START: A Framework for Trusted and Resilient Autonomous Vehicles (Practical Experience Report)

Kevin Leach[*] Vanderbilt University Nashville, USA

Christopher S. Timperley[*] Carnegie Mellon University Pittsburgh, USA

Kevin Angstadt St. Lawrence University Canton, USA

Anh Nguyen-Tuong University of Virginia Charlottesville, USA

Jason Hiser University of Virginia Charlottesville, USA

Aaron Paulos Raytheon BBN Boston, USA

Partha Pal Raytheon BBN Boston, USA

Patrick Hurley Griffiss Institute Rome, USA

Carl Thomas Air Force Research Laboratory Rome, USA

Jack W. Davidson University of Virginia Charlottesville, USA

Stephanie Forrest Arizona State University Phoenix, USA

Claire Le Goues Carnegie Mellon University Pittsburgh, USA

Westley Weimer University of Michigan Ann Arbor, USA

***Abstract*—From delivering groceries and vital medical supplies to driving trucks and passenger vehicles, society is becoming increasingly reliant on autonomous vehicles (AVs). It is therefore vital that these systems be resilient to adversarial actions, perform mission-critical functions despite known and unknown vulnerabilities, and protect and repair themselves during or after operational failures and cyber-attacks. While techniques have been proposed to address individual aspects of software resilience, vulnerability assessment, automated repair, and invariant detection, there is no approach that provides end-to-end trusted and resilient mission operation and repair on AVs.**

In this paper, we describe our experience of building START,[1] a framework that provides increased resilience, accurate vulnerability assessment, and trustworthy post-repair operation in autonomous vehicles. We combine techniques from binary analysis and rewriting, runtime monitoring and verification, automated program repair, and invariant detection that cooperatively detect and eliminate a swath of software security vulnerabilities in cyberphysical systems. We evaluate our framework using an autonomous vehicle simulation platform, demonstrating its holistic applicability to AVs.

***Index Terms*—resilience, availability, autonomous vehicles, automated program repair**

I. Introduction

Robotic autonomous systems (RAS) are on track to become a lynchpin of modern society, helping to solve problems in both commercial and national defense applications, from home delivery [5] to self-driving cab services [31], [32], to agricultural [71] and remote delivery and rescue operations [2], [69]. RAS typically consist of networked embedded devices that sense and interact with the physical world. This interface to the physical world positions RASs as computing substrates where cyber attacks against them can cause injuries to humans and damage to property [26].

A wealth of research exists concerning the safety of deploying RASs in situations with possible human and material costs [52]. For example, the DARPA High-Assurance Cyber Military Systems program led to formally-hardened embedded systems that are more difficult to hack [23]. However, there is little work to provide both assessments of *trust* (the operator's belief that the vehicle is dependable and the willingness to accept associated risk [55]) and *resilience* (i.e., the ability to safely recover from or avoid errors, attacks or environmental challenges and complete the original mission or a variation thereof) [79], [30]. Overall, while RAS security and trustworthiness is a well-understood problem [37], [75], [15], current efforts focus on identifying and mitigating individual emerging threats rather than providing end-to-end security solutions that collectively harden these systems, and detect and respond to malicious behavior at runtime.

In this paper, we describe our experience adapting and combining both novel and existing techniques to form an end-to-end cohesive security framework, START, that provides trust and resilience using binary hardening, continuous measurement, automated program repair, and invariant detection. We demonstrate START's applicability to the problem of hardening an Uncrewed Aerial Vehicle (UAV) platform. Our goal was to develop an overall security solution that hardens RASs against cyber attacks, detects when cyber attacks may be occurring during operation, and automatically repairs compromised RAS software to restore trustworthy behavior and enable it to continue operating under hostile circumstances.

START combines several components. First, we employ binary analysis and translation to harden off-the-shelf binary programs without requiring access to source code [35], [38]. Such techniques reduce opportunities for attackers to exploit the original software by transforming it in a way that is not easily predicted. Similar binary transformation techniques have enjoyed success on conventional desktop and server platforms; however, they have not yet been applied to embedded RAS

[*]Both authors contributed equally to the work and are the corresponding authors: kevin.leach@vanderbilt.edu and ctimperley@cmu.edu

[1]Software Techniques for Automated Resilience and Trust

platforms, whose resource constraints introduce significant engineering and conceptual concerns.

Second, we employ continuous monitoring of vehicle telemetry and execution observables to assert when the RAS's performance deviates from a defined trusted operating region. While intrusion detection systems (IDSs) are well-studied in the security literature [17], [66], [51], straightforward application of IDSs on telemetry and sensor data is insufficient in our context. Although IDSs can be gainfully used to monitor the hosts and network in the ground control/operation center, the computer onboard a UAV has a non-traditional attack surface and threat model. It is not directly connected to the Internet, but instead connects with ground control over dedicated radio channels for telemetry and control. The software is narrowly focused on moving the vehicle according to mission plan, and depends on specific sensor devices, (e.g., GPS, accelerometer, barometer, etc.), which are not common in general purpose computing systems. We therefore incorporated a runtime monitoring mechanism that verifies if an execution of a mission is safe and faithful, compared to an expected mission model, and reports observed problems or abnormalities.

Third, we leverage automated program repair techniques to dynamically identify and repair vulnerabilities automatically [47], [56], [50]. Automated program repair identifies program test cases that lead to failure, and systematically changes the program's source code until the tests no longer exhibit the original failure, thus providing a restored level of trust in spite of an attack occurring. Such techniques have been widely evaluated in desktop and server contexts, but with significantly less attention paid to embedded environments (e.g., [28], [39], [49], [83], [82]). Our instantiation in START assumes source code is available, but we note that this assumption is not fundamental [67], [29]. Embedded RAS and UAV architectures introduce *significant* resource constraints, a key challenge to typically resource-intensive dynamic program repair. Second, such systems require mechanisms for validating repairs, usually via test cases, which in this context may require simulation or emulation of sensor inputs and other hardware-adjacent concerns. Third, repair techniques have not often been assessed in the context of an end-to-end defense system. Finally, we increase trust in the dynamically-generated patches post hoc by building on existing tools for invariant generation (e.g., [22], [58], [59], [57]) to assess the semantic differences between the original and patched program. Invariant detection has been used in many software engineering applications, and we extend and adapt this existing work to provide trustworthy and resilient UAV operation.

We evaluate our framework using an autonomous vehicle platform, demonstrating its applicability to cyber physical systems. In an end-to-end evaluation of START, our approach successfully defeats 12 out of 14 attack scenarios, designed by a Red Team, and is able to safely complete its mission.

To summarize, we contribute a general framework that:

- Applies binary analysis to transform arbitrary, commercially available RAS software to reduce the likelihood of cyberattack,
- Uses runtime monitoring and verification to detect when a cyber attacker successfully compromises a RAS,
- Leverages automated program repair techniques to deploy patches on a UAV that patch vulnerabilities, and
- Incurs minimal runtime overhead, such tha it can be used live in the field without interruption of system execution.

II. Subject System Overview

This paper describes our experience instantiating START for a particular underlying UAV platform, ArduPilot.[2] In this section, we describe this subject system and our threat model, highlighting how START's components apply in this context.

ArduPilot: ArduPilot is an open source autopilot system (UAS) that has been deployed to over a million vehicles ranging from multirotors and helicopters to ground vehicles and submarines.[3] To demonstrate our approach, we implemented START on an autonomous quadcopter running ArduPilot. ArduPilot handles communications between the vehicle and its operator and interfaces with the vehicle sensors and actuators to perform a statically- or dynamically-defined sequence of actions (i.e., a mission). The vehicle is controlled remotely via MAVLink (Micro Autonomous Vehicle Link) [20], a packed-based communication protocol for exchanging messages, including motion commands and telemetry updates, between the vehicle and a ground control station software (e.g., APMPlanner [6] or QGroundControl [21]). Physical and link-layer communication is achieved via radio telemetry devices, where communications are unencrypted and rely on a System ID number to distinguish multiple vehicles. Communication is unencrypted and relies on a System ID number.

