safe to remain stationary) and store the necessary initialization values. Upon moving initialization, we can load in these values instead of calculating them. This functionality is tied to a system parameter that is initially false, and can simply be set to true in the patched version.

Thus, by transplanting the secondary state, we are able to break the assumption that the vehicle is stationary during initialization.

D. Extending an Automated Resiliency Framework to Support Redundant Hardware and Seamless Switchover

Given the previous observations, we extend an automated resiliency framework to support a Simplex architecture using dual controllers. Previously, these resiliency frameworks have only been demonstrated and evaluated in simulation using real autonomous-vehicle software. To the best of our knowledge, this is the first implementation of the framework in hardware. Further, we add seamless restart and switchover to the repaired primary computer. Figure 1 shows a high-level timeline of software components, the use of the primary and secondary controllers, and transplantation of sensor data.

Before a mission begins, the control software is started on both the primary and secondary computer. The complex, primary computer is responsible for controlling the vehicle during a mission, while the simple, secondary computer is used as a backup if the resiliency framework detects a problem. If a problem is detected, our system automatically switches control to the secondary computer, which will begin to loiter (drive in a holding pattern) or drive to known safe location.

Meanwhile, the automated-repair component of the framework will begin to search for a patch to address the fault or attack. We claim no novelty in this stage. However, rather than remaining stationary while the repair takes place, our dual-controller platform is capable of continuing operation while repairing the control-software binary.

After a patch is identified and the control software is recompiled on the primary computer, it is now time to transfer control back. The secondary computer transfers a snapshot of the state of the vehicle's navigation and position, to the primary computer. Without loss of generality, this state may be represented as matrices of numeric data. The control software on the primary computer is launched, and it reads this transplanted data during initialization. The secondary computer then waits for a *ready* signal from the primary computer. When this is received, control is switched from the secondary computer back to the primary computer.

With the addition of this secondary computer and data transplantation, the vehicle can navigate at all times, even when an attack or fault has been detected.

IV. Implementation

In this section, we describe how we apply our approach to a real autonomous vehicle platform, building upon prior work that uses simulation. We develop a low-cost dual-controller Simplex rover platform to support resilient operation in small autonomous vehicles. The vehicle is built from commercial, off the shelf parts using existing control software. Our proposed design relies on the assumptions and model of mission execution described in Section III.

First, we describe the dual-controller architecture and then describe the hardware necessary to choose the active controller.

A. Dual-Controller Architecture

We extend the START framework [26] to support continued operation while the control software is repaired and restarted. We propose a redundant hardware, Simplex-style architecture with a primary and secondary computer and the ability to switch seamlessly between both. As described in Section VIII, redundant hardware and computation are well studied in this domain. Our contribution here is a small autonomous vehicle platform, built using low-cost commercial-off-the-shelf (COTS) and open-source hardware, for both deployment and study of resiliency frameworks. While a single computer would be sufficient, we prefer the physical separation afforded by redundant hardware for the primary and secondary controllers. A multi-computer platform allows for more flexibility with experimentation and research: both single and dual-computer architectures may be explored using our rover platform. Further details are provided in Appendix C.

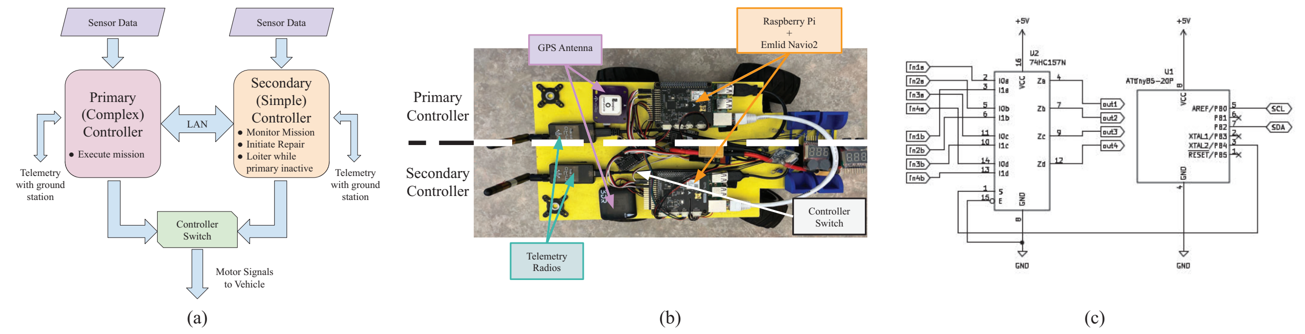

Fig. 2. (a) Block diagram of dual-controller platform. The primary controller is responsible for mission operation (navigation, motor control, etc.) while the secondary controller monitors the system and handle automated repairs. In the event of an anomaly, the controller switch will transfer control to the secondary controller, which is capable of loitering or navigating to a safe location. (b) Overhead view of dual-controller platform. The primary and secondary computers have independent sets of sensors. (c) A custom multiplexer is used to select the computer sending control signals to the motors.

Figure 2(a) provides a block diagram of our proposed architecture. We partition the operation of the vehicle (the primary controller) onto a separate, secondary, device from the controller executing the resiliency framework. This secondary computer also executes a simple controller that is capable of loitering, moving in a holding pattern, or navigating to a known safe location. This simplified controller is minimized for both a smaller attack surface and and a smaller trusted code base. Both controllers have independent sensors providing input environmental data, and both controllers can communicate with a remote operator through bidirectional telemetry radios. The controllers also have a communication link to allow monitoring of mission operation and deployment patches and repairs to the primary controller. A separate controller switch, seen in Figure 2(c), managed by the secondary controller connects the output motor signals from one of the two controllers to the vehicle.

Our platform is built on a 1/12 to 1/16 scale remote control truck chassis. Figure 2(b) provides an overhead view of the computational hardware, sensors, and vehicle platform. This design can support a variety of autopilot computers and controllers. We note that resilient cyber-physical systems are often evaluated only in simulation [26] [52] [21]. Implementing a system on hardware is difficult time consuming. We show that our system is viable on hardware, and provide a platform on which other resiliency systems can be implemented to provide a benchmark for comparison.

B. Multiplexing Motor Control Signals

Key to our platform design is the ability to programmatically switch between the motor control signals being generated by the primary and secondary computers. On a micro vehicle, these signals are typically communicated to motor controllers using *pulse-width modulation* (PWM) [2]. Motor speeds are specified with a varying width of a electrical pulse over a fixed time window, with typical pulse widths ranging from 1ms (no throttle) to 2ms (full throttle). With a dual-controller platform, there are two sets of PWM signals being generated—one from each computer—and hardware is needed to select and route these signals to the motor controllers and servos.

We use an Ateml ATTiny85 microcontroller to switch the multiplexer. Our custom firmware exposes a device on an I^{2}C bus, which is a common serial protocol supported by most COTS autopilot hardware. Using an I^{2}C bus, the multiplexer is connected to the secondary computer. The secondary computer therefore has the ability to switch the control of the motors through this microcontroller.

A schematic of the multiplexer circuit is provided in Figure 2(c). The multiplexer is on the left, and the microcontroller is on the right. Input, output, and bidirectional signals are labeled. The `SCL` and `SDA` signals form the I^{2}C bus (representing clock and data, respectively). Circuitry for the pin headers is elided for space. Our firmware for the microcontroller is available as open-source software and the full circuit designs and circuit-board layout are released as open hardware.

V. Experimental Methodology

Next, we describe the hardware and software configuration used for our experiments. We also outline the benchmarks used to evaluate the transfer of control between primary and secondary computers.

A. Control Software

The autonomous vehicle control software used in our evaluation is ArduPilot [2]. This control software has been deployed to over one million devices and supports a wide range of vehicle types [3]. For this evaluation, we focus specifically on APMRover2, a subset of ArduPilot designed for autonomous ground vehicles. We added our code to transplant EKF state to APMRover2 version 4.1.0 (commit e58d2ec). ArduPilot is controlled using the Micro Autonomous Vehicle Link (MAVLink), a packet-based protocol designed for small vehicles.[4] Therefore, we implement our control messages to trigger transplants on top of this protocol.

B. Hardware and System Configuration

Our hardware platform uses two Raspberry Pi 3B+ computers, each with an Emlid Navio2 autopilot HAT. We run a wired network between the two Pis for transfer of EKF data. The two

[4]MAVLink Protocol: https://mavlink.io/en/

computers are connected to a *ground station* with both WiFi and also MAVLink telemetry. WiFi was used to manage the experiments and data collection, but actual mission commands are sent exclusively using the MAVLink telemetry radios.

Both computers are running a custom Raspberry Pi OS (debian-based) image created by Emlid with a Linux kernel version of 4.14.95-emlid-v7+. This is a customized kernel that contains the necessary drivers to interface with the sensors on the Navio2 autopilot HAT.

See Appendix C for more information about parts and our design choices for the hardware platform.

C. Attack and Fault Scenarios

To test that our dual-controller vehicle is capable of switching between controllers when an attack or fault is detected, we replicate the results presented by Leach et al [26]. Their approach was originally tested in simulation on 14 *scenarios* including attacks and realistic defects designed by an external red team [26]. The red team developed an indicative set of vulnerabilities that would likely be encountered in the wild for autonomous vehicle systems. This previous work, however, only tested these scenarios in a simulation framework. We take these vulnerabilities as written from the START framework, and we integrated them with our dual-controller rover to evaluate the efficiency with which our approach can detect vulnerabilities, switch control, patch indicative defects, and return control to the patched, primary controller.

The first two columns of Table II provide a high-level description of each scenario. Scenario 8 is an x86 (Intel) attack, which we exclude because it is not applicable to our Arm-based platform.

The scenarios developed in Leach et al. consist of (1) ArduPilot source code with the seeded defect, (2) a waypoint mission to execute with the provided software, and (3) a script to trigger the seeded vulnerability through a MAVLink message. The original experiment scenarios were developed for the ArduCopter subcomponent of ArduPilot. We therefore reconstructed the scenarios described by Leach et al. and adapted them to work on the APMRover2 subcomponent. A majority of these scenarios were implemented on the underlying hardware abstraction layer and were therefore common to both control components. We also ported these scenarios to the hardware abstraction layer for the Navio2 HAT as there is dedicated code for both simulation and physical hardware. For all scenarios, we ported the defects and vulnerabilities as faithfully as we could. We also updated the missions to use waypoints in our geographic location, but made an effort to maintain the orientation and overall structure of the mission.

VI. Evaluation

In this section we present an empirical evaluation of our sensor data transplantation algorithm and the dual-controller platform.

We address the following research questions:

RQ 1. Does transplanting sensor data reduce the initialization time for a stationary vehicle?

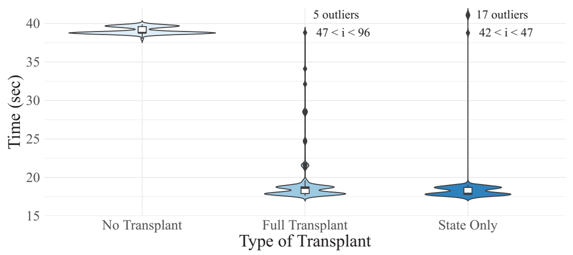

Fig. 3. Time to initialize ArduPilot on a stationary vehicle across 100 trials with no transplant, with full transplant, and only transplanting the state vector. With no transplanting, initialization takes an average of 39.081 seconds. With full transplantation, initialization takes an average of 21.482 seconds. Transplanting only the state vector from the EKF takes an average initialization of 23.376 seconds.

RQ 2. Is the control software able to initialize a moving vehicle?

RQ 3. Does transplanting a subset of EKF data improve initialization times for a moving vehicle?