Threat Model: START is designed to detect and recover from attempts to maliciously gain control of a RAS's execution. We assume the attacker has knowledge of the platform and its underlying communication protocols, and a scenario involving with wireless communication with the system (e.g., telemetry), thereby allowing them to send packets to the RAS. The attacker is assumed to use this communication link to send crafted messages to hijack control of the RAS by corrupting memory (e.g., through stack and heap overflows, format string vulnerabilities, and ROP attacks). After gaining access, the attacker can control vehicle behavior (e.g., crash the system, navigate to a new location, report spurious data, etc.).

Overall Approach: Figure 1 illustrates the overall architecture of START. Given the platform we want to secure, we begin with (1) access to the source code (and corresponding binary), (2) a model normal vehicle operation (the Trust Envelope), and (3) an attacker seeking to compromise execution of software on the RAS. First, we transform and harden the original compiled binary (Section III, Step ① in the Figure). Next, we execute the hardened binary on the RAS, and start running a mission (Step ②). While the RAS executes the mission, we continuously monitor sensor, telemetry, and execution trace data to compare it against the predefined Trust

[2]http://ardupilot.org
[3]https://web.archive.org/web/20210225061416/https://ardupilot.org/about

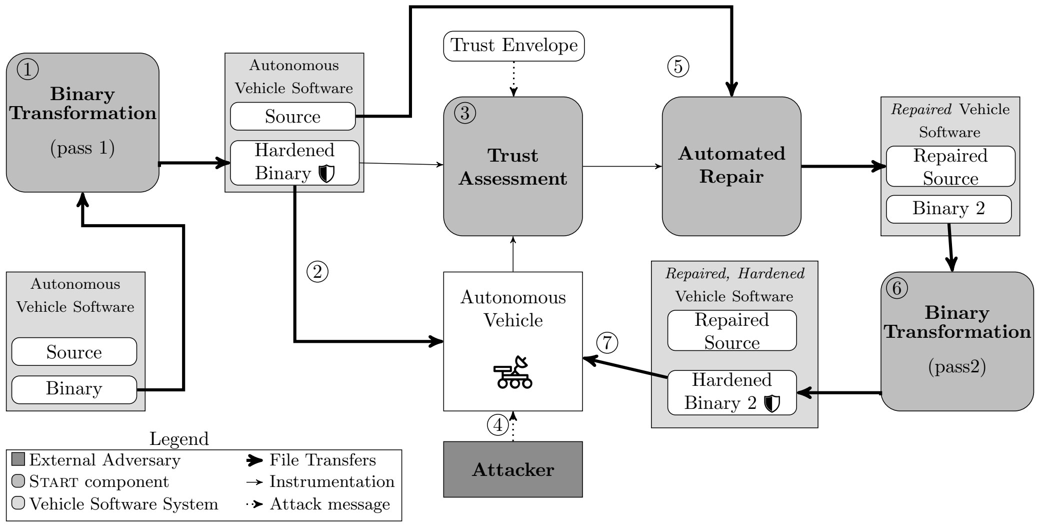

Fig. 1: A high-level overview of the START framework.

Envelope (Step ③, Section IV). This monitoring can detect, for example, a network-based attack that hijacks control of the vehicle (Step ④), leading to divergent vehicle behavior.[4] This resulting analysis is fed to the automatic repair component, which constructs a repaired version of the vehicle software source code that is not vulnerable to the original attack (Section V, Step ⑤). We compile the repaired version of the source code, which we pass back through binary hardening (Step ⑥). This hardened repaired binary is then executed on the RAS, where it is able to complete the original mission while remaining resilient against the original attack (Step ⑦).

Thus, our approach combines several existing techniques together to improve trust and resilience in an ArduPilot system that could be targets of attacks. Critically, START must be able to run on the system itself, allowing it to autonomously respond to and recover from cyber attacks as they occur. That is, we can construct a repair within minutes while the vehicle is online without human intervention. In doing so, we provide a novel approach to applying autonomous healing and recovery to RAS. We describe these tools and techniques in detail in subsequent sections.

Asssumptions and Limitations: Our implementation of START requires source code access to produce repairs and provide certain (but not all) monitoring functionalities. However, source code access is not a requirement of our high-level approach. For example, our automatic repair component could be replaced with an alternative that does not require source code (e.g., [29], [68]).

Upon detecting an attack, START instructs the vehicle to loiter until a repair is found (approximately 10 minutes). Ten minutes may be too long for a consumer drone ($\sim 67\%$ of battery life), but for industrial/military drones or autonomous ground vehicles, 10 minutes may be more feasible.

[4]We assume the attacker will want to compromise execution on the RAS.

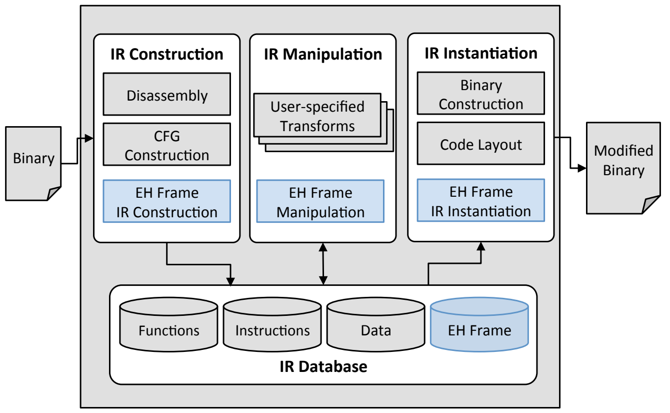

Fig. 2: An overview of the Zipr pipeline for binary analysis, transformation and hardening.

III. Binary Hardening

Prior to the RAS mission (Step ① in Figure 1), START hardens the embedded system binary, increasing its robustness to attack, via the Zipr static binary rewriting infrastructure [86], [33] and its suite of available security transformations plug-ins [34], [60].

While Zipr has been shown robust and effective for securing x86 desktop and server binaries, START sought to evaluate and demonstrate its application to autonomous vehicle software. In this section, we give a brief review of Zipr's algorithms for transforming and hardening software, in particular the compiled RAS control software binary. This process is divided into several phases, shown in Figure 2.

The IR Construction Phase (reverse engineering) accepts a binary program (either an executable or runtime library) and analyzes it to produce an intermediate representation (IR) that is amenable to both analysis and manipulation. Reverse engineering an arbitrary binary with no symbol information (i.e., a stripped binary) is a challenging problem. In general, given access only to a program's machine, it is not always possible to disambiguate between code and data [78]. In practice, most binaries can be reversed engineered with enough precision to allow binary applications to be usefully analyzed and transformed. However, note that higher precision enables more precise transforms which in turn yields better performance and security.

To facilitate various security analyses and transforms, this process calculates the binary's control flow in graph format. This graph is called a control flow graph (CFG), as it is similar in spirit to a typical compiler's CFG, though in Zipr's case it is whole-program instead of per-function. Even under the conservative assumption of perfect disassembly of a program's binary code, constructing a program's CFG from its machine code is still not a straightforward process. There are many complications: indirect control flow, non-return functions, functions that share code, non-contiguous functions and functions with tail calls. In the face of ambiguity, Zipr leverages *address pinning*. Address pinning reserves particular addresses in the program's address space to deal with analysis ambiguity. For example, if it is unclear from analysis whether a particular instruction is used as the target of indirect control flow, that address is reserved for the corresponding instruction and the potential address-generation code is left alone. Thus, if the potential address-generation is used for indirect control flow, the program operates correctly and if if the potential address-generation code is actually generating a particular constant, that code also operates correctly. Consequently, for hardening to operate correctly it is not necessary to precisely determine the set of possible targets for every particular indirect branch instruction; Zipr relies only on the fact that the set of all pinned addresses contains at least all the addresses of possible indirect branch targets in the original program. Further, this mechanism allows Zipr to insert hardening instrumentation before the target of an indirect branch by simply changing the IR's pinned address association to a new instruction.