RQ 4. Is the transplant algorithm robust to perceived error in the transplanted state?

RQ 5. Is the dual-controller platform able to continue operation with a control switchover?

RQs 2 and 5 specifically test the functionality of our platform. Without our approach, the current START framework must stop the control software, leaving the vehicle stationary and stranded for some time. The current version of ArduPilot is not able to initialize while the vehicle is in motion, which is required by our dual-controller platform. RQs 1, 3, and 4 measure ancillary aspects of our system, such as improvements to run time. Note, however, that we consider the system a success if (1) it provides continuous vehicle operation, even while START is constructing repairs on the control software and (2) the control software is able to be restarted while the vehicle is being controlled and in motion.

A. Initialization on a Stationary Vehicle (RQ 1)

First, we test that transplanting EKF data still allows the vehicle to *initialize* while stationary. We wish to validate that our data transplantation does not significantly impede basic operation of the control software. Thus, we measure the time needed to initialize ArduPilot with and without the EKF transplant algorithm. We define initialization time as the time needed for the software to be ready to drive autonomously. Specifically, the software is initialized if it would allow the user to switch it into autonomous mode. The two key data structures in the EKF are the state vector and the covariance matrix. We therefore focus our transplantation algorithm on these two collections of data. We conducted several experiments that tested the effectiveness of transplanting various subsets of this data between the computers. A full transplant copies both the state vector and the covariance matrix, while our second configuration only copies the state vector. Each configuration was timed across 100 trials.

TABLE I
AVERAGE AND MEDIAN TIME NEEDED TO INITIALIZE ON A MOVING VEHICLE

Experiment	Average Init. Time (s)	Median Init. Time (s)	Success
No Transplant[†]	–	–	–
Full Transplant	47.196	29.524	9/10*
State Only	42.899	24.736	10/10

[†] Without our modifications, ArduPilot cannot initialize in motion
* One trial failed to initialize for an unrelated hardware fault

Figure 3 presents the results of these trials. The control software successfully initialized in all trials for which EKF data was transplanted. The violin chart provides a visualization of the density of results of varying duration. Most recorded values are clustered around the mean for each trial. An unmodified version of ArduPilot takes an average of 39.081 seconds to initialize. With Full Transplantation and State Vector Only, initialization of a stationary rover takes 21.482 and 23.376 seconds, respectively. Lower times are better, indicating that ArduPilot was ready to perform autonomous missions more quickly. This reduced time for the vehicle to be ready for autonomous control is an added benefit of our dual-controller platform. Micro autonomous vehicles have limited battery life (20-30 minutes [7]), and thus any reduction in initialization time post-repair will extend the length of a mission.

While most initialization times hover right around the mean, there are a number of outliers when the EKF data is transplanted. When these occur, we observed that the EKF had entered a reset mode, often because of a glitch in the sensor data. In several instances, this resulted in an initialization time slightly longer that the unmodified version of ArduPilot; however, we note that without sensor data transplantation, it will not be possible to initialize ArduPilot in motion.

Transplanting EKF data still allows a stationary vehicle to initialize. An added benefit of our approach is a 40-45% reduction in the initialization time of ArduPilot on a stationary vehicle.

B. Initialization on a Moving Vehicle (RQ 2 & 3)

Next, we test whether state transplantation supports controller initialization while the vehicle is in motion. Recall from Section III that we assume that the software cannot initialize while the vehicle is in motion, and this is true for ArduPilot. As noted in Section I, it is critical that the vehicle remain in motion at all times to reduce the risk of physical attack.

Additionally, we measure the time needed to initialize ArduPilot on a moving vehicle. To collect this data, we drove the vehicle in a circle using the secondary computer while timing the initialization of the primary computer. We tested transplanting both the full state data structure as well as only the state vector. Because the vehicle is in motion for these experiments, the testing procedure had to be monitored. Therefore, we limited our testing to 10 trials each. We present

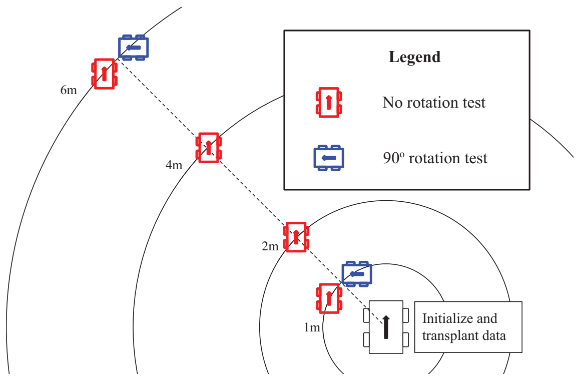

Fig. 4. Experiment setup to test acceptable error in transplanted EKF data. The vehicle was initialized before exporting data. To simulate the vehicle travelling some distance before the transplanted data is injected, the vehicle was then moved 1, 2, 4, and 6 meters away from the initialization point and restarted with the transplanted data. We recorded the time needed for the control software to be ready for autonomous operation. We also tested rotating the vehicle at 1 and 6 meters to demonstrate resilience to orientation.

the results in Table I. Critically, state transplantation allows for a successful ArduPilot initialization in all of our experiments. In one case for the full transplant, the hardware failed for an unrelated reason resulting in a total of nine successful trials. While in-motion restarts are slower than stationary restarts (cf. Figure 3, in-motion restart was not possible on this platform prior to our technique. Therefore, we have demonstrated that sensor data transplantation is a successful approach for initializing ArduPilot while the vehicle is in motion.

We also study the effects of transplanting a subset of data rather than the full EKF state. In comparing a full transplant with transplanting only the state vector portion of the EKF, we observe an inversion of results from those presented in Figure 3 for a stationary vehicle. There is a 16% difference in the median initialization time for these two configurations. We attribute this to covariance—the other data structure in the EKF—being more a property of the individual hardware. Thus it seems faster to recalculate it from scratch rather than starting from the covariance on different hardware.