Along with the CFG, the IR construction phase determines a number of other properties of the binary including function boundaries, the shape of each function's stack frame including incoming arguments, exception handling information, the live registers at each point program point, leaf functions, etc., that are necessary for extensive manipulation and transformation of binaries.

After an intermediate representation of the binary has been produced, we leverage Zipr's transforms in a defense-in-depth strategy by applying the following set of transformations:

- Block-level Instruction Layout Randomization (BILR) randomizes the location of basic blocks.
- Stack Layout Transformation (SLX) adds stack canaries, random padding to the stack frame, and optionally reorders stack variables.
- Heap Layout Randomization (HLX) supports heap-based transformations, including padding and defenses against double-free and user-after-free attacks.
- Global Layout Transformation (GLX) adds canaries and reorders global variables.
- Selective Control-flow Integrity (SCFI) enforces the control-flow graph obtained in the IR Reconstruction step. SCFI is designed to thwart various arc-injection attacks, including return-oriented programming attacks.
- Binary Auto Repair Templates (BinArt) selectively intercepts libc functions and implements repair templates, e.g., when the size of the target buffer in a C string copy operation is known, ensure that the buffer does not overflow and automatically terminate the string.

Finally, Zipr instantiates the transformed IR as a new executable binary. By using different randomization seeds, the final mapping of the IR to an output binary allows for the creation of diversified binaries [33], significantly raising the bar for attacks that rely on intimate knowledge of the binary layout.

Whenever possible, START extends Zipr's functionality with the ability to record diagnosis information emitted by the various Zipr security plugins, e.g. the location of a triggered stack canary, the name of the function to which a repair template has been applied, the address of an illegal jump target, to help identify and localize attacks during the mission.

START also uses Zipr to harden binaries produced by the automatic repair component of our framework in response to detected attacks (Step ⑥ in Figure 1). Further, to diversify the types of attacks that can be identified, START further incorporates runtime monitoring and verification over telemetry and instrumentation, described next.

IV. Runtime Monitoring and Verification

Having produced a hardened binary, START now begins the RAS mission (Step ② in Figure 1). START implements continuous runtime monitoring to identify, and analyze, possible attacks over the course of system operation (Step ③ in Figure 1). This process identifies anomalous behavior by monitoring observables from the software and I/O layers of the RAS, and verifies a priori modeled mission-specific constraints that define a successful operating envelope for the system and a given mission. This process aims to reduce the uncertainty that a mission-operator may have the device's operation, and also raise the level of trust that an operator should have in a device and mission [62].

Our method for assessing trusted operation is organized into two focus areas with considerations for (1) the vehicle's locomotion through physical space and across time, assessed by examining device telemetry (Section IV-A) and (2) the execution behavior and states of onboard software, tracked via instrumentation (Section IV-B).

A. Telemetry Assessment

In general, mission success for a UAV like our target system is defined by timely and accurate autonomous traversal through a set of waypoints defined in the mission plan. START implements a runtime telemetry analysis toolkit for modeling and cross-checking waypoint constraints that are indicative of successful and expected mission traversal. Each waypoint constraint embodies statistical conditions (values or ranges) that are both likely to be observed during a successful mission and highly unlikely to be observed during a failed mission or a flawed controller execution. At runtime, START verifies these constraints by analyzing a stream of telemetry from the UAV, and reports anomalies to the repair module.

We model how the vehicle should approach individual waypoints, align itself with the next waypoint and exit the current waypoint. by measuring positional information such as latitude, longitude, altitude, heading and speed; controller parameters like throttle set-points; and resource state information (e.g., battery voltage) during software-in-the-loop simulations. The models are then constructed via standard statistical learning methods over telemetry from successful and failed training runs, along with compensation weightings for unmeasurables (e.g., tailwind effect). The mission constraints are thus checked by the ground control station at runtime by cross-checking reported telemetry data. By only examining already emitted telemetry, locomotion verification adds no overhead to the device's runtime.

Consequently, locomotion verification can quickly detect and signal the occurrence of suspect behaviors, for example, when the vehicle has trouble aligning with the next waypoint, when it is moving too fast or too slow, or when it is too far from the waypoint for meaningful payload sensor or actuation operations like taking a picture or video. These early warnings can be difficult to visually discern at a distance.

B. Internal Monitoring via Instrumentation

Monitoring telemetry alone is not always sufficient because reported values do not guarantee execution integrity of the onboard software and telemetry output. Failures or software overrun by an adversary can emit spurious or malicious telemetry, masking underlying corrupted values. To address this concern, we further execution observables by instrumenting and monitoring a small set of mission-critical variables and function calls in the vehicle controller (i.e., ArduPilot).

As the first step in our approach, we manually examine the vehicle controller software to identify variables and function calls that are vital to mission success (e.g., handling parameters, sending and receiving communications, handling positional information). We then add source-level instrumentation to these areas of the program and collect representative execution data via simulation. Next, we manually post-process the call graph and data values to identify the structure and associations of regular tasks and ranges of values for data that support I/O. Using this information, we insert constraint checking code code as either scheduled tasks or inline modifications to low-rate task threads of the existing flight stack code. This is also a manual step which requires an expert to minimize the real-time overhead of instrumentation. Finally, at runtime, verification occurs on-board the device and only emits a signal to the repair module if a trust violation is detected. It is technically possible that a clever attacker can bypass or disable the scheduled tasks or instrumentation. However, we argue that the risk is minimal for embedded platforms (e.g., ArduPilot), because it requires an attacker to modify onboard software with new logic through a very small C2 channel—that is, it would produce a highly visible and privileged operation or require physical access. This risk can be mitigated by moving monitoring to a separate, trusted co-processor.

V. Automated Repair of Detected Vulnerabilities

Given an identified attack produced in Step ④ of Figure 1 (detected by either binary hardening or runtime monitoring), START aims to transform the running program to repair the underlying vulnerability. We thus adapt a heuristic program repair [47] paradigm to identify a suitable source-level patch that allows the system to recover from the attack in Step ⑤. At a high level, heuristic program repair assumes a program and a set of passing and failing test cases, where the failing test cases correspond to the bug to be repaired. These techniques aim to identify (typically small) edits to the input program that cause it to pass all tests. Broadly speaking, a typical repair process consists of a combination of *fault localization*, to identify potentially-faulty locations to be edited; *patch generation*, which instantiates candidate repair templates into candidate patches; and *patch validation* for sampling and assessing generated patches.

To automatically repair the RAS controller in response to attacks, START adapts Darjeeling, an existing language-agnostic framework for search-based program repair,[5] to *safely* and *efficiently* repair an embedded system. Below, we describe each of the stages of our repair process, shown in Figure 3.

A. Attack Reproduction and Fault Localization

Repair begins when either the runtime monitor or hardened binary informs the repair module of an attack. The repair module uses data from the mission logs, including the sequence of commands executed by the vehicle, to construct an executable test that safely recreates the attack in simulation.

Given these inputs, the repair module uses a coverage analysis to determine the source code lines that were executed during the attack, and combines that information with its existing coverage information for its regression test (a nominal simulated execution of the predetermined mission without the attack) to determine the likely location of the fault via spectrum-based fault localization [43], [61].

B. Patch Generation

After determining suspicious areas of the program, a set of candidate patches are generated at those locations by applying a heuristic set of repair operators. We use the same *statement-level* repair operators as GenProg [46], AE [80], and RSRepair [80], and others. These operators are intended to be generic enough to fix arbitrary bugs: Statements may be deleted, swapped, replaced, or appended, where compatible statements from the same file are used as ingredients when replacing and appending.

[5]https://github.com/squaresLab/Darjeeling

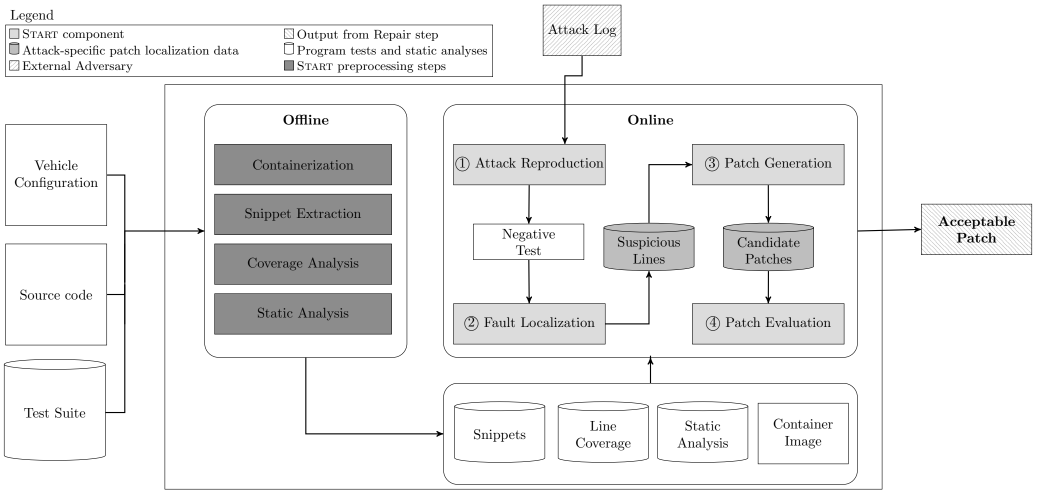

Fig. 3: An overview of START's automated repair process.

C. Safe Patch Evaluation

Finally, the repair module exhaustively searches for a *plausible repair*, which causes the program to pass all tests, including the existing test suite and the recreated attack. To decrease the cost of evaluation, we abort the evaluation of a candidate patch on its first test failure and uses test prioritization to reduce the cost of rejecting patches. The evaluation process continues until a resource limit is reached (e.g., wall-clock time) or a human operator signals that a plausible patch has passed its trust evaluation (Section V-E).

Safety is particularly important in this domain, to ensure that the execution of arbitrary code introduced by candidate patches does not interfere with the safe operation of the vehicle. To prevent such interference with minimal compromise to efficiency, we use containerization to evaluate candidate patches within a sandboxed simulated environment.

D. Efficiency

Efficiency is a paramount concern in this context. Efficiency is required to ensure that an acceptable repair is found in a timely manner, before the battery is depleted or failure of the mission. Furthermore, the repair process must run on an embedded platform with minimal resources and no connection to the internet. To allow START to rapidly respond to attacks, we perform the following:

- We lift parts of the repair process, including a coverage analysis of the nominal mission execution (without the attack), container preparation, and a set of static analyses, to an *offline* preprocessing step. This stage (or parts of it) is executed each time the source code or configuration for the vehicle is changed before the start of a mission.
- We use a Clang-based plugin to pre-compute a set of static analyses for the program, and build a database of donor code snippets. We then use the results of these analyses during repair to identify and prune redundant program transformations [80].
- To reduce the cost of evaluating candidate patches by several orders of magnitude (measured by time), we combine *low-fidelity simulation* and *clock acceleration*. Motivated by the recent finding [76] that many robotics failures do not rely on complex environmental conditions and can be reproduced in a low-fidelity simulation, we use a lightweight software-in-the-loop (SITL) simulator. Using a lightweight simulator reduces compute and memory usage, and allows multiple candidate patches to be safely evaluated in parallel using separate containers. Additionally, accelerating the simulated clock further decreases the time taken to evaluate a candidate patch.

E. Trust

Heuristic program repair can (and does, in this context) produce numerous plausible patches that defeat the immediate attack and allow the mission to continue. However, some may introduce unintended side effects that are not surfaced during patch evaluation (i.e., patches are not guaranteed to be optimal). Without additional test cases, it is difficult to measure and ensure patch quality [72].

To achieve both trust and resilience, we temporarily deploy the first plausible patch to ensure mission continuity while we present alternative patches to the developers for review. Since our approach can produce dozens of patches within a short period of time, manually reviewing each of them places a large burden on the developer. Our approach exploits a popular invariant detection tool, Daikon [22], to considerably reduce this manual effort by partitioning patches into a smaller number of classes based on their run-time behavior.

Specifically, we collect trace data from executions of both the original and patched programs. We then use Daikon to infer likely invariants, in the form of function pre- and post-conditions, for each of these alternatives.

VI. EVALUATION

A primary contribution of this work is the unified evaluation of a combination of component techniques in the context

TABLE I: Summary of end-to-end evaluation scenarios and results. **Bin**, **Mon**, and **Rep** denote our binary hardening, runtime monitoring, and repair components. “Some” indicates that the attack was defeated in at least one, but not all, runs; “All” indicates that the attack was defeated in all runs.

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

of uncrewed autonomous vehicles. We evaluate START in two parts. First, we use an autonomous vehicle simulator to help develop indicative end-to-end cyber attack scenarios, focusing on the types of cyber attacks our system can prevent and recover from (Section VI-A). Second, we individually measure the runtime performance overhead incurred by each of the binary transformation (Section VI-B), intrusion detection (Section VI-C), and automated repair (Section VI-D).

A. End-to-end Evaluation

To assess the effectiveness of START, a Red Team seeded 14 unique vulnerablities, listed in Table I, into our case study system together and created accompanying attack scenarios to exploit each vulnerability in software-in-the-loop simulation.

Each vulnerable scenario is coupled with a MAVLink packet that remotely triggers the vulnerability during SITL execution against a predefined mission, which is a series of predefined waypoints. We evaluated these attacks within a virtualized environment to parallelize as many scenario executions as possible. We ran the virtualized hosts on a server configured with vSphere ESXi 6.0 and hosting 14 Ubuntu 16.04 virtualized hosts concurrently, each enumerating all subsets of defenses for end-to-end evaluation. This virtual infrastructure allowed for the enumeration of all scenarios with demonstrative vulnerabilities within the SITL environment to determine whether the specific vulnerability under evaluation is (1) detected, (2) repaired, and (3) results in mission success. Each scenario was repeated ten times to identify any non-determinism and measure reliability.

Table I shows results. We find that: Binary hardening detects or mitigates (i.e., the vulnerability no longer applies to the transformed binary) the attack in 8 out of 14 scenarios, Runtime monitoring detects the attack for all 14 scenarios. We are able to automatically repair the underlying vulnerability in 11 scenarios. In total, START is able to defeat the attack and successfully complete the mission for 12 of the 14 scenarios.

B. Binary Hardening Performance

In the interest of generalizability, we evaluated the overhead of the Zipr baseline platform. While it might be tempting to evaluate the overhead on the actual UAV software, accurately benchmarking it is difficult. The UAV software spends much of its time idle, and thus timing is not a practical way to measure overhead. Further, the software is not prepacked with representative inputs, making any attempt to benchmark it inherently bias by its input selection. Thus, to help measure overheads, we leverage the SPEC CPU2006 benchmark suite [73], a suite containing a wide variety of real-world programs written in C, C++ and Fortran. The SPEC benchmarks, while older and desktop oriented, include many programs that have made it onto UAV software stacks, such as the *bzip2* and *perlbmk* programs. Further, the suite contains performance and correctness tests which we can leverage for unbiased results.

We compiled the binaries using `gcc`, `g++`, and `gfortran` version 4.8.4 with `-O2` optimization level. Note that we exclude `dealII` from the benchmark as it does not build correctly at any optimization level, and we compile `perl` and `wrf` at a lower optimization level to ensure that the program works correctly before we apply any rewriting.

The transformed binaries successfully pass their associated regression tests. Figure 4 provides Zipr’s baseline (i.e., only the BILR transformation enabled) performance and file size overhead results normalized to the original binaries (i.e., less than 1.0 indicates speedup/space decrease and greater than 1.0 indicates slowdown/size increase). On average, Zipr incurs performance and filesize overheads of 3% and 30%, respectively.