Similar to the stationary experiments, we observed EKF resets that resulted in longer initialization times. These lasted longer when the vehicle was in motion but eventually cleared. This suggests an opportunity for improving the initialization times of a vehicle in motion.

Transplanting EKF data allows ArduPilot to initialize successfully while the vehicle is in motion. Transplanting only the state vector improves the median initialization time by approximately 16%.

C. Error in Transplanted Data (RQ 4)

Because the vehicle is in motion while the EKF data is being transplanted, there is a small difference or error in the state by the time ArduPilot begins to initialize. We measure how tolerant initialization is to discrepancies in the transplanted

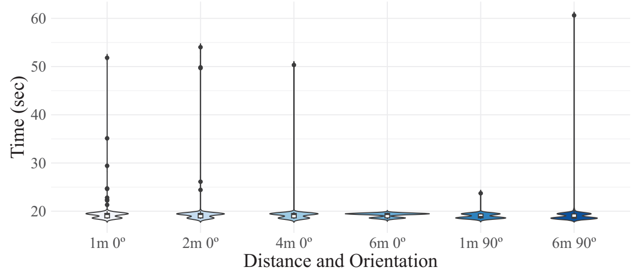

Fig. 5. Average runtime to initialize ArduPilot when stationary vehicle is placed at varying distances and orientations from where EKF data was collected. Each location was measured 100 times. There is no evidence that initialization times are a function of the distance or orientation from the origin point where EKF data was transplanted.

data. Figure 4 provides an overview of our experiments. We first initialized ArduPilot and exported the EKF data to use for transplantation. Then, we moved the vehicle to set distances from this initial location. To increase the challenge of initialization, we move the vehicle at a diagonal to the direction the vehicle was oriented. Keeping the vehicle oriented in the same direction, we measured initialization time with the transplanted data at one, two, four, and six meter intervals from the original location. Six meters exceeds the distance a vehicle can travel in the time it takes to transfer EKF data between controllers, but we provide these larger distances as stress tests for our approach. We also tested with the vehicle rotated by 90 degrees at both one and six meters.

Results from these experiments are presented in Figure 5. For each position of the vehicle, we conducted 100 trials to record the initialization time. The violin charts allow us to observe that initialization times continue to be clustered around the mean for all configurations. In each case, the average initialization time is approximately 20 seconds with no discernible differences in recorded times. This suggests that the initialization procedures of ArduPilot are fairly robust and are capable of initializing with EKF data within a distance of at least six meters. For small vehicles, this provides a sufficient radius to allow for the copying of data between the two computers while the vehicle is in motion.

The distance traveled by the vehicle while EKF data is being transplanted has no discernible impact on ArduPilot's ability to initialize. Initialization time are consistent across all measured distances and orientations.

D. Control Switchover (RQ 5)

To test our dual-controller hardware, we replicate the experimental setup and evaluation scenarios used in the START framework [26]. We focus on our platform's ability to transfer control from the primary computer to the secondary computer when START detects an anomaly, and critically, to transfer control back to the primary controller after a repair has been constructed. Other stages of the START framework (such as the ability to repair a fault) have been previously evaluated for the same scenarios, and these results would not change substantially between simulation- and real-world deployments.

TABLE II
SUCCESSFUL HARDWARE SWITCHOVER ON AUTONOMOUS VEHICLE FAULT SCENARIOS

#	Scenario Description	Successful Switch
1	Use after free	✓
2	Format string: Information Leak	✓
3	Format string: Crash	✓
4	Stack-based buffer overflow: GCS	✓
5	Heap-based buffer overflow: GCS	✓
6	Stack-based buffer overflow: MAVLink	✓
7	Heap-based buffer overflow: MAVLink	✓
8	x86 code injection†	N/A
9	Infinite loop	✓
10	Segmentation fault	✓
11	Mathematical logic bug	✓
12	Denial of Service (DoS)	✓
13	Integer error	✓
14	Floating point exception	✓

† This scenario is specific to Intel assembly and cannot be replicated on our hardware, which is based on 64-bit ARM.

We programmed the secondary controller to send an I^2C message to the onboard multiplexer to switch control when an anomaly was detected. On the secondary computer, we programmed ArduPilot to drive the rover to a specified coordinate, allowing us to test for a seamless transfer of control. We define a seamless transfer as one where there is no pause or noticeable change in the vehicle's speed or trajectory.

Table II shows results of these experiments. The final column indicates whether the switch of control was seamless. In all scenarios, our approach successfully and seamlessly transferred control from one computer to the other. For each experiment, the control transfer took less than 10 ms.

Our hardware multiplexer correctly switches control between primary and secondary controllers for all 13 scenarios that apply to Arm-based computers. When combined with the results for EKF transplant, our dual-controller architecture improves autonomous vehicle resiliency and repair.

VII. DISCUSSION

In this section, we discuss the implications of this work as well as threats to validity of our approach.

Our evaluation demonstrates that automated resiliency frameworks can apply to physical vehicle platforms and that transfers of control can be made seamlessly between hardware and software control stacks. This approach assumes that a dual-controller architecture is employed. We use this setup because it facilitates isolation of execution between the primary and secondary control boards. In practice, this means that, although the rich, full-featured primary control software may become compromised by an attacker, the simplified, reduced feature secondary controller is less likely to be compromised because of its reduced-trusted code base and attack surface.

A. *Generalizability of Results*

In this paper, we focus on an extension of the START framework for autonomous vehicle resiliency. However, while our prototype currently only support this framework, the key algorithmic technique concerning the seamless transfer of sensor state is not limited to a specific automated resiliency framework. In practice, our work could be combined with other software assurance techniques to enable more efficient recovery of autonomous vehicles from defects or faults. Generalizing our findings to other platforms would require an assessment of the representation of sensor state and an ability to capture and transplant that state in a new target platform.