Most transformations add effectively no or very minimal performance or file size overheads, namely SLX, HLX and GLX. The overhead implications of BinArt depend on how heavily the relevant functions are used, but for our use case, we observed no noticeable difference in performance in the UAV control software. SCFI adds the most notable overhead, averaging approximately 10% slowdown. Thus, we feel that START’s binary hardening approach provides a low-overhead technique suitable for hardening a safety-critical UAV control software stack.

C. Runtime Monitoring Performance

In this section, we describe the performance of our runtime monitoring-based intrusion detection system, in terms of its telemetry-based locomotion verification and controller software instrumentation and monitoring.

To evaluate the overhead of START’s telemetry checking, we measured CPU and memory usage of the ground control system (GCS) across 11 back-to-back simulated mission executions on a four-core, hyperthreaded Intel processor with 32 GB of memory (a style of machine commonly used as a GCS). Across those runs, we observed a steady state CPU and memory utilization of 2% and 1% respectively. That is, the runtime overheads of telemetry checking were negligible.

We also performs on-board runtime monitoring and periodic verification of the controller state via manual software in-

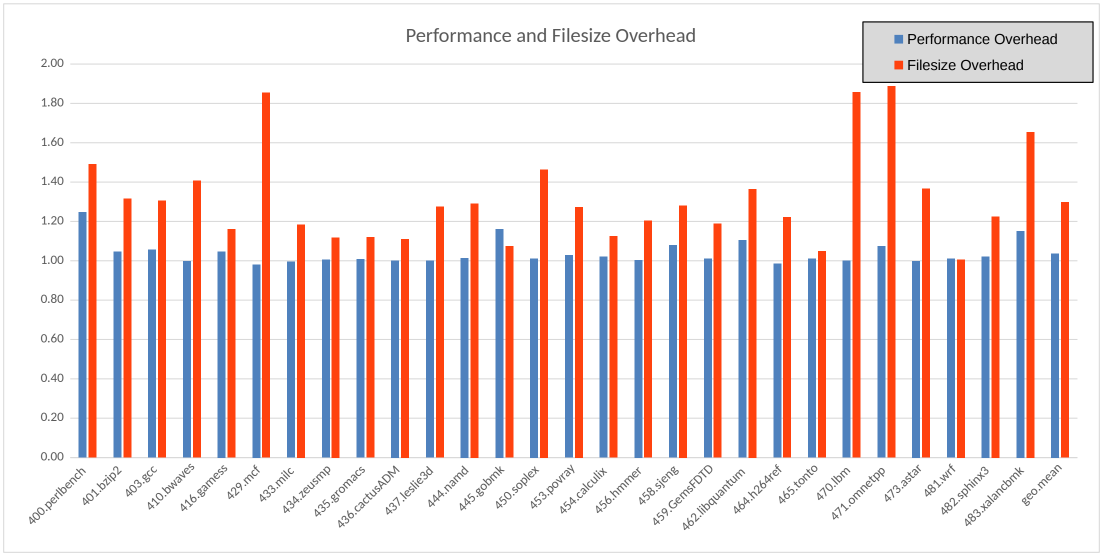

Fig. 4: Performance and file size overhead incurred by Zipr on SPEC CPU2006 Benchmarks. Results are normalized against native execution speed and original binary filesize.

TABLE II: Instrumentation overhead for verifying access to flight parameters.

Minimum	Maximum	Spread	Average
$15\mu s$	$252\mu s$	$237\mu s$	$33\mu s$

strumentation (see Section IV). Table II summarizes profiling results that examine the overhead and spread for periodic verification of ArduPilot parameters. To gather these measurements, 80 controller parameters were periodically cross-checked against their boot-time values by a separately scheduled verification task. The cross-checking task is scheduled at 50 ms and compares a block of read-only memory containing boot-time parameters against dynamically configurable values on the controller.

Across thousands of timed samples, we observed the minimum, maximum, and average latencies shown in Table II. In practice, the average overhead of 33 μs has no visible effect on a live device at runtime. While the listed overhead is excellent, it is narrowly focused on one type to state check at a certain size of data. Logically, such checks may be extended to support verification where the target data size of the check is small enough to fit in a 50 ms window with similar results, e.g., examining waypoint mission plans or checksumming select blocks of memory.

In the Red Team exercises, we complemented the task-based approach described above with a strict form of embedded logical assertion to guard against invalid modification of waypoint settings. In this configuration, a single reference counter was “woven” around the expected access pattern to modify mutable state (e.g., an operator sends a mission plan update message), and a pre-condition check was set at the point of receiving the command and a post-condition check was used to unlocked access to the waypoint update operation assuming the precondition was met. In terms of overhead, this approach introduces a single global variable and two basic blocks to control access to the waypoint state. Overhead during such an execution is negligible.

In terms of network overhead stemming from periodic verification of execution behavior and runtime states of onboard software, runtime monitoring introduces a single periodic and fixed-size MAVLink message as part of the trust verification process. This message is reported every 50 ms, and for most of the time, serves only as a heartbeat indicating that the monitor is capable of reporting. In the case that a trust violations needs to be reported, select flags are set within the message, but the message size does not increase, thus keeping overhead low.

Finally, in terms of quantifying the instrumentation needed to support runtime verification, our experience is that the actual amount of code inserted is minimal (e.g., storing a set of values at boot and periodic condition checking, computing MD5 checksums, etc.). However, the real overhead is the expert analysis of the controller software that is needed to identify high-value low-overhead instrumentation points.

D. Automated Program Repair Performance

We measured the effectiveness of our repair approach across eight Red Team bug scenarios in terms of (a) the wall-clock time taken to find the first patch that defeats the attack, and (b) the total number of patches found within a 15-minute window, shown in Table III. Together, these metrics tell us whether the repair module allows the vehicle to resume its mission within a reasonable period of time (before the battery dies), and give us insight into the number of alternative patches that are presented as long-term options to the developers. Across all of the repaired scenarios, the median time taken by the search to discover a patch is 72 seconds. For all scenarios, repair was able to discover multiple acceptable patches within the

TABLE III: A summary of repair performance metrics. Vulnerability describes the type of vulnerability that is exploited by each scenario. TFP gives the time taken to discover the first acceptable patch. # Patches specifies the total number of acceptable patches that were found within the time limit.

Scenario	Vulnerability	TFP (secs.)	# Patches
1	Use after free	38	17
5	Heap overflow	98	10
6	Stack overflow	65	18
9	Infinite loop	79	3
10	Seg. fault	40	6
11	Logic	201	6
13	Integer error	35	17
14	Floating point exception	297	6

TABLE IV: A breakdown of the time spent on each stage of the repair process.

Step	Duration (s)
Docker	120
Static Analysis	200
Test Suite Coverage	35
Fault Localization	40
Search	72
Recompilation	60

allocated time limit: the median number of patches produced is 8.

Table IV presents a breakdown of the end-to-end time taken to discover an acceptable repair. Both the online and offline steps were performed on the same machine, a Ubuntu 18.04 LTS system with 64GB RAM, an AMD2990WX with 32 cores at 4.2GHz, and 1 TB of NVMe SSD storage. The cost of the online steps is directly linked to the simulation speedup factor that is used: using a larger speedup factor allows the bug to be localized faster and increases the number of candidate patches that can be evaluated within the given time frame.

Patch quality: Although we were able to find a plausible repair for each bug scenario within 15 minutes, not all of these repairs are equally desirable. To illustrate, we examine two such undesirable patches.

Figure 5 provides an example of a low-quality patch for Scenario 5, a heap buffer overflow in the code responsible for handling parameter requests. The vulnerability is exploited by the attacker by issuing a request to set the value of a non-existent parameter during the mission. The low-quality patch resolves the vulnerability by indirectly disabling the vehicle's ability to respond to parameter modification requests. Since the exploit is not triggered, all of the test cases pass and the patch is accepted. Although this patch prevents the exploit, it may impede the completion of the mission.

Figure 6 provides an example of an undesirable patch for Scenario 13. This scenario contains a divide-by-zero vulnerability that is triggered by sending a navigation request with certain parameters to the vehicle. The example patch addresses the vulnerability by removing the statement at which the divide-by-zero error occurs. The removed statement is

```
  void GCS_MAVLINK::handle_param_set(
      mavlink_message_t *msg)
  {
      mavlink_param_set_t packet;
-     mavlink_msg_param_set_decode(msg, &packet);
      enum ap_var_type var_type;
```

Fig. 5: An example of a low-quality patch that was discovered for Scenario 5.

```
  d = d - 1;
- e = c / d;
+ this->send_message(MSG_EKF_STATUS_REPORT);
```

Fig. 6: An example of a patch for Scenario 13 that fixes the bug (second line) while introducing undesired but otherwise benign functionality (third line).

replaced with a donor statement taken from elsewhere in the same file. The donor statement causes the program to send an status report to the ground control station whenever the attacker attempts to exploit the vulnerability. This unwanted, albeit benign, change to the program is not rejected since the program satisfies the oracle: all waypoints in the mission are visited and the vehicle ends the mission in an expected state. As long as the changes do not interfere with the completion of the mission, the oracle will permit them.

Using our patch invariant analysis, we are able to reduce the number of patches that are presented to the developer in these scenarios by between 40 and 45%. By doing so, we significantly reduce the burden of assessing patch quality and finding an acceptable long-term patch that should be applied to the vehicle.

The end-to-end process of computing an invariant set for a given patch produced by the repair tool took approximately three minutes using a single thread. This overhead is largely due to the runtime overheads incurred by the instrumentation necessary to collect execution traces. In practice, this process can be parallelized by spreading candidate patches across separate threads.

VII. Related Work

Binary Analysis and Transformation: There are many tools that can harden software. Compile-time and OS-enabled load-time transforms can reliably transform software [13], [1], [63]. Binary rewriting, both static and dynamic, can also transform and protect software [70], [9], [53], [7], [34], [24]. However, understanding hardening transformations at various times and comparing individual techniques is well-studied and beyond the scope of this paper [33], [34], [60].

Instead, this paper focuses on a practical use for hardening transformations in the context of RASs, and is thus distinct from the vast body of prior work related to compiler-based hardening and binary-rewriting issues. In essence, this work is a practical application of the application information repository idea [81], where information collected by a static binary rewriter is stored for run-time monitoring and automatic repair when a fault is detected during a mission.

Runtime Monitoring and Verification: Prior works [25], [18], [62], [11], [14], [65] have used models of vehicle dynamics and input/output and software execution profiles of UAV controllers to formulate the conditions to be verified at runtime during deployment. These works have also shown that select classes of violations (e.g., unplanned modifications to a controller's task scheduler) may be detected before any sign of failure is visible to operator.

A wide variety of intrusion and anomaly detection systems have also been proposed as a means of inferring and enforcing runtime monitors for cyberphysical systems (e.g., [3], [4], [36], [12], [41]). While many of these approaches infer monitors that generalize across multiple missions, they often do so at the cost of permitting a greater number of false positives. For START, we exploit the fact that missions are predetermined to build a reliable statistical model of the vehicle's locomotion. Given the negligble runtime overhead of our prototype implementation, additional monitoring techniques (such as those described above) could be added to START in the future to further enhance its capacity to identify and explain attacks.

Program Repair: For our prototype implementation of START, we devised a repair approach based on the GenProg, AE, and RSRepair family of search-based repair tools [47], [64], [80]. Many other search-based repair approaches have been proposed with various repair operators [49], [45], [42], algorithms [8], [16], [19], [77], and optimizations [29], [82], [39]. Most of these approaches can be integrated into START's high-level framework and low-level infrastructure.

Beyond search-based repair, other approaches to automatic program repair include semantics-based repair [56], [44], [85], which typically uses symbolic execution to obtain a specification for the repaired program before using program synthesis to construct a patch that satisfies that specification, and data-driven repair, which uses neural networks and statistical methods to generate and prioritize patches [10], [54], [48]. Techniques have also been proposed specifically for repairing security vulnerabilities [28], [40].

In the interest of urgency and ensuring mission completion, our approach applies the first plausible patch to the vehicle to allow it to keep running. We then rely on a human operator to address the problem of determining which patch should be applied to the vehicle in the long-term. To reduce the burden of inspecting multiple patches, we use an invariant analysis to partition patches based on behavioral differences. A number of other approaches to the problem of assessing and ensuring patch quality have been proposed: For example, using antipatterns to exclude likely bad patches from consideration [74], checking if the patched program crashes on additional inputs generated via fuzzing [27], and measuring similarity between the patched and original program on test outputs and executions [84]. These techniques could be incorporated into START to automatically exclude certain patches from consideration and to further reduce the burden of manual patch inspection.

VIII. CONCLUSION

Society's growing interest in robotic and autonomous systems, combined with documented system vulnerabilities and software failures, necessitates new solutions in system resiliency and trustworthiness. In this paper, we presented START, a general framework for securing RASs by improving resilience, vulnerability assessment, and trusted post-repair operation. We combined techniques from binary analysis and rewriting, machine learning, automated program repair, and invariant detection to cooperatively detect and repair indicative software security vulnerabilities in these systems.

To demonstrate START, we implemented our approach on a simulated quadcopter and evaluated its ability to detect and overcome attacks through a Red Team exercise. START's combined defenses allowed the vehicle to continue its mission for 12 out of 14 scenarios. By applying lightweight hardening transformations to the binary of our subject system, we are able to either immediately defeat the attack (i.e., the attack is no longer able to exploit the vulnerability) or otherwise identify that an attack has occurred and allow the program to safely terminate. For the 2 scenarios that we were unable to defeat, our runtime monitor was still able to identify that an attack was occurring, thereby allowing a human operator to potentially respond. The runtime overheads imposed by binary monitoring and runtime monitoring are minimal and, most importantly, have no observable effect on the safe and timely operation of the vehicle (e.g., missing deadlines).

By adapting existing search-based repair techniques, we are able to produce a repair for 11 of the 14 scenarios. Due to the hard time and resource limitations of operating on an embedded device, we immediately apply the first plausible patch discovered by repair to the vehicle. As a result of this choice, certain non-essential functionality may be disabled as part of the repair. While this outcome is less desirable than directly patching the underlying vulnerability, it is nonetheless valuable within our particular operating context as it renders the attack inert and allows the mission to continue. To inform the construction of a long-term solution to the vulnerability, we allow the repair process to produce and present alternative patch options to the developers. To reduce the burden of considering and comparing multiple plausible patches, we group patches based on their behavioral differences as determined by an analysis of their likely invariants.

While repairs were generated within a reasonable window of time during our evaluation (less than 10 minutes), there are opportunities to further reduce the time and resources necessary to generate an effective repair, thereby allowing our approach to be used in a wider variety of deployments. Most notably, when an attack is detected by the runtime monitor or the transformations applied to the binary, the repair module is alerted but very few details about the attack are shared. A more sophisticated approach may take inspiration from vulnerability repair (e.g., ExtractFix [28] and Senx [40]) to use information about the type of vulnerability and state of the execution (e.g., the program stack) to enhance the fault localization and inform the construction of candidate patches.

Acknowledgment

This research was funded by AFRL (#FA8750-15-2-0075); the authors are grateful for their support. Any opinions, findings, or recommendations expressed are those of the authors and do not necessarily reflect those of the US Government.

References

[1] Abadi, M., Budiu, M., Erlingsson, Ú., and Ligatti, J. Control-flow integrity. In *Conference on Computer and Communications Security* (2005), CCS '05, pp. 340–353.

[2] Ackerman, E. Royal Mail Is Doing the Right Thing With Drone Delivery. https://spectrum.ieee.org/delivery-drones-uk, May 2022.

[3] Afzal, A., Le Goues, C., and Timperley, C. Mithra: Anomaly Detection as an Oracle for Cyberphysical Systems. *Transactions on Software Engineering* (2021).

[4] Aliabadi, M. R., Kamath, A. A., Gascon-Samson, J., and Pattabiraman, K. ARTINALI: dynamic invariant detection for cyber-physical system security. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2017), ESEC/FSE '17, pp. 349–361.