VIII. Related Work

Checkpoint and Restore methods are a means for introducing fault tolerance into a system [41]. In this approach, a copy of vital data is stored at specific points during the mission (or, more generally, during execution of any software). Checkpointing can be embedded directly into the code base. This method was used by Selective Checkpoint/Restore (SCR), an algorithm that combines a type-guided search with developer annotations to embed Checkpoint/Restore support into autonomous vehicle control software [18]. We present a complementary approach that leverages domain knowledge of the controller software and architecture to transplant a small portion of state from a simple controller to the complex controller, allowing reinitialization while the vehicle is in motion. Our approach could be combined with a Checkpoint/Restore system to add an additional layer of redundancy.

Hardware and Software Redundancy are well-established strategies for enhancing software and hardware resilience [9], [32], with applications spanning spacecraft [14], satellites [33], web systems [31], and embedded systems [16]. Hot spares/hot standbys [17], where an actively initialized secondary controller takes over in the event of a primary controller failure, offer both full duplication and simpler safety controller options. Software-based redundancy can be achieved through N-version systems [51], which run multiple independently developed program versions and utilize a voting system to combine their results. We capitalize on hardware redundancy, employing dual controller computers to enable a vehicle to maintain autonomous operation via secondary hardware while repairing software defects and vulnerabilities in the primary hardware. While our approach primarily focuses on hardware redundancy, it is conceivable to implement an N-version system on top of our proposed platform.

Sensor Attacks are a class of attack that seeks to subvert or disrupt the ability of the vehicle to measure the world around it. Examples of this include electromagnetic interference injection attacks [19] and acoustic injection attacks [20]. These attacks are capable of interference with sensor measurements, such as those taken by onboard *inertial measurement units* or IMUs. In this scenario, the IMUs of all onboard computers would be simultaneously compromised. Although our system is largely vulnerable to these attack models, there have been shown to be effective counter measures to some classes of sensor attacks [19] [20] [30]. Our technique focuses primarily on protecting the *software* rather than the *hardware*. However, in practice, our proposed architecture would benefit from the hardening techniques to mitigate the sensor attacks.

Trusted Execution Environments (TEEs), such as Arm Trust-Zone [4], offer the possibility of implementing the secondary controller functions, providing robust isolation guarantees with a single physical controller. A shared platform could also use **virtualization**, employing separate guest virtual machines or containers for each control stack while allowing the hypervisor to manage I/O pathways to sensors and actuators. However, these approaches come with security concerns, as attackers could compromise the hypervisor or escape the virtualized environment [27], [39], [40], posing a threat to the secondary controller VM. Furthermore, both TEEs and virtualization have drawbacks. They require careful consideration of sensor design, as sharing sensors between primary and secondary control software may expose vulnerabilities like sensor-based attacks. Addressing this issue might involve enabling switchover hardware to switch between physical sensors. Given that autonomous vehicles are real-time systems, any assurance techniques must not introduce excessive overhead that violates real-time constraints. Introducing a secondary controller VM or context switching within the TEE environment could potentially introduce too much runtime overhead, jeopardizing the safe operation of the vehicle.

IX. Conclusions

Uncrewed, autonomous vehicles are being integrated into numerous industries. While there are many benefits, latent software defects and vulnerabilities in the controller source code pose a real threat to the success of these platforms. Recent efforts, such as the START framework, seek to increase the resiliency of autonomous vehicles through binary hardening, runtime monitoring, and automated program repair, but do not support continuous mission operation while defects are being repaired, and have not yet been applied to real physical vehicle platforms. In this work, we develop a full hardware implementation of the previously theoretical START framework. We present a Simplex-style architecture using redundant, dual computers on a small vehicle platform to extend the START framework to support continuous, in-motion operation and recovery. To allow the primary controller to be restarted while the vehicle is under the control of the secondary controller, we present a technique for transplanting sensor data between software controllers. This allows the control software to skip over initialization steps that require the vehicle to be stationary, thereby increasing efficiency and physical safety of the vehicle during recovery. We perform an empirical evaluation of our dual-controller architecture and data transplantation algorithm, demonstrating that our design supports continuous operation and restart while the vehicle is in motion. Our approach has the added benefit of reducing controller initialization time by an average of 40% compared to the original, stationary initialization times.

References

[1] O. Alhawi, M. Mustafa, and L. Cordeiro, "Finding security vulnerabilities in unmanned aerial vehicles using software verification," in *IEEE International Workshop on Secure Internet of Things (SIoT)*, Aug. 2019.

[2] ArduPilot Development Team, "ArduPilot," https://ardupilot.org, 2023.

[3] ——, "ArduPilot Source Code," https://github.com/ArduPilot, 2023.

[4] ARM, "TRUSTZONE FOR CORTEX-A," https://www.arm.com/technologies/trustzone-for-cortex-a, 2022.

[5] S. Bak, D. K. Chivukula, O. Adekunle, M. Sun, M. Caccamo, and L. Sha, "The system-level simplex architecture for improved real-time embedded system safety," in *RTAS*, 2009, pp. 99–107.

[6] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *ACM CCS*, 2017, pp. 2329–2344.

[7] C. H. Choi, H. J. Jang, S. G. Lim, H. C. Lim, S. H. Cho, and I. Gaponov, "Automatic wireless drone charging station creating essential environment for continuous drone operation," in *ICCAIS*, 2016, pp. 132–136.

[8] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: a secretless framework for security through diversity," in *USENIX - Volume 15*, 2006.

[9] S. Dutt, F. Rota, F. Trovo, and F. Hanchek, "Fault tolerance in computer systems—from circuits to algorithms," in *The Electrical Engineering Handbook*. Burlington: Academic Press, 2005, pp. 427–457.

[10] H. Fang, M. A. Haile, and Y. Wang, "Robust extended kalman filtering for systems with measurement outliers," *TCST*, vol. 30, no. 2, 2022.

[11] P. Frogerais, J.-J. Bellanger, and L. Senhadji, "Various ways to compute the continuous-discrete extended kalman filter," *TACON*, vol. 57, 2012.

[12] M. F. Garrett, *Master Minimum Equipment List: Boeing B-737 100/200/300/400/500/600/700/800/900*, Federal Aviation Administration, Renton, WA, USA, May 2010.

[13] W. Hawkins, J. D. Hiser, A. Nguyen-Tuong, J. W. Davidson *et al.*, "Securing binary code," *Security & Privacy*, vol. 15, no. 6, 2017.

[14] C. Hersman and K. Fowler, "Chapter 5 - best practices in spacecraft development," in *Mission-Critical and Safety-Critical Systems Handbook*, K. Fowler, Ed. Boston: Newnes, 2010, pp. 269–460.

[15] K. Highnam, K. Angstadt, K. Leach, W. Weimer, A. Paulos, and P. Hurley, "An uncrewed aerial vehicle attack scenario and trustworthy repair architecture," in *DSN-W*, 2016, pp. 222–225.

[16] A. Höller, T. Rauter, J. Iber, and C. Kreiner, "Towards dynamic software diversity for resilient redundant embedded systems," in *SERENE*. Cham: Springer International Publishing, 2015, pp. 16–30.

[17] Y. Hu, X. Long, and C. Wen, "Virtual machine based hot-spare fault-tolerant system," in *ICISS*, 2009, pp. 429–432.

[18] Y. Huang, K. Angstadt, K. Leach, and W. Weimer, "Selective symbolic type-guided checkpointing and restoration for autonomous vehicle repair," in *ICSEW*, 2020, p. 3–10.

[19] J. Jang, M. Cho, J. Kim, D. Kim, and Y. Kim, "Paralyzing drones via emi signal injection on sensory communication channels," in *NDSS*, 2023.

[20] J. Jeong, D. Kim, J. Jang, J. Noh, C. Song, and Y. Kim, "Un-rocking drones: Foundations of acoustic injection attacks and recovery thereof," in *NDSS*, 2023.

[21] H. Kim, M. O. Ozmen, Z. B. Celik, A. Bianchi, and D. Xu, "PGPatch: Policy-guided logic bug patching for robotic vehicles," in *SP*, 2022.

[22] J. Kim, S. Kim, C. Ju, and H. I. Son, "Unmanned aerial vehicles in agriculture: A review of perspective of platform, control, and applications," *Ieee Access*, vol. 7, pp. 105 100–105 115, 2019.

[23] A. Konert and T. Balcerzak, "Military autonomous drones (UAVs) - from fantasy to reality. legal and ethical implications." *Transportation Research Procedia*, vol. 59, pp. 292–299, 2021.

[24] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *ICSE*, 2012, pp. 3–13.

[25] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *TSE*, vol. 38, 2012.

[26] K. Leach, C. S. Timperley, K. Angstadt, A. Nguyen-Tuong, J. Hiser, A. Paulos, P. Pal, P. Hurley, C. Thomas, J. W. Davidson, S. Forrest, C. Le Goues, and W. Weimer, "START: A framework for trusted and resilient autonomous vehicles (PER)," in *ISSRE*. IEEE Computer Society, 2022.

[27] K. Leach, F. Zhang, and W. Weimer, "Combining secure guard extensions and system management mode to monitor cloud resource usage," in *RAID*, 2017.

[28] F. S. Leira, T. A. Johansen, and T. I. Fossen, "A uav ice tracking framework for autonomous sea ice management," in *ICUAS*, 2017.

[29] J. Li, X. Wei, and G. Zhang, "An extended kalman filter-based attitude tracking algorithm for star sensors," *Sensors*, vol. 17, no. 8, 2017.

[30] M. Liu, L. Zhang, P. Lu, K. Sridhar, F. Kong, O. Sokolsky, and I. Lee, "Fail-safe: Securing cyber-physical systems against hidden sensor attacks," in *RTSS*, 2022, pp. 240–252.

[31] N. Looker, M. Munro, and J. Xu, "Increasing web service dependability through consensus voting," in *COMPSAC*, vol. 2, 2005, pp. 66–69.

[32] A. Mattavelli, "Software redundancy: What, where, how," Ph.D. dissertation, USI Università della Svizzera italiana, 2016.

[33] I. V. McLoughlin and T. R. Bretschneider, "Reliability through redundant parallelism for micro-satellite computing," *ACM Trans. Embed. Comput. Syst.*, vol. 9, no. 3, 2010.

[34] U. Mehmood, S. Roy, A. Damare, R. Grosu, S. A. Smolka, and S. D. Stoller, "A distributed simplex architecture for multi-agent systems," *Journal of Systems Architecture*, p. 102784, 2022.

[35] S. M. S. Mohd Daud, M. Y. P. Mohd Yusof, C. C. Heo, L. S. Khoo, M. K. Chainchel Singh, M. S. Mahmood, and H. Nawawi, "Applications of drone in disaster management: A scoping review," *Science & Justice*, vol. 62, no. 1, pp. 30–42, 2022.

[36] A. Nguyen-Tuong, D. Melski, J. W. Davidson, M. Co, W. Hawkins, J. D. Hiser, D. Morris, D. Nguyen, and E. Rizzi, "Xandra: An Autonomous Cyber Battle System for the Cyber Grand Challenge," *Security & Privacy*, vol. 16, no. 2, pp. 42–51, 2018.

[37] S. Poeplau and A. Francillon, "Symbolic execution with {SymCC}: Don't interpret, compile!" in *USENIX Security*, 2020, pp. 181–198.

[38] G. Rigatos and S. Tzafestas, "Extended kalman filtering for fuzzy modelling and multi-sensor fusion," *Mathematical and Computer Modelling of Dynamical Systems*, vol. 13, no. 3, pp. 251–266, 2007.

[39] J. Rutkowska, "Red Pill," http://www.ouah.org/Red_Pill.html.