[5] Amazon. Amazon Prime Air. https://www.amazon.com/Amazon-Prime-Air.

[6] ArduPilot Development Team. ArduPilot. https://github.com/ArduPilot.

[7] Bauman, E., Lin, Z., and Hamlen, K. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In *Annual Network and Distributed System Security Symposium* (2018), NDSS '18.

[8] Bian, Z., Blot, A., and Petke, J. Refining Fitness Functions for Search-Based Program Repair. In *International Workshop on Automated Program Repair* (2021), APR '21, pp. 1–8.

[9] Bruening, D., Garnett, T., and Amarasinghe, S. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization* (2003), CGO '03, pp. 265–275.

[10] Chen, Z., Kommrusch, S., Tufano, M., Pouchet, L.-N., Poshyvanyk, D., and Monperrus, M. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *Transactions on Software Engineering 47*, 9 (2019), 1943–1959.

[11] Choi, H., Lee, W.-C., Aafer, Y., Fei, F., Tu, Z., Zhang, X., Xu, D., and Deng, X. Detecting Attacks Against Robotic Vehicles: A Control Invariant Approach. In *Conference on Computer and Communications Security* (2018), CCS '18, pp. 801–816.

[12] Choi, H., Lee, W.-C., Aafer, Y., Fei, F., Tu, Z., Zhang, X., Xu, D., and Xinyan, X. Detecting attacks against robotic vehicles: A control invariant approach. In *Conference on Computer and Communications Security* (2018), CCS '18, pp. 801–816.

[13] Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., and Hinton, H. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium* (1998), vol. 98, pp. 63–78.

[14] Dash, P., Li, G., Chen, Z., Karimibiuki, M., and Pattabiraman, K. PID-Piper: Recovering Robotic Vehicles from Physical Attacks. In *International Conference on Dependable Systems and Networks* (2021), DSN '21, pp. 26–38.

[15] De La Torre, G., Rad, P., and Choo, K.-K. R. Driverless vehicle security: Challenges and future research opportunities. *Future Generation Computer Systems* (2018).

[16] de Souza, E. F., Le Goues, C., and Camilo-Junior, C. G. A novel fitness function for automated program repair based on source code checkpoints. In *Genetic and Evolutionary Computation Conference* (2018), GECCO '18, pp. 1443–1450.

[17] Denning, D. E. An intrusion-detection model. *Transactions on Software Engineering*, 2 (1987), 222–232.

[18] Desai, A., Dreossi, T., Seshia, S. A., Lahiri, S., and Reger, G. Combining model checking and runtime verification for safe robotics. In *Runtime Verification* (2017), pp. 172–189.

[19] Ding, Z. Y., Lyu, Y., Timperley, C., and Le Goues, C. Leveraging Program Invariants to Promote Population Diversity in Search-Based Automatic Program Repair. In *International Workshop on Genetic Improvement* (2019), GI '19, pp. 2–9.

[20] Dronecode Project. MAVLink: Micro air vehicle message marshalling library. https://github.com/mavlink/mavlink.

[21] Dronecode Project. QGroundControl. http://qgroundcontrol.com.

[22] Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming 69*, 1-3 (2007), 35–45.

[23] Fisher, K. HACMS: High Assurance Cyber Military Systems. In *High Integrity Language Technology* (2012), HILT '12, pp. 51–52.

[24] Flores-Montoya, A., and Schulte, E. Datalog disassembly. In *USENIX Security Symposium* (2020), pp. 1075–1092.

[25] Florian-Michael, A., Faymonville, P., Finkbeiner, B., Schirmer, S., Torens, C., Lahiri, S., and Reger, G. Stream runtime monitoring on uas. In *Runtime Verification* (2017), pp. 33–49.

[26] Gander, K. Drone delivering asparagus to Dutch restaurant crashes and burns. https://www.independent.co.uk/life-style/food-and-drink/news/drone-delivering-asparagus-to-dutch-restaurant-crashes-and-burns-10179731.html, April 2015.

[27] Gao, X., Mechtaev, S., and Roychoudhury, A. Crash-avoiding program repair. In *International Symposium on Software Testing and Analysis* (2019), ISSTA '19, pp. 8–18.

[28] Gao, X., Wang, B., Duck, G. J., Ji, R., Xiong, Y., and Roychoudhury, A. Beyond tests: Program vulnerability repair via crash constraint extraction. *Transactions on Software Engineering and Methodology 30*, 2 (2021), 1–27.

[29] Ghanbari, A., Benton, S., and Zhang, L. Practical program repair via bytecode mutation. In *International Symposium on Software Testing and Analysis* (2019), ISSTA '19, pp. 19–30.

[30] Haimes, Y. Y. On the definition of resilience in systems. *Risk Analysis 29*, 4 (2009), 498–501.

[31] Hawkins, A. A day in the life of a Waymo self-driving taxi. https://www.theverge.com/2018/8/21/17762326/waymo-self-driving-ride-hail-fleet-management, August 2018.

[32] Hawkins, A. Uber approved to restart self-driving tests in Pennsylvania. https://www.theverge.com/2018/12/18/18147114/uber-self-driving-test-approved-restart-pennsylvania, December 2018.

[33] Hawkins, W. *Static Binary Rewriting to Improve Software Security, Safety, and Reliability*. PhD thesis, University of Virginia, 2018.

[34] Hawkins, W., Hiser, J. D., Nguyen-Tuong, A., Davidson, J. W., et al. Securing binary code. *Security & Privacy 15*, 6 (2017), 77–81.

[35] Hawkins, W. H., Hiser, J. D., Co, M., Nguyen-Tuong, A., and Davidson, J. W. Zipr: Efficient Static Binary Rewriting for Security. In *International Conference on Dependable Systems and Networks* (2017), DSN '17, pp. 559–566.

[36] He, Z., Chen, Y., Huang, E., Wang, Q., Pei, Y., and Yuan, H. A system identification based oracle for control-cps software fault localization. In *International Conference on Software Engineering* (2019), ICSE '19, pp. 116–127.

[37] Highnam, K., Angstadt, K., Leach, K., Weimer, W., Paulos, A., and Hurley, P. An uncrewed aerial vehicle attack scenario and trustworthy repair architecture. In *International Conference on Dependable Systems and Networks—Industrial Track* (2016).

[38] Hiser, J., Nguyen-Tuong, A., Hawkins, W., McGill, M., Co, M., and Davidson, J. Zipr++: exceptional binary rewriting. In *Workshop on Forming an Ecosystem Around Software Transformation* (2017), pp. 9–15.

[39] Hua, J., Zhang, M., Wang, K., and Khurshid, S. Towards Practical Program Repair with On-Demand Candidate Generation. In *International Conference on Software Engineering* (2018), ICSE '18, pp. 12–23.

[40] Huang, Z., Lie, D., Tan, G., and Jaeger, T. Using safety properties to generate vulnerability patches. In *Symposium on Security and Privacy* (2019), SP '19, pp. 539–554.

[41] Jiang, H., Elbaum, S., and Detweiler, C. Inferring and monitoring invariants in robotic systems. *Autonomous Robots 41*, 4 (2017), 1027–1046.

[42] Jiang, J., Xiong, Y., Zhang, H., Gao, Q., and Chen, X. Shaping Program Repair Space with Existing Patches and Similar Code. In *International Symposium on Software Testing and Analysis* (2018), ISSTA '18, pp. 298–309.

[43] Jones, J. A., and Harrold, M. J. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Automated Software Engineering* (2005), pp. 273–282.

[44] Le, X.-B. D., Chu, D.-H., Lo, D., Le Goues, C., and Visser, W. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Joint Meeting on Foundations of Software Engineering* (2017), FSE '17, pp. 593–604.

[45] Le, X.-B. D., Lo, D., and Le Goues, C. History Driven Program Repair. In *International Conference on Software Analysis, Evolution, and Reengineering* (2016), SANER '16.

[46] Le Goues, C., Dewey-Vogt, M., Forrest, S., and Weimer, W. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering* (2012), ICSE '12, pp. 3–13.