[40] ——, "Blue Pill," http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html, 2006.

[41] J. C. Sancho, F. Petrini, K. Davis, R. Gioiosa, and S. Jiang, "Current practice and a direction forward in checkpoint/restart implementations for fault tolerance," in *IPDPS*, 2005.

[42] N. Schiller, M. Chlosta, M. Schloegel, N. Bars, T. Eisenhofer, T. Scharnowski, F. Domke, L. Schönherr, and T. Holz, "Drone security and the mysterious case of dji's droneid," in *NDSS*, 2023.

[43] R. Schneider and C. Georgakis, "How to not make the extended kalman filter fail," *Industrial & Engineering Chemistry Research*, vol. 52, no. 9, pp. 3354–3362, 2013.

[44] D. Seto, B. Krogh, L. Sha, and A. Chutinan, "The simplex architecture for safe online control system upgrades," in *ACC (IEEE Cat. No.98CH36207)*, vol. 6, 1998, pp. 3504–3508 vol.6.

[45] J. Vargas, S. Alsweiss, O. Toker, R. Razdan, and J. Santos, "An overview of autonomous vehicles sensors and their vulnerability to weather conditions," *Sensors*, vol. 21, no. 16, 2021.

[46] P. Vivekanandan, G. Garcia, H. Yun, and S. Keshmiri, "A simplex architecture for intelligent and safe unmanned aerial vehicles," in *RTCSA*, 2016, pp. 69–75.

[47] D. Wang, S. Li, G. Xiao, Y. Liu, and Y. Sui, "An exploratory study of autopilot software bugs in unmanned aerial vehicles," in *ESEC/FSE*. ACM, 2021, p. 20–31.

[48] X. Wang and E. E. Yaz, "Second-order fault tolerant extended kalman filter for discrete time nonlinear systems," *TACON*, vol. 64, no. 12, 2019.

[49] S. Wen, Z. Cai, and X. Hu, "Constrained extended kalman filter for target tracking in directional sensor networks," *IJDSN*, vol. 11, 2015.

[50] Y. Xia, Y. Liu, H. Chen, and B. Zang, "Cfimon: Detecting violation of control flow integrity using performance counters," in *DSN*, 2012.

[51] M. Xie, C. Xiong, and S.-H. Ng, "A study of n-version programming and its impact on software availability," *IJSS*, vol. 45, no. 10, 2014.

[52] L. Zhang, P. Lu, F. Kong, X. Chen, O. Sokolsky, and I. Lee, "Real-time attack-recovery for cyber-physical systems using linear-quadratic regulator," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 5s, sep 2021.

[53] Y. Zhang, T. Liu, H. Zhao, and C. Ma, "Risk analysis of can bus and ethernet communication security for intelligent connected vehicles," in *AIID*. IEEE, 2021, pp. 291–295.

Appendix

A. Extended Kalman Filters

For correct operation in the presence of sensor measurement noise, autonomous vehicles must maintain an accurate model of position, orientation, and other metrics, which we collectively refer to as the vehicle's *state*. The algorithm used by our target software platform is the Extended Kalman Filter (EKF) [11]. EKFs have been used in a wide variety of applications, including altitude tracking for star sensors [29], target tracking in directional sensor networks [49], and autonomous vehicles [38].

An EKF blends sensor measurements with a mathematical model to get the best approximation of the actual state of the vehicle. Sensor measurements are known to be affected by random noise, introduced through ambient environment, other vehicles or electromagnetic sources, or manufacturing idiosyncrasies. Thus, a running estimate of the state to compare against means that no one sensor measurement needs to be fully trusted.

The high level estimation pipeline for an EKF is depicted in Figure 6. Since the vehicle is in motion, the state will change between each time step. Thus, the EKF first applies a mathematical model to the existing state to update the estimate to account for this motion. At this point, the estimate and the sensor measurements are compared and used to calculate a covariance matrix. This covariance matrix is then used to calculate the Kalman *gain* for the given time step. The gain determines the degree to which the sensor measurement or the running estimate should be favored in calculating the actual state of the vehicle. The state is updated with a weighted combination of the sensor data and the state estimate.

This process then repeats on every time step. Thus, the result is a highly accurate calculation of the vehicle's state at all times, even if there are spurious readings from sensors. When properly tuned, EKFs have been demonstrated to be robust to varying degrees of sensor error and accuracy [10], [43], [48].

Upon initialization, the EKF is typically seeded with random values for the state. The vehicle is required to remain stationary while the state values converge. After the assumed error calculated from the covariance falls below a certain threshold, the vehicle can be operated.

In this work, we leverage the state calculated by a second, already initialized, EKF filter to jump start the position estimation in our primary EKF filter. By doing this, our approach does not require the full initialization of the primary filter and can restart the control software on a moving vehicle.

B. Software Techniques for Automated Resilience and Trust

A recent body of work developed by Leach et al. is the START framework START takes a program as input—in this case the vehicle control software—as well as a corresponding mission plan to execute. Prior to mission execution, it builds a *trust envelope* to characterize the expected runtime behavior (e.g., maximum vehicle speed, expected GPS locations, etc.) by simulating the mission plan over multiple trials to build a distribution of expected sensor readings over time. This trust envelope is then used for runtime monitoring.