[47] Le Goues, C., Nguyen, T., Forrest, S., and Weimer, W. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering 38*, 1 (2012), 54–72.

[48] Li, Y., Wang, S., and Nguyen, T. N. Dlfix: Context-based code transformation learning for automated program repair. In *International Conference on Software Engineering* (2020), ICSE '20, pp. 602–614.

[49] Liu, K., Koyuncu, A., Kim, D., and Bissyandé, T. F. TBar: Revisiting template-based automated program repair. In *International Symposium on Software Testing and Analysis* (2019), ISSTA '19, pp. 31–42.

[50] Long, F., and Rinard, M. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (2015), ESEC/FSE '15, pp. 166–178.

[51] Loukas, G., Vuong, T., Heartfield, R., Sakellari, G., Yoon, Y., and Gan, D. Cloud-based cyber-physical intrusion detection for vehicles using deep learning. *Access 6* (2018), 3491–3508.

[52] Lozano-Perez, T. *Autonomous robot vehicles*. Springer Science & Business Media, 2012.

[53] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Notices 40*, 6 (2005), 190–200.

[54] Lutellier, T., Pham, H. V., Pang, L., Li, Y., Wei, M., and Tan, L. Coconut: combining context-aware neural translation models using ensemble for program repair. In *International Symposium on Software Testing and Analysis* (2020), ISSTA '20, pp. 101–114.

[55] Lyons, J. B., Stokes, C. K., Eschleman, K. J., Alarcon, G. M., and Barelka, A. Trustworthiness and it suspicion: An evaluation of the nomological network. *Journal of Human Factors* (2011), 219–229.

[56] Mechtaev, S., Yi, J., and Roychoudhury, A. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *International Conference on Software Engineering* (2016), ICSE '16, pp. 691–701.

[57] Nguyen, T., Dwyer, M. B., and Visser, W. SymInfer: inferring program invariants using symbolic states. In *Automated Software Engineering* (2017), pp. 804–814.

[58] Nguyen, T., Kapur, D., Weimer, W., and Forrest, S. Dig: A dynamic invariant generator for polynomial and array invariants. *Transactions on Engineering and Methodology 23*, 4 (2014).

[59] Nguyen, T., Kapur, D., Weimer, W., and Forrest, S. Using dynamic analysis to generate disjunctive invariants. In *International Conference on Software Engineering* (2014), ICSE '14, pp. 608–619.

[60] Nguyen-Tuong, A., Melski, D., Davidson, J. W., Co, M., Hawkins, W., Hiser, J. D., Morris, D., Nguyen, D., and Rizzi, E. Xandra: An Autonomous Cyber Battle System for the Cyber Grand Challenge. *Security & Privacy 16*, 2 (2018), 42–51.

[61] Parnin, C., and Orso, A. Are automated debugging techniques actually helping programmers? In *International Symposium on Software Testing and Analysis* (2011), pp. 199–209.

[62] Paulos, A., Pal, P. P., Clark, S. S., Usbeck, K., and Hurley, P. Trusted mission operation — concept and implementation. In *Runtime Verification* (2017), pp. 416–423.

[63] PaX Team. PaX address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt.

[64] Qi, Y., Mao, X., Lei, Y., Dai, Z., and Wang, C. The Strength of Random Search on Automated Program Repair. In *International Conference on Software Engineering* (2014), ICSE '14, pp. 254–265.

[65] Quinonez, R., Giraldo, J., Salazar, L., Bauman, E., Cardenas, A., and Lin, Z. SAVIOR: Securing Autonomous Vehicles with Robust Physical Invariants. In *USENIX Conference on Security Symposium* (2020), SEC '20.

[66] Roesch, M., et al. Snort: Lightweight intrusion detection for networks. In *Lisa* (1999), vol. 99, pp. 229–238.

[67] Schulte, E., DiLorenzo, J., Forrest, S., and Weimer, W. Automated repair of binary and assembly programs for cooperating embedded devices. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (2013), ASPLOS '13, pp. 317–328.

[68] Schulte, E., Weimer, W., and Forrest, S. Repairing COTS Router Firmware without Access to Source Code or Test Suites: A Case Study in Evolutionary Software Repair. In *International Workshop on Genetic Improvement*, GI '15.

[69] Schwartz, D. Why Drones Are the Future of Outdoor Search and Rescue. https://www.outsideonline.com/outdoor-adventure/exploration-survival/drones-search-rescue, October 2021.

[70] Scott, K., and Davidson, J. Safe virtual execution using software dynamic translation. In *18th Annual Computer Security Applications Conference, 2002. Proceedings.* (2002), pp. 209–218.

[71] Sheikh, K. A Growing Presence on the Farm: Robots. https://www.nytimes.com/2020/02/13/science/farm-agriculture-robots.html, February 2020.

[72] Smith, E. K., Barr, E. T., Le Goues, C., and Brun, Y. Is the cure worse than the disease? overfitting in automated program repair. In *Joint Meeting on Foundations of Software Engineering* (2015), FSE '145, pp. 532–543.

[73] Standard Performance Evaluation Corporation. SPEC CPU® 2006, 2017.

[74] Tan, S. H., Yoshida, H., Prasad, M. R., and Roychoudhury, A. Anti-patterns in search-based program repair. In *International Symposium on Foundations of Software Engineering* (2016), FSE '16, pp. 727–738.

[75] Thing, V. L., and Wu, J. Autonomous vehicle security: A taxonomy of attacks and defences. In *International Conference on Internet of Things and Green Computing and Communications and Cyber, Physical and Social Computing and Smart Data* (2016), IEEE, pp. 164–170.

[76] Timperley, C. S., Afzal, A., Katz, D., Hernandez, J. M., and Le Goues, C. Crashing simulated planes is cheap: Can simulation detect robotics bugs early? In *International Conference on Software Testing, Validation and Verification* (2018), ICST '18, pp. 331–342.

[77] Timperley, C. S., Stepney, S., and Le Goues, C. An Investigation into the Use of Mutation Analysis for Automated Program Repair. In *International Symposium on Search Based Software Engineering* (2017), vol. 10452 of *SSBSE '17*, pp. 99–114.

[78] Wartell, R., Zhou, Y., Hamlen, K. W., Kantarcioglu, M., and Thuraisingham, B. Differentiating code from data in x86 binaries. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases* (2011), Springer, pp. 522–536.

[79] Weimer, W., Forrest, S., Kim, M., Le Goues, C., and Hurley, P. Trusted software repair for system resiliency. In *International Conference on Dependable Systems and Networks Workshops* (2016), pp. 238–241.

[80] Weimer, W., Fry, Z. P., and Forrest, S. Leveraging program equivalence for adaptive program repair: Models and first results. In *International Conference on Automated Software Engineering* (Nov. 2013), ASE '13, pp. 356–366.

[81] Williams, D., Hiser, J. D., and Davidson, J. W. Using program metadata to support sdt in object-oriented applications. In *Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems* (2009), pp. 55–62.

[82] Wong, C.-P., Santiesteban, P., Kästner, C., and Le Goues, C. VarFix: balancing edit expressiveness and search effectiveness in automated program repair. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2021), ESEC/FSE '21, pp. 354–366.

[83] Xin, Q., and Reiss, S. P. Leveraging syntax-related code for automated program repair. In *International Conference on Automated Software Engineering* (2017), ASE '17, pp. 660–670.

[84] Xiong, Y., Liu, X., Zeng, M., Zhang, L., and Huang, G. Identifying patch correctness in test-based program repair. In *International Conference on Software Engineering* (2018), ICSE '18, pp. 789–799.

[85] Xuan, J., Martinez, M., Demarco, F., Clement, M., Marcote, S. L., Durieux, T., Le Berre, D., and Monperrus, M. Nopol: Automatic repair of conditional statement bugs in java programs. *Transactions on Software Engineering 43*, 1 (2016), 34–55.

[86] Zephyr Software. Zipr Toolchain. https://git.zephyr-software.com/opensrc/zipr/.