START uses a binary rewriting engine that automatically rewrites a vehicle control stack with security primitives such as stack canaries, instruction randomization, and instrumentation that detects memory corruption attacks. Missions are executed using this hardened control software, rather than the original provided to START. While the hardened control software is executed to actuate motors and move the vehicle through

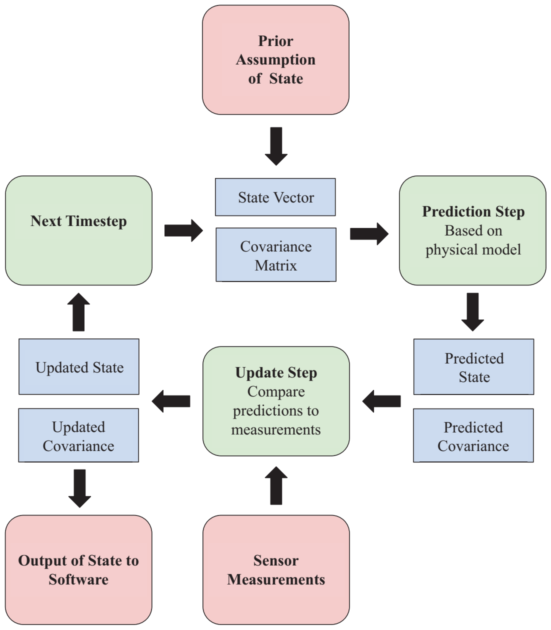
Prior Assumption of State
State Vector
Covariance Matrix
Next Timestep
Prediction Step Based on physical model
Predicted State
Predicted Covariance
Update Step Compare predictions to measurements
Updated State
Updated Covariance
Output of State to Software
Sensor Measurements

Fig. 6. Abstraction of an Extended Kalman Filter (EKF). Sensor data is fused with estimates of vehicle location to produce highly accurate knowledge of the vehicle's state. The EKF encodes this information in a *state vector* and a *covariance matrix*.

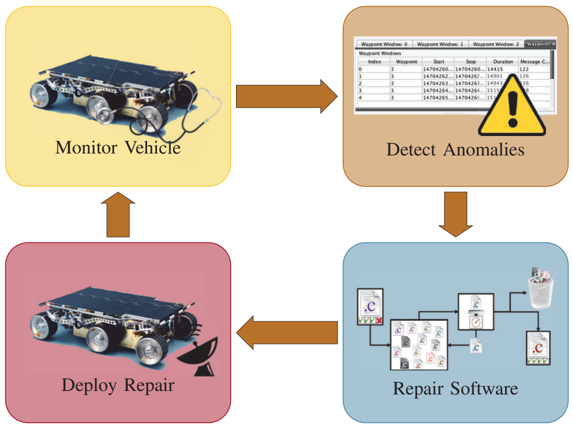

Fig. 7. START Framework Monitor-Detect-Repair-Deploy Loop. A hardened binary is deployed to the vehicle for autonomous operation. During vehicle operation, START monitors signals for signs of anomalies. An automated repair algorithm searches for a patch that mitigates the anomaly, and the updated software is deployed to the vehicle.

TABLE III
PARTS LIST FOR DUAL-CONTROLLER AUTONOMOUS VEHICLE. PRICES LISTED ARE ACCURATE AS OF DECEMBER 2022.

Item	**Quantity**	**Price (USD)**
1/12 to 1/16 scale 4-wheel-drive RC truck	1	$60
2S 20C LiPo Battery Pack	2	$20
Raspberry Pi 3 B+	2	$70
Emlid Navio2 Autopilot	2	$400
SiK Telemetry Radio V2 Pair	2	$154
u-Blox Neo-M8N GPS	2	$144
Multiplexer Board	1	$20
Optional Radio Controllers		
RC Controller	2	$140
RC Transmitter	2	$152
RC Receiver	2	$64
Total		**$1,224**
Total (without optional parts)		**$868**

a given mission, an anomaly detection engine monitors the vehicle by comparing real-time signals to the previously trained trust envelope. If an attacker compromises the primary controller via the network, or a software fault is encountered, the framework shuts down the control software and deploys an automated program repair task. The goal of this repair task is to produce a variant of the control software that does not exhibit the same fault behavior. The repair task is guided by an on-board simulation that randomly mutates the controller software and checks if the same network inputs would lead the mission to fail. Once a repair has been constructed, the software is re-flashed on the platform, and the mission is restarted. The monitor-detect-repair-deploy loop implemented by START is depicted in Figure 7.

The START framework was originally evaluated in simulation on a number of attack scenarios developed in a red team exercise. In simulation, the framework was successful in continuing a mission in over 85% of the scenarios. However, the original results do not measure the feasibility of START on an actual autonomous vehicle, and did not provide a detailed basis for physically deploying a secondary control stack on which to execute the automated program repair task. Critically, START also requires landing or stopping the vehicle while finding and deploying the repair, leaving a large window for physical attacks (against the vehicle platform) and precluding rapid responses to attacks.

C. Hardware

Our platform is built on a 1/12 to 1/16 scale remote control truck chassis. This design can support a variety of autopilot computers and controllers; however, to provide the necessary computational power for onboard software monitoring and repair, we use single-board Raspberry Pi computers with Emlid Navio2 autopilot Hardware Attached on Top boards (HATs). Through our testing, we found that the Navio2 provides a good selection of sensors and has a low failure rate. Unfortunately, the built-in GPS does not provide suitable accuracy on an RC car platform, so we have also added external GPS units to our design. The software controllers are configured to use these external units rather than the lower-accuracy built-in modules.

Table III provides a high-level parts list for our autonomous vehicle platform. The vehicle can be built for under 1,000 US Dollars and adding optional RC controllers for manual control can be done for under 1,500 US Dollars.

Our prototype implementation uses a crossover Ethernet connection between the two controllers, which is desirable for ease of data and file transfer in testing environments, but exposes a potential attack surface [53]. With a security-critical deployment, a hardened, shielded serial connection may be more desirable. With this proposed connection, the secondary computer is only monitoring the primary controller and restoring its state in the event of a failure. Therefore the secondary computer cannot be tainted by a compromised primary computer. An attacker could also possibly subvert the monitoring process of START by sending falsified data to the secondary computer. Although this is possible in theory, Leach et al. found that in red team exercises, this was not an indicative attack [26]. Further, this is an inherent limitation of the START framework and falls outside of the scope of this work, which aims to adapt the START framework to a real hardware platform. Our prototype demonstrates the viability of our approach, especially given our threat model that considers an attacker using crafted packets over a wireless channel.